A Graph-Based Meta-Data Model for DevOps in Simulation-Driven Development and Generation of DCP Configurations

Stefan H. Reiterer[1] Clemens Schiffer[1]

[1]Department E, Virtual Vehicle Research, {stefan.reiterer,clemens.schiffer}@v2c2.at

Abstract

To improve the quality of model-based development and to reduce testing effort DevOps practices gain more and more importance. However, most system engineers are not DevOps specialists and there are a lot of manual steps involved when writing build pipelines and configurations of simulations. For this purpose, an abstract graph-based meta-data model is proposed. This allows auto-generation of scenario descriptions for the DCP standard and code for the build server where the simulation is set up and executed. A simple use case is described as an example of how this could be applied in practice. Furthermore, a Python implementation of a DCP master and a simple FMI to DCP wrapper are presented in this work.

Keywords: Continuous Integration, DevOps, MBSE, NoSQL Graph Data Bases, DCP, SysML, UML, SSP

1 Introduction

To tackle the growing complexity of software on electronic control units (ECUs) in cars or co-simulation of physical phenomena of different parts of the vehicle the use of practices from software development is on the rise. Especially DevOps plays an important role. According to Bass, Weber, and Zhu (2015) DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into production while ensuring high quality.

However, while DevOps is well established in software development there remain a few challenges when simulations come into play. Simulations are often very complex and need a lot of expert knowledge from other fields like mechanical or electrical engineering. Thus, there is a need to provide frameworks which need only little knowledge of DevOps tools like build servers to enable the application of proper development practices.

Furthermore, when performing model-based engineering there may be several components available from previous or parallel projects. Thus, it would be convenient to integrate them in the current workflow and test the system in several variations (models, versions, or different parameters). Hence, it would be more efficient to allow automatic setup of existing artifacts and pipelines from an abstract description provided in UML, SysML or the SSP standard and then generate the necessary simulation setup from that description, at least in a semi-automatic fashion. For this, the Distributed Co-Simulation Protocol (DCP) standard was chosen since it allows abstract description of co-simulation configuration for very different kinds of setups. In Section 2 a simple use case is provided as an illustrating example.

To achieve this goal an abstract graph data structure is introduced in Section 3 to build the link between system engineering tasks, DevOps and co-simulation. Furthermore, it is shown that the data structure is suited for data transformations between general scenario descriptions and co-simulation scenarios and we will discuss the implementation of the use case in more detail in Section 4.

2 A Simple Use Case

In this section a typical process in development of simulation models and the roles involved are described. A use case for this process is derived. The challenges and potential benefits of applying dedicated DevOps methods are highlighted.

2.1 Use Case Description

The following scenario is considered. A system engineer wants to test two related software components. Their common behavior defines a system, which is subject to usage in product development projects at a later point in time. They know that there are two prototypes from development available but they want to rely on the nightly version to use the most up to date version. They want to experiment and incorporate small changes, hence the software components have to be build from scratch in a regular fashion. To use them, the system engineer has to

- get access to the code,
- build a pipeline (or script) and
- set up a co-simulation scenario and run it.

See Figure 1 for a schematic of the use case. The content of the red box highlights what the system engineer has to define. The code, shown in green boxes, is maintained by developers. The boxes in blue are related to process automation. Typically, a DevOps engineer is responsible for implementing these activities. A clear separation of tasks enables every member of the team to focus on their respective role in the process.

