MCX — An Open-Source Framework for Digital Twins

*Sajad Shahsavari, Eero Immonen, and Mohammed Rabah
Computational Engineering and Analysis (COMEA)
Turku University of Applied Sciences
20520 Turku, Finland
Email: *sajad.shahsavari@turkuamk.fi

Mohammad-Hashem Haghbayan, Juha Plosila
Department of Computing
University of Turku (UTU)
20500 Turku, Finland

KEYWORDS

Digital twin, Cyber-physical systems, on-line simulation

ABSTRACT

This article describes *ModelConductor-eXtended* (*MCX*), which is an open-source software architecture for digital twins. The *MCX* framework facilitates co-execution of, and asynchronous data communication between, physical systems and their digital simulation models. *MCX* supports running FMUs (simulation models packaged according to the FMI specification) as well as machine learning models and customized models. We propose extensions to the previously published *ModelConductor* framework for higher performance and better scalability. The extensions include decoupling of the queue and the model computation module, utilization of a standard data transmission protocol and implementation of the facility to run time-consuming simulation models in a time synchronous manner. Additionally, three new validation case studies are presented. A performance evaluation shows that the extensions improve the average response time almost 4 times in three specific experiments.

I INTRODUCTION

I-A Background

The cyber-physical systems of the celebrated fourth industrial revolution — so-called *digital twins* (DT), i.e. accurate numerical simulation models operated alongside their physical counterparts — are projected to constitute the backbone of modern industrial automation (see Colombo, Karnouskos, Kaynak, Shi, and Yin (2017); He, Chen, Dong, Sun, and Shen (2019)). While the promise of numerical simulation and system modeling has traditionally been in saving time and money during *product development*, today, digital twins can extend far into *product operations* through proliferation of ubiquitous wireless communication. There are, however, a number of practical challenges in co-execution of physical and digital assets, which is the concern of the present article.

Many examples of digital twin simulation models interacting in real-time with a physical device have been reported in the literature. For example, in *predictive maintenance*, real-time measurement data from a physical system is processed in a simulation model for predicting potential damage in the physical system, should the prevailing situation or the trend continue (see e.g. de Azevedo, Araújo, and Bouchonneau (2016) for discussion on wind turbine applications). On the other hand, with the advent of 5G technology and cloud, edge, fog and mist computing, digital twins also facilitate *remote control*, whereby only minimal data acquisition and safety circuitry reside onboard the physical device, and model-based control is calculated on the digital twin (see e.g. Lee, Suh, Kwak, and Han (2020) for discussion on remote drone control). In spite of this progress, it seems that practical applications of digital twins are still designed, implemented and operated on a case-by-case basis.

The big promise of digital twins for the future is, ultimately, in them providing *full system autonomy*: That any design and/or adaptation of control mechanism for the physical system would be first optimized and implemented on the digital simulation model and then be transferred verbatim to the physical device. To realize this, a flexible *software framework* for running a physical system and its digital twin simulation model (or a collection thereof) seamlessly alongside each other is required. Perhaps somewhat surprisingly, this is not trivial, and, to the authors' knowledge, there are no off-the-shelf solutions for this purpose. In fact, while there is an abundance of simulation software that facilitate creation of digital twins, and there are many Internet-of-Things (IoT) solutions for data transfer between devices, such solutions that address both aspects at once appear to be few and far between. This is reflected in the case-by-case nature of practical applications mentioned above.

Recently, Aho and Immonen (2020) introduced an open-source software framework, called *ModelConductor* that defines a digital twin design pattern to facilitate on-line asynchronous data interexchange between a physical device and a digital twin simulation model. *ModelConductor*, basically a connector of the physical devices to their digital twins as described in Subsection II-B, is capable of handling multiple asynchronous data streams via a variable-length queue containing objects measured from the physical environment, waiting to be processed by the simulation model. It also supports running different simulation model types through the Functional Mock-up Interface (FMI) (Blochwitz et al., 2011), which standardizes model exchange and co-simulation between different computation environments and is now supported by more than 150 simulation platforms.

While *ModelConductor* provides a proof-of-concept solution addressing the above basic concerns for co-execution of physical systems and digital twin simulation models, it is not fully compatible with many-to-many relationship between data sources and running simulation instances because of direct function calls between the queue structure and the simulation model.

