Advanced Edge Deployment: Abstracting Cyber-Physical Models via FMU Mastery

Fanping Bu[1] Mikalai Filipau[1] Nikolay Baklanov[1]

[1]Integrated Productivity and Conveyance Center, SLB, USA,

{fbu2, mfilipau, nbaklanov2}@slb.com

Abstract

Deploying cyber-physical models at the edge or in the cloud as software components is the key step of model-based-design. Depending on run-time environment, an extensive customization often needs to be made. To streamline and facilitate the deployment of models and simulators in production, a unified framework is developed. The implementation utilizes functional mockup units (FMUs) as the executable binary for the models and JavaFMI as the simulation engine. Each model deployment is encapsulated inside a microservice with all the software dependencies, with communication realized through RabbitMQ. A generalized approach to manage the model namespace has been implemented, ensuring that the FMU executor remains agnostic to changes in both model and application, as long as the AsyncAPI specification includes a mapping of the model's input-output space to the protocol's topics. Two examples are presented to illustrate the convenience and effectiveness of the proposed framework: a winch controller at the edge for oil and gas wireline operation and a wireline logging unit simulator in the Azure DevOps pipeline for software-in-the-loop testing.

Keywords: FMU, Edge, Wireline, Oil&Gas, FMI, Cyber-Physical Systems, Deployment, Microservices

1 Introduction

Rapidly evolving edge computing prioritizes convenience of deployment of advanced physical models designed for real-time control applications and data processing. The shift from monolithic software architecture to microservices has been facilitated by containerization tools like Docker and Kubernetes, which allow isolation of applications into distinct environments, thereby enhancing the scalability and manageability aspects via smaller and independently deployable services.

The workflow requires careful handling of parameters, inputs, and outputs. Dealing with unique namespaces is a part of a larger challenge - the need to manually adjust naming conventions and identifiers for each functional mockup unit (FMU) import, which significantly complicates the deployment process. We present a solution of using an interim Java layer to abstract the FMU's namespace that addresses the "at the edge integration" challenge by standardizing the interface between the FMUs and the microservices architecture.

Before discussion of specifics of our proposed framework, it is essential to provide an overview of the current state of the art. This will contextualize our work within the broader landscape of this technology block and highlight gaps and opportunities that our approach aims to address. The papers analyzed below stress the complexity and challenges involved in FMU integration and deployment, especially when FMUs from different tools form a single simulation environment.

The functional mockup interface (FMI) has been instrumental in advancing interoperability and integration within the modeling and simulation community (Gomes et al. 2018; Blochwitz et al. 2011). Multi-year efforts from various cross-domain institutions have explored diverse FMI applications. One of the earliest studies (Chen et al. 2011) introduces a generic FMU interface for Modelica for enhanced reusability and interoperability within the OpenModelica framework for multiple instances of an imported FMU. While this approach effectively facilitates FMU integration and connection within the designated simulator engine, it lacks interoperability extension to a wider range of modeling environments. The work by Cabral et al. (2018) explores FMI applications in industrial automation by enabling co-simulation (Gomes et al. 2018) per the IEC 61499 standard for distributed systems, which facilitates the virtual commissioning process by allowing co-simulation of physical plants and their PLC-based control by elevating mapping of internal variables, parameters and inputs/outputs between IEC 61499 models and the FMI. Despite its contributions to Industry 4.0 automation via paying great attention to correlation between the inter-standard data types, this research does not scale up deployment scenarios and model types and thus avoids the context of cloud and edge computing.

The co-simulation FMU-proxy framework (Hatledal et al. 2019) achieves language and platform independence using a remote procedure call (RPC) technique in a client-service architecture and offers FMU discovery. The solution significantly contributes to collaborative modeling and heterogeneous simulation expanding array of previously unsupported languages and on incompatible platforms. However, it primarily focuses on co-simulation and intellectual property protection and does not address the emerging need for flexible and scalable model deployments, such as microservice-based architectures. A recent work (Juhlin et al. 2022) breaks long-standing lack of interoperability at the system level and presents a cloud-enabled simulation platform for drive-motor-load systems using asset administration shells[1] (AAS) and FMUs. This approach significantly enhances the flexible deployment of asset models in complex simulations by leveraging containerization. However, it does not fully exploit the potential of microservices for heterogeneous applications, as it relies on a more rigid RESTful API server architecture.