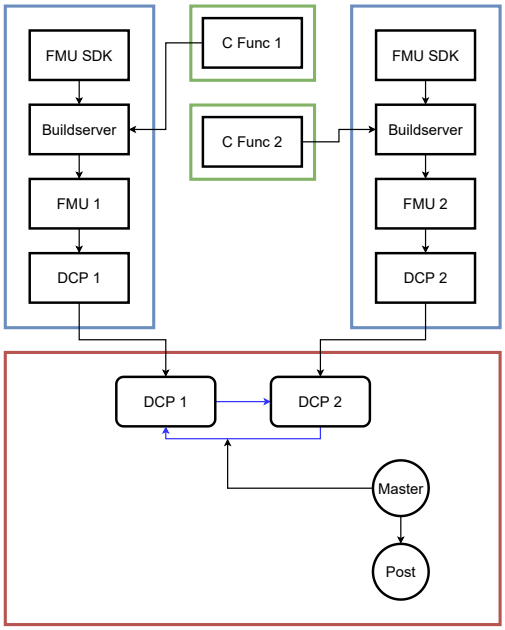

Figure 1. Schematics of a Simple Build and Simulation Pipeline

- The system engineer defines the system designs including model boundaries, their scope and in particular the flow of model signals between models. This can either be done from scratch or by working in part with previously established models.
- The model developer creates the models and their implementation according to the previously defined system design.
- The DevOps engineer provides build pipelines – or templates for build pipelines – to build the models and deploy the resulting instances of these models as artifacts.

Similarly pipelines for running the simulation need to be provided. These pipelines should be as flexible as possible to be parametrized for different configurations and situations. In Figure 2 the description of the simulation participants is depicted. Here as an example UML was used for the proof of concept. However, this is not limited to UML since other standards like SysML or SSP could be used for the transformation as well since they are easily parsable.

2.2 Challenges

Although the tasks described in the previous subsection are manageable in general, they suffer from several well-known problems. First of all, the entire process is considered complex, and involves many different tasks as sources for faults and errors. Fixing problems is time consuming and unnoticed errors can lead to disasters. Especially setting up the infrastructure for builds is not always

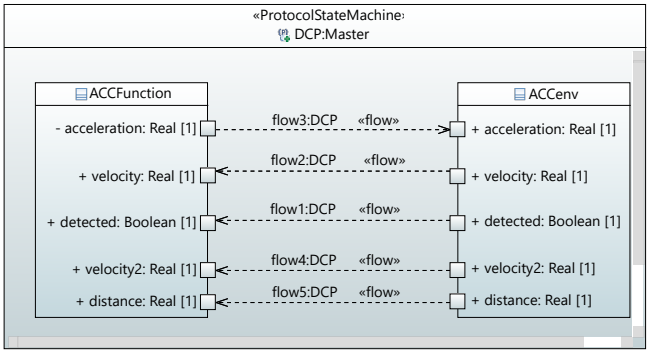

Figure 2. System described in UML

straight forward and needs proper configuration management. This includes setup of scripts, pipelines, specific software versions, etc. This may be an additional challenge for someone with little software engineering background. Even with the help of the developers and DevOps engineers it may be cumbersome. Staff might not be available all the time. Reaction time might be limited, so several of the arising issues may not be fixed immediately. For these reasons it would be desirable to have dedicated mechanism in place. This paper contributes by (1) introducing a method for setup of simulation-driven development processes that rely on graphs, (2) provision of an implementation consuming these graphs, automating the build process, and generate prototypical systems for simulation and testing.

3 Co-Simulation Process Graphs and the DCP Standard

In this section the idea of using a process graph for simulation execution is elaborated. The used co-simulation standards are presented and the specific challenges of using graph-based DevOps methods for configuration management are identified. An theoretical overview of the implementation is given.

3.1 Motivation

There are several data formats and tools available to tackle the tasks for the systems engineer described in the previous section. However, the main problem for modern engineers are the interfaces and steps which are necessary to go from one domain to the other. Co-simulation protocols like DCP on the one hand solve this problem from the view of simulation by providing uniform interfaces and descriptions of the simulation participants. Graph-based workflow descriptions on the other hand allow setting up continuous integration pipelines, as discussed e.g., for Gitlab CI in (Pundsack 2018). Nevertheless, when dealing with simulations there are problems. Existing graph-based solutions for workflow descriptions either rely on structural properties like having no directed cycles (directed acyclic graphs or DAGs for short) or only work in an online set-