Communications of the ECMS, Volume 35, Issue 1,
Proceedings, ©ECMS Khalid Al-Begain, Mauro Iacono,
Lelio Campanile, Andrzej Bargiela (Editors)
ISBN: 978-3-937436-72-2 / 78-3-937436-73-9(CD) ISSN 2522-2414

Additionally, two way communication facility with the purpose of controlling the physical device by its digital twin is not implemented there. Moreover, the structure of the *ModelConductor* framework restricts distribution of computation on several machines and balancing the requests load between multiple instances running the computational models. This could prevent the framework to scale properly with execution of several computational simulation models being fed by big data streams. In this paper, we propose an extension to the *ModelConductor* framework, namely *ModelConductor-eXtend*, or *MCX* for short, that addresses the above concerns. Several use cases are included for validation and illustration.

I-B Contributions and key limitations

In this article, we describe these extensions to the *ModelConductor* framework:

- Increase scalability and performance of the framework under the condition of high load by decoupling the queue which stores the measurements from the actual execution of computational model.
- Follow a more general message passing and data transmission protocol.
- Introduce a solution for executing time-consuming simulation models (with relatively high data income rate).
- Provide new use cases for system validation by real-world applications.

We emphasize that the *MCX* framework is, at present, only tested for transferring data from physical devices to their simulation model replications. Model-based actuation and feedback control are important implementation steps left for future work.

I-C Relation to previous work

The contributions described in Subsection I-B, all are implemented on the *ModelConductor* framework introduced by Aho and Immonen (2020). The present work is its continuation to improve its functionality and performance, and introduce new use cases for validation.

Toward realizing the digital twin, there exist software platforms that are capable of creating virtual representation of a physical object or system to act as their digital replica. Using these simulation modeling tools, such as Matlab/Simulink, ANSYS twin builder, Dymola, STAR CCM+, Unity3D engine, etc., several studies have been carried out to propose implementation of digital twins for a variety of applications. List of different studies can be found in (Lim, Zheng, & Chen, 2019) and (Cimino, Negri, & Fumagalli, 2019). Previously conducted studies, mostly for pre-specific device, process or system, are limited to simulation and modeling, are not connected to physical devices and do not incorporate real time data. On the other hand, open-source and commercial frameworks are developed to provide the necessary interfaces for handling and authorizing real time data streams in order to securely collect and monitor the data from multiple physical devices (such as Eclipse Ditto (Eclipse, 2020) and Microsoft Azure Digital Twins (Microsoft, 2020)), but they do not address co-execution of simulation models.

This work attempts to bridge the gaps between these distinct domains. More specifically, the *MCX* framework, as an open-source software infrastructure enabling seamless data communication and execution of the twin's simulation model, regardless of application and modeling tool is developed to address the above two capabilities both together.

I-D Organization of the article

Next sections of this paper is structured as follow: In section II, we will introduce the *MCX* framework briefly, explain its capabilities and discuss particularly the new development and alterations proposed in this work. Afterward, three implemented digital twin case examples are discussed in section III as validation applications. Finally, we conclude the paper in section IV and discuss the way to continue the work in the future.

II *MCX* FRAMEWORK

II-A Framework overview

MCX is an open-source software program designed as a ready-to-use structure for the basic connections for implementing and running a digital twin packaged in a standardized general form. The framework itself does not include any simulation, measurement or sensory data, but, on the other hand, it includes the placeholder for running the simulation model and asynchronous message handling structure.

The simplest conceptual use case of the framework is illustrated in Figure 1, with one physical device and its corresponding simulation/prediction model (digital twin). The measured data from a physical sensor, typically (but not necessarily) in a constant frequency, is transmitted in a standard format to the queue by the client. Then, the queue stores the measurements until they are fetched by the subscribed simulation/prediction model. Whenever the model is ready to process new data, it will receive the first element in the queue, then the step method of the model will be executed to produce the output. The output (model response) is logged and could be used for monitoring purpose and also sent back to the physical device's actuator. Although the actuator is not implemented in the framework yet, it is used to demonstrate the concept.