A noteworthy paper by Stüber and Frey (2021) presents a cloud-native simulation as a service (SIMaaS) implementation utilizing FMUs for co-simulation, leveraging the FMPy[2] framework. This implementation is realized as a microservice in the form of a RESTful API. In our development, though, we have identified that JavaFMI[3] is a more performant alternative (Hatledal et al. 2018). Our generalized approach for model namespace management ensures that the FMU executor remains agnostic to model-application mapping changes, with much less restrictions on FMU parametrization as offered in the analyzed paper. Furthermore, our study showcases a practical, real-time, complex industrial automation systems example (Segura et al. 2023), offering a significant advancement in solution integration over the SIMaaS demo.

2 Concepts

The method described below abstracts models from Simulink or other modeling environments, enhancing workflow efficiency and user-friendliness from inception to Dockerized edge deployment using a microservices, RabbitMQ, Linux VM, Kubernetes, and Rancher ecosystem. The revealed methodology utilizes FMI and streamlines model's input/output space, to suit better microservice deployments outside the original, often Windows-based, software ecosystem. The FMI concept, combined with our mapping explained below, ensures that abstracted and vectorized models maintain their functional integrity and ease integration with various computational environments. The JavaFMI engine enables cross-platform configuration and execution of models, regardless of the originating modeling tool and allows for scalable complexity. The presented technology extends beyond the edge, allowing physical models to be embedded as FMU objects in web applications and cloud platforms or even be invoked via command line interface during quick prototyping.

In our approach, we introduce a novel concept of model anonymization[4], which allows integrators to use the model without needing prior knowledge of the exact namespace of its inputs and outputs. We employ I/O vectorization specification and an inter-system mapping layer on top of FMU, which generalizes the interface and allows flexible interaction. This approach simplifies the integration process and also somewhat obfuscates sensitive details, while enhancing flexibility and ease of integration in complex systems. Application teams can now work with standardized interface definitions focusing on a single or a limited set of specification files, such as YAML for AsyncAPI/OpenAPI or similar, instead of navigating through specific I/O names. This technique is particularly beneficial in environments where model reusability and interoperability are paramount, providing a seamless cross-platform method for deploying and interacting with models in real-world physical systems.

The abstraction of models from Simulink or other model-based design tools for Dockerized edge deployment using FMI is still an emerging concept, particularly in the context of Kubernetes on Linux VM platforms. This area appears to be underexplored in the current literature, highlighting the avenue for our work. While the use of FMU/FMI for model exchange and co-simulation is well-adopted (Modelica Association 2022), the specific edge deployments in containerized environments is less discussed (Schranz et al. 2021).

Abstracting models to enhance workflow efficiency and deploying them as microservices on the edge, while preserving functional integrity through vectorized mapping of I/O specifications, introduces a less explored approach, particularly for the oil and gas industry, and when integration with "system of systems" architecture solutions is considered.

3 System Design and Implementation

In this section, system architecture and software stacks chosen to enable easy deployable software for control and simulation at edge or cloud are introduced. First, a microservice based system architecture is described in Section 3.1. Section 3.2 provides descriptions on how we encapsulate the designed model in an FMU and how the FMU is executed with a JavaFMI library. Section 3.3 illustrates how interfaces are defined among different microservices with AsyncAPI, and data (or messages) are shared among different microservices. Section 3.4

[1] A central concept in the context of Industry 4.0, the digital representation of an asset.
[2] https://pypi.org/project/FMPy/
[3] https://bitbucket.org/siani/javafmi/src/master/

[4] Not widely mentioned in literature in the context of Modelica, Simulink, or other modeling languages. The anonymization concept for hiding model's namespace particularities seems has little to no explicit discussions.

describes how a JavaFMI-based wrapper and RabbitMQ message passing are combined and executed inside a Docker container-based microservice. Finally, the development process of modeling and simulation microservice, and how we can utilize DevOps pipeline to automate it, is presented in Section 3.5.