DOI 10.3384/ecp21181411

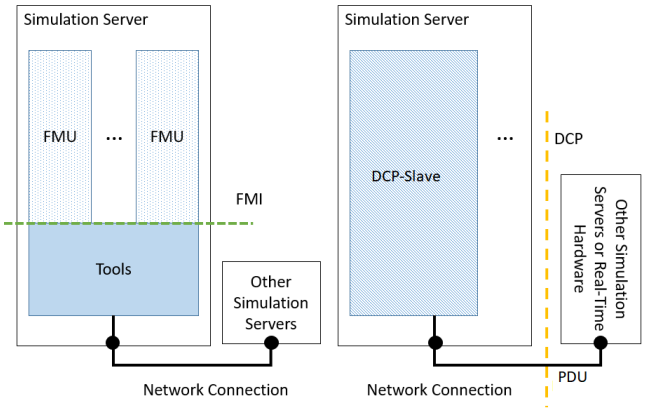

Figure 3. Comparison FMI-based simulation and DCP-based Simulation

ting. This stems from the fact that orders of workflows can only be described by a graph if there are no cycles, but when dealing with closed loop simulations such cycles are introduced naturally. It is beneficial for optimization, analysis and resource management tasks and automatic code generation of build scripts to provide a data structure which works offline and still describes all dependencies and all communication ways.

3.2 Co-Simulation Standards

The FMI standard has greatly simplified the exchange of simulation models across tool boundaries (Blochwitz et al. 2012). However, a corresponding standard for real-time systems that includes network communication is missing. The Distributed Co-Simulation Protocol (DCP) is an application-level communication protocol, designed for cost-effective development processes and opportunities to easily integrate models into simulation environments (Krammer and Blochwitz 2018). It standardizes the exchange of simulation-related configuration information and data (Modelica Association Project DCP 2019). The DCP standard fills this gap of FMI by standardizing the behavior of the model and the exchanged messages on the level of the communication protocol to simplify the integration of different real-time systems. It reduces the configuration effort required drastically thereby increasing the efficiency of tests and simulations. See Figure 3 for a comparison of the two standards, for FMI see (FMI-Working-Group 2020, p.97). Furthermore, the DCP standard defines a way to describe scenarios by providing a specified XST tranformation for scenarios which can be directly used for the automatic configuration of the simulation setup (Krammer and Benedikt 2018). Hence it is perfectly suited for the goals of the use case from the foregoing section, and we can use it as leverage for the automation of the whole workflow.

3.3 The Co-Simulation Process Graph

To use the leverage of the tools available without introducing a new software chain a graph-based meta-data model was introduced which allows the description of workflows and simulation scenarios in a unified manner. The so-called co-simulation process graph was formally defined in (S. H. Reiterer et al. 2020) and is an extension to the classical process graph (Tick 2007). It solves the problem of cycles introduced by closed loop simulations and models without the need of separating the workflow sequence and the topology of the simulations. By definition a co-simulation process graph is a directed graph with the following properties:

- The set of nodes consists of data nodes, transformation nodes, master nodes, signal nodes and communication (or gateway) nodes.
- To represent the instantiation of a process or the usage of a signal inside a simulation, copies of the nodes which represent these instances are made. Instances have to be directly connected to their originals.
- Instead of using the bi-partite structure to represent data transformations, only instances of processes can connect to data nodes to perform operations. In this way, the nodes which perform operations and their instantiation can be determined with a suitable algorithm, which determines a different partition of the graph with help of the defined structure, to provide the correct order of executions. This is necessary since it is allowed that transformation nodes are neighboring, e.g., a Docker container which is built and then used for executing a program afterwards.
- An information node can never be the successor or predecessor of another information node. A process must be placed in between. However, neighboring process nodes are allowed. This may happen if a program-performing transformation at a later stage is modified beforehand by another process (e.g., parameterization of tools).
- A simulation is a sub-graph with the following properties: a) It contains the instance of a master node. b) The instance of the master node is connected to all instances of signal nodes that belong to the simulation. c) All the other nodes inside the simulation (i.e. the simulation participants and communication gateways) neighbor a signal instance. d) Each instance of a signal is only allowed to appear once inside a simulation.
- Cycles are only allowed inside a simulation subgraph.

A more detailed description of the data structure and analysis of the used algorithms can be found in the paper (S.

Reiterer and Kalab 2021), which was recently accepted in the International Journal of Simulation and Process Modelling. An example is shown in Figure 4. A possible example is that the nodes c_1 and c_2 represent software sources (e.g., source code of a model) b represents a build tool like CMAKE and b_1 and b_2 represent two processes of this build tool which are started, which leads to the simulation units P_1 and P_2, while the node M represents a simulation master. After the build in stage 1) the simulation is executed and the master is configured by the information contained in the node M and gets additional parameters from node I, while the node O represents the output of the simulation. The nodes i_j and o_j represent in- and outgoing signals like velocity or acceleration, while g_j represents the communication protocols (e.g., a network protocol like IP) for $j = 1, 2$.