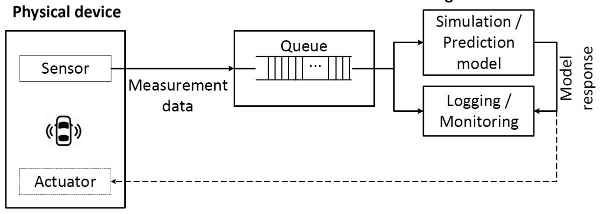

Fig. 1: Basic components of *MCX*

II-B The starting point for MCX *development*

The basis for *MCX* is the *ModelConductor* framework introduced by Aho and Immonen (2020). There, the `Experiment` class was used to hold the model, queue and results all together. The sequence diagram of its main data processing loop is reproduced for reference in Figure 2. It shows that, on each iteration of the loop, the code checks whether there is at least one element in the buffer. If so, and also the model is in a `Ready` state (i.e. not preoccupied processing a previous data element), a measurement data point is removed from the buffer and used to make an inference from the associated model by calling `step` method of the `ModelHandler` object. The result — a `ModelResponse` object — is then appended to another list, an attribute of the `Experiment` object.

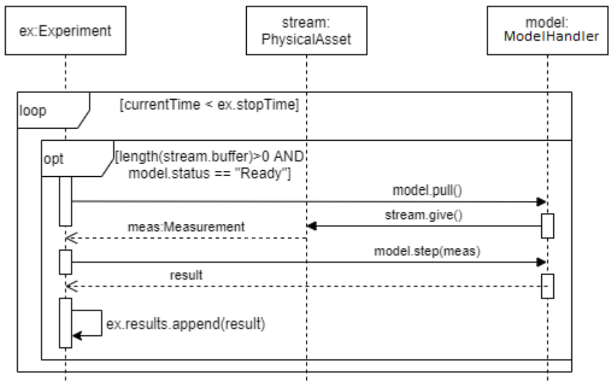

Fig. 2: Sequence diagram in *ModelConductor* (based on Aho and Immonen (2020)).

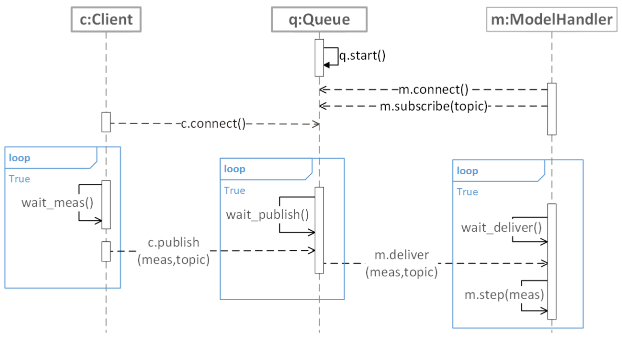

Fig. 3: Sequence diagram in *MCX*

II-C New developments in MCX

The sequence diagram of the proposed *MCX* framework is shown in Figure 3. In the remainder of this section, we will compare the two frameworks to each other and point to the shortcomings of the initial version one by one. We will also discuss the proposed modifications and new functionalities in detail.

1) Decoupling the queue from the model

The queue data structure in Figure 1 is a FIFO (First In, First Out) data buffer storing measurements which have arrived from the physical device and are waiting to get processed by simulation model. Utilizing this structure is a necessity to accommodate asynchronous data streams. As explained in the (Aho & Immonen, 2020), the queue operations (push & pop) for handling measurement data observations are both executed in one process but in separated threads. As illustrated in Figure 2, the `Experiment` object explicitly waits until a new message is appended to the queue buffer. It means that an instance of the `ModelHandler` class (that is exactly the place where one step of the simulation model is executed) must wait until the arrival of a new measurement data once it has processed current data. The condition for the availability of new data is checking the queue being non-empty (length of `stream` be greater than zero). Hence, it can be observed that the class which holds the simulation model is tightly coupled with the queue structure. In such coupling situation, a change in one module may enforce changes in other modules, affecting code reusability and scalability. Tightly coupled systems are often seen as disadvantage (Beck & Diehl, 2011).