3.1 System Architecture

The microservice-based system architecture is depicted in Fig. 1. An FMU, combined with a JavaFMI-based wrapper, is encapsulated within a Docker container-based microservice. Data sharing and communication among different microservices is realized through the open-source message-broker software, RabbitMQ.

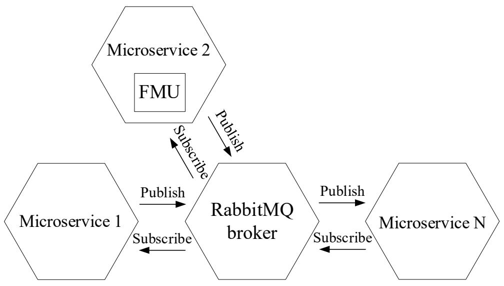

Figure 1. System architecture.

Microservice-based architecture has many advantages and is popular for both cloud and edge development. Docker container-based microservices package all the runtime dependencies and thus can be deployed across different platforms. At a high level, developers can choose appropriate modeling software tools for the development of physical system models, plants, or control algorithms. If the selected software tool supports FMU export, the exported FMU can be plugged into the proposed framework for cloud or edge deployment. In a previous iteration of our framework, we directly employed a binary Linux shared library generated by Simulink to represent the system model and control algorithm. This approach constrained development to Matlab/Simulink and tied it to a specific version of the software. With FMU, the tool selection is more flexible if it conforms to the standard.

3.2 FMU Wrapper with JavaFMI

As shown in Fig. 2, the resulting core of the modeling simulation microservice is a wrapper dealing with binary inside an FMU. There are many existing libraries that can run FMU simulations. We adopted JavaFMI for its fast execution capabilities to meet the real-time requirements of our field application deployments.

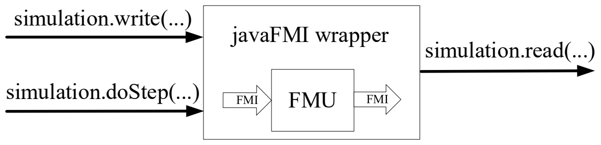

Figure 2 JavaFMI wrapper interface.

JavaFMI library is a suite of components to interact with the FMU interface. The FMU wrapper is a key component for easy access to the FMU models in co-simulation mode. It provides access to the abstract simulation class with a set of methods to interact with inputs, outputs, and parameters. A typical algorithm of application to run an FMU includes:

- Creation of simulation class with a pointer to the FMU file: *Simulation simulation = new Simulation("path/to/file.fmu")*.
- Initialization with a start time and, optionally, an end time using *simulation.init (startTime, stopTime)* method.
- Writing parameters with *simulation.write* method.
- Updating inputs in a loop with desired update rate.
- Updating tunable parameters in a loop.
- Running one simulation step in a loop with the *simulation.doStep (stepSize)* method.
- Reading outputs in a loop with the *simulation.read* method.
- Resetting or terminating the FMU simulation.

Since the co-simulation mode is used, FMU has its internal sampling rate, defined during the FMU compilation stage. Method *simulation.doStep (stepSize)* requires *stepSize* time to be a result of multiplying the internal sample time by an integer number. To achieve continuous model simulation, the process involves reading inputs, updating tunable parameters, running the FMU synchronously with a specified step size, and generating outputs, as illustrated in Fig. 3.

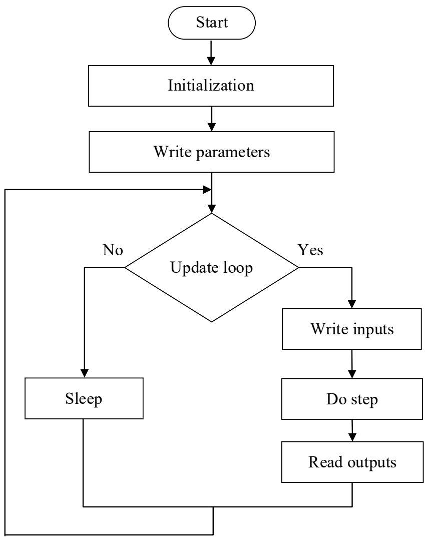

Figure 3 JavaFMI-based Java wrapper execution loop.