It can be shown that a co-simulation process graph can efficiently be transformed into a DAG. Those transformations are based on contractions which are of particular importance since they allow a simplification of the graph to make the representation more accessible for human users, because the data structure was designed to be computer friendly and can become quite complex.

The transformation nodes can also be filled with existing scripts and code snippets to improve reusability of existing build scripts. This way the graph structure can be either directly used as a simple low code platform for programming pipelines or linked with existing technologies to make the combination of the continuous integration world with the realm of simulations easier. Since it is a universal data structure it is not necessary to introduce new tooling but allows linking the existing tools into the framework.

See Figure 5 for the database view on the build graph for the adaptive cruise control (ACC) function as a further example. This graph contains all necessary steps for the build steps of the ACC function participant. Additionally, the database holds the signals which are associated with the participant which should be used in the scenario description for the DCP simulation. Several manufacturers have fixed catalogs of signals which are allowed to use.

Storing them in the database together with simulation participant helps developers to avoid using wrong signals.

4 Implementation

In this section the details of the implementation are presented. This includes the description of the used simulation participants and how the co-simulation standards are used. Further the configuration management using meta-data embedded in the process graph and the deployment of the resulting configuration to the simulation are discussed.

4.1 General Implementation

The prototype of the graph transformation service was implemented in Python with usage of the networkX library (Hagberg, Schult, and Swart 2008). The transformation service transforms the graph description of the build

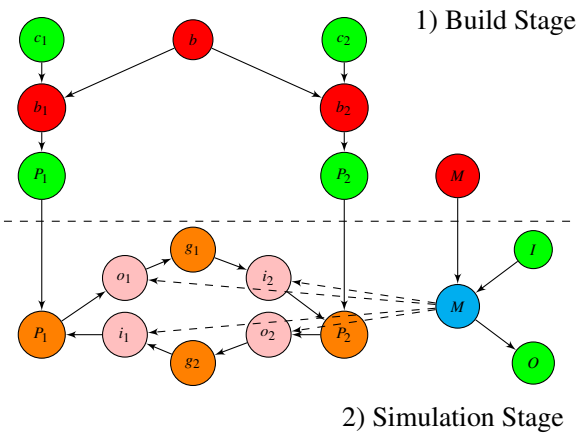

Figure 4. Simple Example of a Co-Simulation Process Graph

pipeline from XML and stores it in a graph database. Here ArangoDB was used due to the generous license model for the community edition (BSD like license) and since it is recommended as one of the stronger candidates (Fernandes and Bernardino 2018). Furthermore, the scenario description was drawn with the Eclipse plugin Papyrus since it is one of the few free SysML/UML modeling tools available which allow export of the UML diagrams as text file (Lanusse et al. 2009). As a build server Jenkins was used for which the build pipelines were generated. To ease the deployment Docker containers were built in which the simulations could run without worrying if the necessary libraries are available on the Jenkins slave which performs the build process.

4.2 Implementation of the Use Case in Detail

From the abstract description depicted in Figure 2 a simple XML parser in Python was used to transform the model exported from Papyrus to generate the necessary nodes and edges for the simulation subgraph (in the sense of Subsection 3.3) and the provided meta-information which was entered into the description is parsed and entered into the nodes.

The service then sends AQL (Arango Query Language) queries to the graph DB and gets the related build pipelines for the simulation participants and the master and combines them. After this the DCP description file is generated from the graph description directly on the build servers' workspace where the simulation is executed. Hereby the graph service enters all connections between the participants and the configuration of the master. Furthermore, the signals with the used value references are taken from the database. After that all subgraphs for the pipelines are connected and the necessary code for the build server is generated and executed. See Listing 1 for the generated code. In lines 4-13 the setup of the participants is executed. The necessary commands and parameters for the setup are stored within the graph database. In lines 14-18 a simplified configuration file is written which represents the signal connections via reference values. However, this configuration is for demonstration only. Every line in the configuration file config.ini rep-

DOI 10.3384/ecp21181411

resent the connection of one output to one input referred to by the corresponding participant name and value reference. For demonstration purposes we simply write these lines one by one by diverting the output the echo commands to the file. Note that these commands were automatically generated from the structure of the simulation graph. In a production environment such configuration data has to be supplied in a structured manner, such as an XML-file, a prototype which uses an advanced scenario description format exists as well. Finally in line 20 the simulation is started; this is achieved by running the dcpmaster-service in its own docker. This service in turn uses the generated configuration file to configure and start the simulation participants, which were started up before and were waiting in an idle state for commands.