In *MCX*, see Figure 3, the queue structure is decoupled from the `ModelHandler` and each of them are executed in different processes. Then the connectivity between the two modules (Queue and Model) is established via an MQTT (Message Queuing Telemetry Transport) connection (see Section II-C2). This allows for distribution of computational load, as the queue can be processed on a remote computer. This not only increases the scalability and loosens the coupling of the system, but also facilitates many-to-many relationships between multiple data producers and multiple data consumers.

2) Replacing raw TCP with MQTT

In *ModelConductor*, measurement data was transmitted using a TCP socket with a pre-defined message format (stringified JSON with fixed header size describing the length of the message). In the *MCX*, we use MQTT instead of TCP. MQTT is a Client Server publish/subscribe messaging transport protocol which has been widely used in data-intensive IoT applications. The motivation for using MQTT include:

- MQTT offers a standard messaging protocol which is supported by IoT community. It also provides integration to the open-source and commercial cloud services (such as Google Cloud, Microsoft Azure and Amazon web services).
- MQTT enables two-way communication between physical device and its digital twin in an effective and scalable way. The structure of MQTT also facilitates many-to-many relationships in data streams.
- MQTT messaging transport is agnostic to the content of the payload. This make the messaging protocol indifferent to the application of the digital twin.
- There are three different qualities of service (QoS) for message delivery status in MQTT protocol. These qualities (including “at most once”, “at least once” and “exactly once”) are practical in digital twin.
- MQTT is considered as an application layer protocol in the well-known Open Systems Inter-

connection model (OSI model)[1] utilizing TCP as the infrastructure for message transportation.

A schematic of the general connectivity facilitated by the proposed MQTT-based communication is shown in Figure 4. The built-in queue in the MQTT broker enables asynchronous data connection and the structure follows the publisher-subscriber paradigm. Each publisher can be seen as an individual sensor, sending the measured data into the system with specific topic which one or more subscribers are listening to its messages. In the context of this paper, subscribers could be seen as computational simulation models. With this generic structure, the framework can fit to the variety of applications while maintaining scalablilty and performance. Additionally, reverse data stream i.e. from model to the physical device, is feasible which facilitates model-based control (not experimented in this work though).

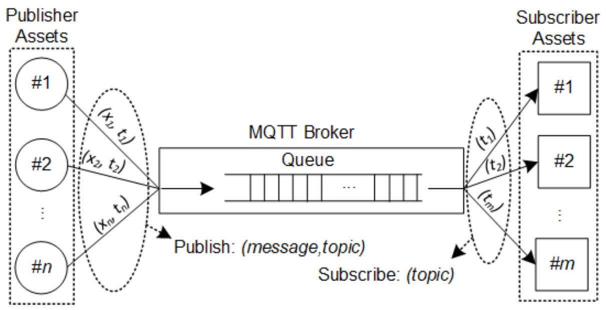

Fig. 4: General overview of the *MCX* framework with multiple publisher assets streaming the sensory data each with specific topic and multiple computational models

3) *Memory*

ModelConductor was first designed to process one measurement data at a time and produce its response by executing one step of simulation/prediction model. This can be referred to as sample-synchronized memoryless model execution: Each iteration of the model execution is only dependent on the current input data, and each input measurement data has precisely one model response. However, in some practical applications, the model can be computationally expensive, with execution time exceeding the arrival time of new measurements. Thus, following a sample-synchronized memoryless procedure may cause an accumulative delay in responses and also lengthening of the queue over time.

MCX addresses this issue by proposing a time-synchronized memoryful historical modeling procedure by adding a local buffer which stores incoming data while the model is being executed on previous data points. Historical modeling can be further categorized into two types based on the length variability of their local buffer memory: fixed-length and variable-length buffer (see the example in Subsection III-B).

II-D MCX *interfaces to simulation models*

To use the *MCX* framework, the simulation model of the specific application is embedded into the framework acting as the running digital twin. There are three different ways in *MCX* for this purpose:

1) Functional Mock-up Unit (FMU) models: If the simulation model is developed in one of the 150+ tools that support FMU export[2] (such as ANSYS, CATIA, GT-SUITE, MapleSim, MATLAB® Simulink®, SimulationX and etc.), then the exported model could be easily embedded in the framework. The facilities for importing the model, setting input/output variables and running one step of the model are implemented in the `FMUModelHandler` class (using FMPy library[3]).
2) Scikit-learn model: If the digital twin is based on the predictive machine learning model developed in Scikit-learn library (Pedregosa et al., 2011), it could be integrated into *MCX* using `SklearnModelHandler` class.
3) Custom class: If neither of the above options apply, then the user can implement customized behavior in a Python class and embed it as a running simulation model. The class is inherited from `ModelHandler` with specific methods to load, step and shutdown the model.