3.3 RabbitMQ and AsyncAPI

For proper execution and results exchange during the active phase of FMU simulation, communication with other microservices is essential. To fulfill this requirement, we have adopted RabbitMQ, an open-source message broker software. Initially designed to implement the advanced message queuing protocol, RabbitMQ has evolved through a plug-in architecture to support additional protocols such as the streaming text-oriented messaging protocol and MQ telemetry transport.

To define the specific content and format (schema) of messages exchanged among different services, AsyncAPI is used for RabbitMQ communication. The schema may reference other files for additional details or shared fields, but it is typically a single, primary document that encapsulates the API description. Furthermore, the AsyncAPI schema acts as a communication contract between receivers and senders within an event-driven system. It specifies the payload content required when a service sends a message and offers clear guidance to the receiver regarding the message's properties.

3.4 Modeling and Simulation Microservices

Modeling and simulation microservices are created by combining JavaFMI-based execution wrapper for FMU simulation and RabbitMQ communication with other microservices. The integration code is implemented in a Java loop, as shown in Fig. 4.

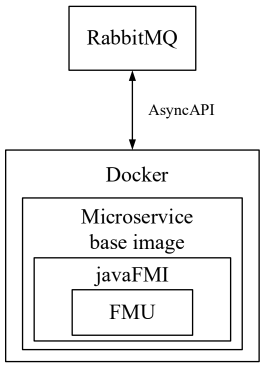

Figure 4 Modeling simulation microservice structure.

During the initialization phase, information about inputs, outputs, and parameters are extracted from the FMU's *ModelDescription.xml* to compare against the AsyncAPI payload schema. All inputs and outputs are mapped to periodically updated “state” topics to reproduce continuous input/output signals. Tunable parameters are mapped to “configuration” topics updatable by request using RPC calls; for example, to achieve the anonymization of a model, we map the I/O of an arbitrary FMU to the AsyncAPI schema referring to the *ModelDescription.xml* content. The internal structure, calculation algorithms, and the origin of the FMU will not impact RabbitMQ communication and other services. The modeling and simulation microservice with the FMU receives messages with inputs and parameters and sends messages with outputs from the model.

An example of the definition for an input port of a model in AsyncAPI properties is shown below. In this example, topic “*model.input.command.v1*” contains a link to the “*input_port”* model input of array type with four elements.

```
channels:
 model.input.command.v1:
  subscribe:
   summary: Model input topic.
   message:
    $ref: "#/components/messages/model_input"

 schemas:
  model_input:
   type: object
   required:
    - input_field
   properties:
    input_field:
     type: array
     minItems: 4
     maxItems: 4
     items:
      type: number
```

x-units: NA
x-inport: "input_port"
x-initialvalue: [1.0, 1.0, 1.0, 1.0]

With a defined AsyncAPI document, it is easy to find the corresponding "*input_port*" in ModelDescription.xml and update the *"input_port"* of FMU with the values in the message received through RabbitMQ.

The main update loop runs at a defined update rate and is shown in Fig. 5 below.

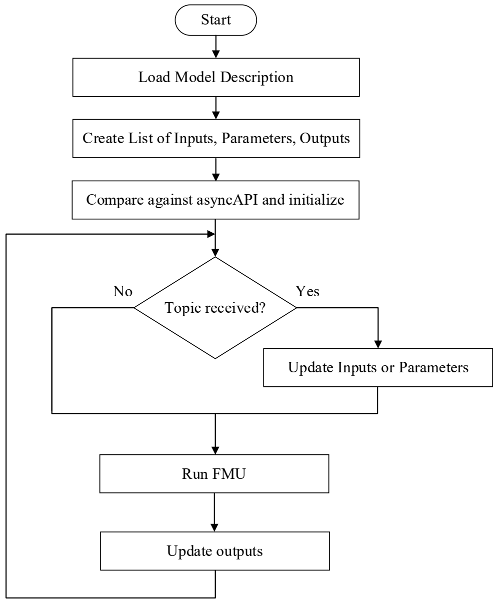
Start
Load Model Description
Create List of Inputs, Parameters, Outputs
Compare against asyncAPI and initialize
No
Topic received?
Yes
Update Inputs or Parameters
Run FMU
Update outputs

Figure 5 Modeling simulation microservice loop.

The microservice code is responsible for

- Handling received configuration messages and updating tunable parameters of the FMU by request.
- Handling received state messages and updating inputs according to the update rate and, in this way, mapping sampled time series input signal to the predefined input of the FMU.
- Executing the FMU by running the *simulation.doStep (stepSize)* method.
- Updating RabbitMQ message topics mapped to the outputs of FMU with desired update rate.
- Providing service information and statistics as periodic state messages: model time and model states (running/stopped/paused).
- Handling received control messages to start, stop, pause, or reset the FMU.

3.5 Microservice Development Process for Modeling and Simulation

Developers can choose any preferred software modeling tools to develop models for physical systems or control algorithms. FMUs can then be exported. Modeling and simulation microservice can be built with following steps, as shown in Fig. 6:

- Access is added to the FMU utilizing wrapper with the JavaFMI library.
- Custom microservice implementation provides an interface to the AsyncAPI and synchronization of FMU execution and API state messages to and from the microservice.

The last stage is deployment of the microservice as a Docker container as a part of complex software application.

All steps in building the pipeline may be automated and integrated into DevOps pipeline (for instance, Azure DevOps) with automated integration tests and API validation, providing safe and robust application deployment.

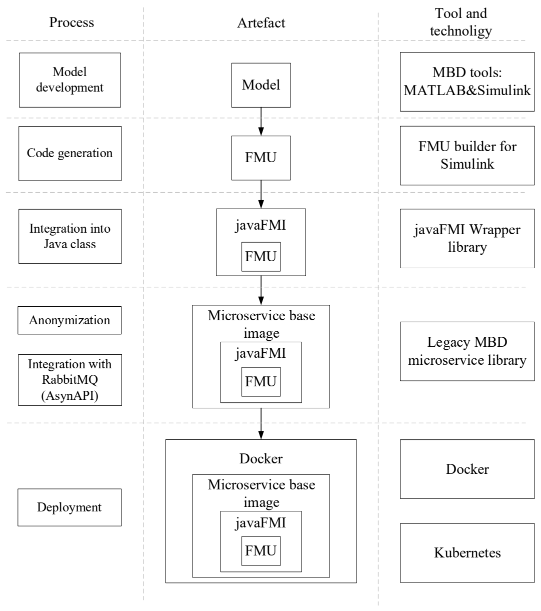

Figure 6 Modeling simulation microservice development process.

4 Case Studies

To illustrate the effectiveness, convenience, and versatility of the proposed framework in real production software deployment, wireline automation development for assisted conveyance in oil and gas operations at SLB is used as an example. Two design cases are presented. In the cloud, a wireline winch and cable simulator is deployed in the Azure DevOps pipelines for software-in-the-loop testing. At the edge, a winch controller is deployed for an automatic winch control. Although the application environments are different between cloud and edge, the same framework can be used due to the portability of the FMU and microservice.

Wireline operation is widely used in the oil and gas industry to measure the properties of a formation using electronic instruments. Fig. 7 illustrates a typical wireline logging operation. A drum of electric cable is driven by a hydraulic winch with a toolstring packed with different formation measurement sensors attached at the free end of the cable. The winch drum is rotated by a hydraulic motor that moves the toolstring up and down along the wellbore. Sensors packaged inside the toolstring conduct sensing measurements while moving and send back measurement results through the connected cable. During operation, an operator is required to control the hydraulic winch manually so that the toolstring movement will follow a desired motion profile.

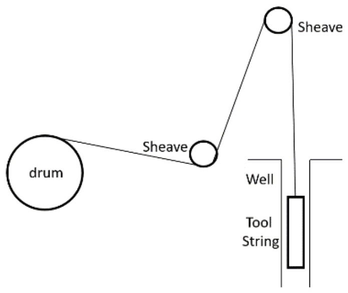

Figure 7 Wireline logging operation for oil and gas industry.