III CASE STUDIES

III-A Drone simulation and control

In this validation example, an open-source MATLAB/Simulink based dynamic modeling and simulation of a drone (quadcopter) (see Hartman, Landis, Mehrer, Moreno, and Kim (2014)) has been exported into FMU container and then the FMU package has been embedded into the *MCX* to act as the flying drone's digital replica. The simulation is an attitude-command-only model which means the controller only tries to track attitude commands (orientation in terms of angles: ϕ for roll, θ for pitch and ψ for yaw) and altitude command (z) using a PID controller. The idea of drone's digital twin, here, is that a copy of the control command (ϕ, θ, ψ, z), initiated from the drone remote controller, is sent to the drone's digital simulation running on *MCX*. The idea is illustrated in Figure 5, however, dashed arrows were not considered in this example and a client has been placed in the $\times$ sign to mimic the behaviour of the remote controller. With this set up, we can follow the internal dynamic variables of the drone using its simulation while the actual physical device is running.

In this 50 second simulation, the drone is initially stationary at $z = 3.048$m. Then at $t = 25s$, a simple roll command is performed (with ϕ changing as a step signal while keeping the other command variables θ, ψ and z constant). The input command and observed positional values y and z of the drone are illustrated in Figure 6 (x position remains close to zero and is omitted). As shown, the controller tries to keep the altitude (z) constant and due to the roll command drone starts to move forward along the y direction.

[1]https://osi-model.com/

[2]https://fmi-standard.org/tools/

[3]https://github.com/CATIA-Systems/FMPy

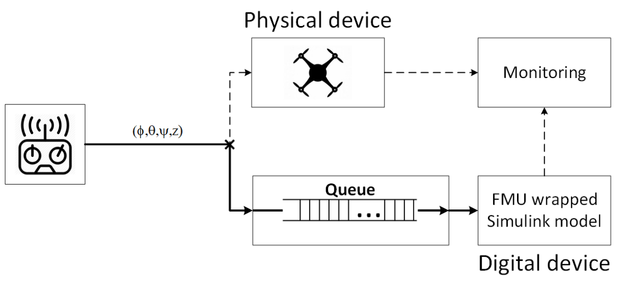

Fig. 5: Overview of the components in the drone experiment.

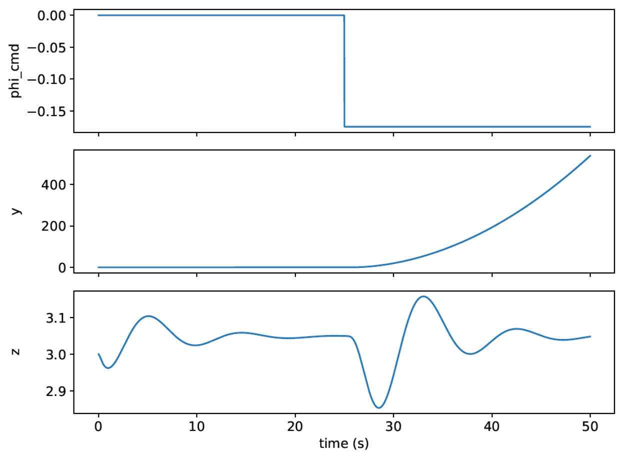

Fig. 6: Input command and positional values of the drone simulation over 50 seconds of the experiment

III-B Historical time series prediction

As a proof-of-concept example of the memoryful historical procedure (explained in II-C3), we built an artificially slow (computationally expensive) model by explicitly "sleeping" during the model response computation. In this demonstrative experiment, we tried to predict the next value of a time series by fitting a linear regression model to the trailing window of the series (last N data points).