4.1 Winch Simulator

The hydraulic winch drives the drum via a gear transmission. As shown in Fig. 8, the hydraulic winch is a hydrostatic transmission system consisting of a variable displacement pump, a variable displacement motor, and a charge pump. The pumps are driven by a vehicle engine through gears. The drum is driven by variable displacement motor through transmission gears.

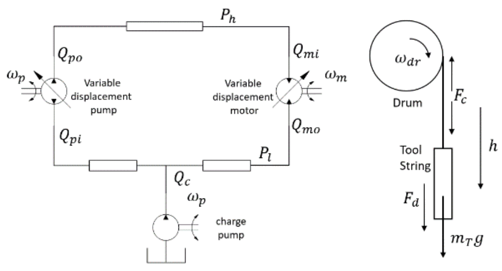

Figure 8 Schematics of a hydraulic winch.

The winch simulator model is developed using the Simulink/Simscape package from MathWorks. The hydraulic circuit is modeled using components from the MathWorks' fluids library and is derived from the hydrostatic transmission example[5] shown in Fig. 9.

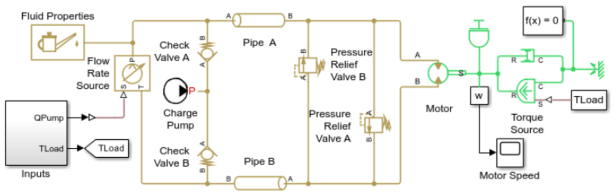

Figure 9 An exemplary model of a hydraulic circuit.

During wireline operation, the released cable can be over thousands of feet long. Therefore, it is necessary to model the dynamic effect of cable elasticity. The entire cable is discretized into serialized mass-spring-damper blocks, as shown in Fig. 10.

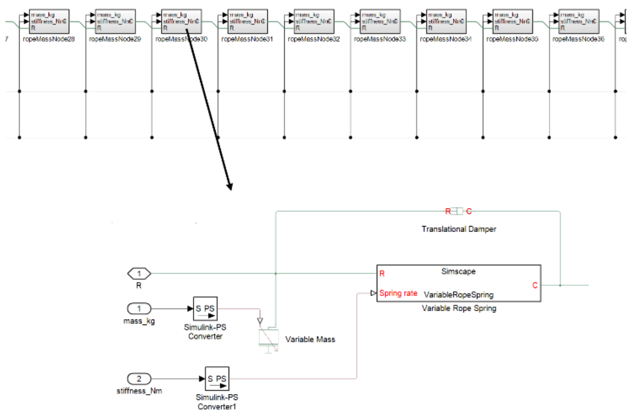

Figure 10 Mass-spring-damper node and serialized cable model.

Calibration tests are conducted to collect data and identify system parameters. The calibrated system response matched actual system behavior, as shown in Fig. 11.

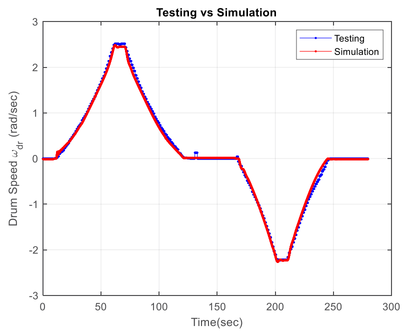

Figure 11 Simulation vs testing data.

5 https://www.mathworks.com/help/hydro/ug/hydrostatic-transmission.html

Following the procedure described in previous sections, FMU is exported from the Simulink/Simscape model and packaged into a microservice. The resulting simulator microservice was used for the software-in-the-loop testing of the winch controller and other software services of wireline automation. Software developers with no knowledge of modeling software such as Matlab/Simulink/Simscape could easily incorporate the simulator microservice into their test need to mimic hydraulic winch behavior if they follow the interface specified in AsyncAPI. Those tests can be made automatic and regressive and can run in Azure DevOps pipeline as is done today for wireline automation development in SLB. Other than software testing, the FMU-based simulator microservice can be deployed in the cloud or at the edge as the core of digital twin applications for predictive maintenance, operation planning, and optimization.