We send a numerical value as a synthesized measurement data (with noisy 3rd degree polynomial pattern) every 10 milliseconds while the model computation takes R milliseconds long where R is sampled from a uniform distribution in $[0, 1000)$ interval for every measurement. This setup leads us to the variable-length window time-synchronous historical modeling. Hence, the window (buffer) holds $N \in [0, 100)$ data points each time and those are used to fit a regression model and predict the next value (Figure 7).

III-C NO_x emission prediction

In this example, Scikit-learn machine learning models were used as digital twin simulation models for predicting NO_x exhaust emission from a real 4-stroke offroad diesel engine during run time. The experiment is the same as described Aho and Immonen (2020) (Section III) except that 1) input feature vectors are normalized to have zero mean and unit variance, and 2) NO_x emission output value is normalized as relative to initial NO_x output. Three regression models were fitted to the data: Random forest with 100 trees and maximum depth 25, Linear and 2nd order polynomial regression models. The obtained results are shown in Table I as Mean Absolute Error and coefficient of determination (also known as r-squared or R^2) for both training and test data. The normalized data is now publicly available in the project repository on Github.

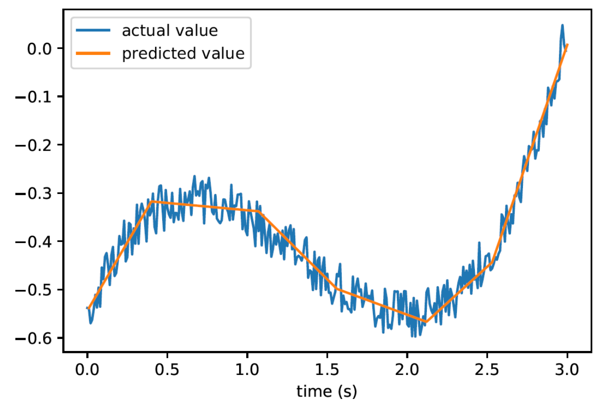

Fig. 7: Demonstrative example of memoryful historical modeling: fit a linear regression model (orange curve) at times to predict next value of time series (blue curve).

TABLE I: Training and test results for NO_x emission prediction on different regression models

Model Type	train R^2	test R^2	train MAE	test MAE
Linear regression	0.75	0.73	0.64	0.65
Polynomial regression	0.96	0.96	0.24	0.24
Random forest	0.99	0.99	0.03	0.08

III-D Performance evaluation

A key motivation for the proposed *MCX* development is performance optimization. Table II describes the results of a performance evaluation comparison between *ModelConductor* and *MCX* in different computational experiments. Here, performance is measured in terms of response time, defined as time difference from timestamp client sends the data until the timestamp model response is ready, averaged across a number of samples. The experiments were carried out with the client and model on the same computer to mitigate the effect of random network packet transmission delays. The results show that *MCX* is almost 4 times faster than *ModelConductor*.

TABLE II: Average response time for different experiments: A comparison between *ModelConductor* and *MCX* (in milliseconds)

Experiment	# of samples	ModelConductor	MCX
FMU couple-clutches	4000	52.7	9.9
FMU drone simulation	2500	67.31	18.98
Historical time-series prediction	3000	-	11.54
Sklearn NO_x emission	1422	92.5	22.6

First experiment in the Table II, is a simple open-source simulation for drive train with 3 dynamically coupled clutches implemented in MapleSim. The other three experiments are explained in Subsections III-A, III-B and III-C respectively. Since *ModelConductor* does not support historical modeling, the average response time regarding historical time-series prediction example in the Table II could not be calculated.

IV CONCLUSIONS

In this work, *MCX* framework was presented as an open-source software platform enabling digital twin implementation by providing asynchronous scalable data transmission facilities as well as online co-execution of different simulation models. Three extensions to the previous version of the framework (namely *ModelConductor*) were described which are: 1) decoupling the queue from the model computation module, 2) usage of MQTT instead of raw TCP, and 3) implementing a solution for time-synchronization in computationally expensive models (memoryful historical models). With these extensions applied, *MCX* performed faster and more scalable compared to *ModelConductor*. In addition, the experimental setup and results of three validation examples of the framework were described.