4.2 Winch Controller

As the first step toward automation of the wireline logging operation, it is necessary to control the hydraulic winch, or toolstring motion, following a desired motion profile, automatically, without operator intervention. A nonlinear model based adaptive robust controller (ARC) is designed for this purpose. The detailed controller design can be found in Bu (2020). The designed controller is constructed in Simulink, as shown in Fig. 12.

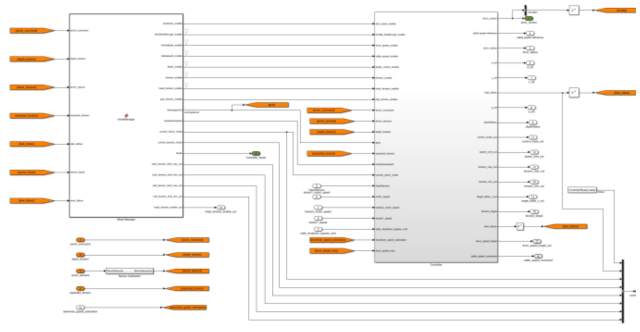

Figure 12 Simulink diagram of winch controller.

The winch controller developed in Simulink can be exported to FMU and packaged into the microservice the same way as the winch simulator in previous sub-section. The winch controller can be deployed via two scenarios, as shown in Fig. 13:

- Software-in-the-loop testing is executed in the Azure DevOps pipeline in the cloud together with the winch simulator as a virtual winch. In this case, a software "switch" will map winch simulator inputs/outputs to the proper RabbitMQ messages. From the winch controller point of view, it is receiving sensor inputs, and sending out actuator commands, from/to the real hardware.
- In the "edge at wellsite" application, the winch controller microservice is deployed at the edge, namely at the automation server physically installed inside the wireline logging unit at a wellsite. The software "switch" will map real sensors and actuator signals from the hardware interface microservice to the proper RabbitMQ messages. The winch controller will be able to control the actual winch drum for automation purposes.

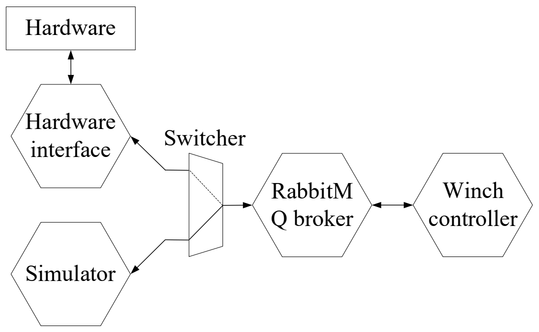

Figure 13 Winch controller deployments.

It should be noted that by adopting a model-based-design process, the winch controller has been matured to the product level and deployed in SLB's wireline logging units globally, completing over several million feet in automated conveyance services.

5 Conclusions

Our work presents a versatile unified framework that is not bound to specific platforms able to generate FMUs, expanding FMU support to a broad range of environments beyond Simulink, Modelica, and similar systems. It covers the deployment needs of arbitrary multidomain simulation engines. By fusing FMUs, JavaFMI, and an AsyncAPI-based anonymization layer[6], we provide a standardized platform for co-simulation and verification suitable for industrial automation systems and well beyond, demonstrated with real-world O&G oil and gas applications. This could be applied, but is not limited to, advanced wireline conveyance assistance systems comparable to automotive ADAS (MathWorks 2024; Dassault Systemes 2024) and autonomous driving functionalities.

Our microservice-based architecture is both flexible and scalable, serving the needs of both edge and cloud model

[6] not limited to, can be a RESTful API wrapping

deployment, promoting collaboration and efficiency in composite asset engineering. The presented comprehensive approach enhances interoperability and streamlines the model delivery process in production software. It allows the automated wrapping and execution of highly arbitrary models at the small cost of introduction of a very thin model-to-RabbitMQ I/O mapping.

Our results demonstrate that the proposed solution enables scalable, flexible, and practical modular deployment of models as software components in cyber-physical and control systems, with a particular focus on Docker and Kubernetes for real-world commercial products in modern computing environments (Segura et al. 2023). Cross-platform model wrapping, configuration, and execution enable the reuse of models in various deployment scenarios.

Our future work focuses on scalability and performance optimization for large-scale deployments, enhancing security measures for edge and cloud, and integrating alternative communication protocols for broader interoperability. The introduced solution utilizes the FMI 2.0 standard, and we are moving onto FMI 3.0, which lets us naturally reduce restrictions for data types. Additionally, we are exploring automated mechanisms for model updating and versioning, with deeper DevOps pipeline integration.

Acknowledgements

The authors would like to thank Ken Ditlefsen (SLB) for his initial “abstract executor” architecture and non-FMU code implementation, which we developed and elevated to the solution presented in this paper. We also express our appreciation to MathWorks for providing consultancy services for their tools, which significantly contributed to the success of our project.

References

Blochwitz, Torsten et al. (2011). “The Functional Mockup Interface for Tool independent Exchange of Simulation Models”. In: *the 8th International Modelica Conference,* 105-14. Dresden, Germany. DOI: 10.3384/ecp11063105

Bu, Fanping. (2020). "Nonlinear adaptive robust motion control for hydraulic winch in oil and gas wireline operation." In: *21st IFAC World Congress*, 8991-96. Berlin, Germany. DOI: 10.1016/j.ifacol.2020.12.2015

Cabral, J. et al. (2018). "Enable Co-Simulation for Industrial Automation by an FMU Exporter for IEC 61499 Models." In: *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, 449-55. Turin, Italy. DOI: 10.1109/ETFA.2018.8502654

Chen, Wuzhu et al. (2011). "A Generic FMU Interface for Modelica." In: *the 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, 19-24. Zurich, Switzerland.

Dassault Systemes. 2024. "Dymola". URL: https://www.3ds.com/products/catia/dymola.

Gomes, Cláudio et al. (2018). "Co-Simulation: A Survey", In: *ACM Computing Surveys*, 51: 1-33. DOI: 10.1145/3179993

Hatledal, L. I. et al. (2018). "FMI4j: A Software Package for working with Functional Mock-up Units on the Java Virtual Machine." In: *The 59th Conference on Simulation and Modelling (SIMS 59)*. Oslo, Norway. DOI: 10.3384/ecp1815337

Hatledal, L. I. et al. (2019). "FMU-proxy: A Framework for Distributed Access to Functional Mock-up Units." In: *the 13th International Modelica Conference*. 79-86. Regensburg, Germany. DOI: 10.3384/ecp1915779

Juhlin, P. et al. (2022). "Cloud-enabled Drive-Motor-Load Simulation Platform using Asset Administration Shell and Functional Mockup Units." In: *2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA)*, 1-8. Stuttgart, Germany. DOI: 10.1109/ETFA52439.2022.9921678

MathWorks. (2024). 'What Is ADAS? 3 things you need to know'. URL: https://www.mathworks.com/discovery/adas.html.

Modelica Association. (2022). "Functional Mock-up Interface for Model Exchange and Co-Simulation. Version 3.0 " Modelica Association.
URL: https://fmi-standard.org/docs/3.0.1/

Schranz, Thomas et al. (2021). "Portable runtime environments for Python-based FMUs: Adding Docker support to UniFMU." In: *14th Modelica Conference 2021*. 419-424 Linköping, Sweden. DOI: 10.3384/ecp21181419

Segura, J., Tran, V.V., Meirkhan, J. et al. (2023). "Autonomous Slickline and Wireline Conveyance Improves Performance of Offshore Interventions". Paper presented at the SPE Offshore Europe, Aberdeen, Scotland, 5–8 September. SPE-215586-MS. DOI: 10.2118/215586-MS

Stüber, Moritz, and Georg Frey. (2021). "A Cloud-native Implementation of the Simulation as a Service-Concept Based on FMI." In: *14th Modelica Conference*, 393-402. Linköping, Sweden. DOI: 10.3384/ecp21181393