In spite of the functionality implemented in the *MCX* framework, it has some limitations. First, an explicit waiting routine for the data to be ready is used in measurement handling procedure which consumes processing resources inefficiently. This can be refined with event driven implementation and callback functions to improve the performance. Another limitation is that the framework does not include a standard implementation for a two-way communication, although it is supported by the architecture.

Future research work on the topic should focus on implementing model-based control over *MCX* with the two way communication infrastructure between physical and digital devices. Another interesting direction for future research is identification and adaptation of the simulation model during run time. Here, one begins with a rough model and attempts to refine it as new measurement data becomes available (this feature may not be supported by the current FMU specification). Finally, more real-world validation applications for *MCX* should be considered in the future, also including digital twins for manufacturing processes besides the physical devices considered thus far.

SOURCE CODE AND EXAMPLES

The source code of the framework is available on: https://github.com/COMEA-TUAS/mcx-public

ACKNOWLEDGEMENTS

The authors gratefully acknowledge funding from Business Finland (e3Power project).

REFERENCES

Aho, P., & Immonen, E. (2020, Sep). Modelconductor: An on-line data management architecture for digital twins. In *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)* (Vol. 1, p. 1397-1400).

Beck, F., & Diehl, S. (2011). On the congruence of modularity and code coupling. In *Proceedings of the 19th acm sigsoft symposium and the 13th european conference on foundations of software engineering* (pp. 354–364).

Blochwitz, T., Otter, M., Arnold, M., Bausch, C., Clauß, C., Elmqvist, H., ... others (2011). The functional mockup interface for tool independent exchange of simulation models. In *Proceedings of the 8th international modelica conference* (pp. 105–114).

Cimino, C., Negri, E., & Fumagalli, L. (2019). Review of digital twin applications in manufacturing. *Computers in Industry*, *113*, 103130.

Colombo, A. W., Karnouskos, S., Kaynak, O., Shi, Y., & Yin, S. (2017). Industrial cyberphysical systems: A backbone of the fourth industrial revolution. *IEEE Industrial Electronics Magazine*, *11*(1), 6–16.

de Azevedo, H. D. M., Araújo, A. M., & Bouchonneau, N. (2016). A review of wind turbine bearing condition monitoring: State of the art and challenges. *Renewable and Sustainable Energy Reviews*, *56*, 368–379.

Eclipse, F. (2020). *Ditto: where iot devices and their digital twins get together.* Retrieved 2020-11-01, from `https://www.eclipse.org/ditto/`

Hartman, D., Landis, K., Mehrer, M., Moreno, S., & Kim, J. (2014). *Quadcopter dynamic modeling and simulation (quad-sim) v1.00.* Retrieved 2020-11-01, from `https://github.com/dch33/Quad-Sim`

He, R., Chen, G., Dong, C., Sun, S., & Shen, X. (2019). Data-driven digital twin technology for optimized control in process systems. *ISA transactions*, *95*, 221–234.

Lee, W., Suh, E. S., Kwak, W. Y., & Han, H. (2020). Comparative analysis of 5g mobile communication network architectures. *Applied Sciences*, *10*(7), 2478.

Lim, K. Y. H., Zheng, P., & Chen, C.-H. (2019). A state-of-the-art survey of digital twin: techniques, engineering product lifecycle management and business innovation perspectives. *Journal of Intelligent Manufacturing*, 1–25.

Microsoft. (2020). *Azure digital twins: Next-generation iot solutions that model the real world.* Retrieved 2020-11-01, from `https://azure.microsoft.com/en-us/services/digital-twins/`

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., ... Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

AUTHOR BIOGRAPHIES

SAJAD SHAHSAVARI works as Researcher at Turku University of Applied Sciences, and is a PhD student at University of Turku, Finland.

EERO IMMONEN is an Adjunct Professor at Department of Mathematics at University of Turku, Finland, and works as Principal Lecturer at Turku University of Applied Sciences, Finland.

MOHAMMAD-HASHEM HAGHBAYAN is a post-doctoral researcher at University of Turku, Department of Future Technologies, Finland.

MOHAMMED RABAH (PhD) is working as a Research Engineer at Computational Engineering and Analysis Research Group, Turku University of Applied Sciences, Finland.

JUHA PLOSILA is a Professor in the field of Autonomous Systems and Robotics at the University of Turku, Department of Future Technologies, Finland.