This pipeline can be subdivided in a build stage and a simulation stage as depicted in Figure 4. During the build stage the simulation participants are built. Two models are to be simulated in the system: An adaptive cruise control (ACC) function which controls the acceleration of a vehicle based on the speed of the vehicle and the distance to and speed of a vehicle ahead, if such a vehicle is detected. This ACC functionality is tested by coupling it with a second model that simulates the entire environment of the ACC, including the ego-vehicle and its surroundings, refer to Figure 2 for the signal flow. Both models are implemented as C code with accompanying meta-data. A build pipeline template uses this code and meta-data from a repository to build the models with their dependencies resulting in finished artifacts, in this use case two FMUs. These artifacts are then tested – both with respect to formally complying to the FMI standard as well as their basic functionality – and subsequently uploaded to an artifact repository.

In the next step of the build stage the simulation environment is set up. Docker containers are build in which the simulation participants will run. A DCP wrapper for FMUs was implemented based on the DCPLibrary – the open-source reference implementation of the DCP. This is used to the make functionality of each FMU available in a DCP slave in a distributed FMI master configuration. This is possible because the DCP standard was designed with the goal of basic comparability with FMI in mind. Such a wrapper can be used to integrate existing high-quality FMU in a networked DCP-simulation or real time system. During the simulation stage the simulation is configured and executed. The docker containers – one for each simulation participant – are started. The configuration of the scenario is generated from the meta-data of the simulation graph, the lower part of Figure 4, this includes which input is connected to which output, parameters and the timing configuration. This configuration is provided to the DCP master, which deploys this configuration to the slaves and starts the simulation. In the use case the DCP master is implemented in Python, demonstrating the interoperability of DCP implementations. A feature was implemented that reads in a DCP scenario description, containing slave descriptions of all DCP slaves involved in the scenario and the desired configuration details, and can generate from this the necessary configuration sequence to roll out the configuration and run the simulation. The master controls the simulation with DCP protocol data units (PDUs) that are sent via network connections to the DCP slaves. The DCP slaves in turn exchange simulation data via data PDUs until the master stops the simulation.

Listing 1. Generated Code for Simulation

```
1  stage('Stage: Sim77706 77706,Sim888') {
2  steps{
3        sh 'echo "Run Simuation Sim888"'
4        script {
5                DockerFolder = "
                     dcp_docker_run"
6                Key = "part1"
7                sh "cd ${DockerFolder}/;
                     docker-compose up -d $
                     {Key}"
8               }
9        script {
10               DockerFolder = "
                     dcp_docker_run"
11               Key = "part2"
12               sh "cd ${DockerFolder}/;
                     docker-compose up -d $
                     {Key}"
13              }
14       sh 'echo "ACCenv, 6, ACCFunction,
             4" >> config.ini'
15       sh 'echo "ACCenv, 3, ACCFunction,
             2" >> config.ini'
16       sh 'echo "ACCenv, 4, ACCFunction,
             3" >> config.ini'
17       sh 'echo "ACCenv, 1, ACCFunction,
             1" >> config.ini'
18       sh 'echo "ACCFunction, 5, ACCenv,
             7" >> config.ini'
19       sh "cp config.ini dcp_docker_run/
             dcp_py_master/ACC/"
20       sh "cd dcp_docker_run/; docker-
             compose up dcpmaster"
21         } }
```

The simulation results are then post-processed with a Python script, zipped and put on an artifact repository, e.g., JFrog Artifactory, where they can be downloaded. In Figure 6 an overview of the generation and linking procedure is provided.

5 Conclusion and Outlook

In this work we presented a simple proof of concept to showcase how CI pipelines could be described in the context of simulations. For this an abstract graph model was described which can be stored within a database and can be used for splitting work and autogenerate simulation scenarios out of model descriptions. A simple simulation was described in UML which was used directly for generating the simulation configuration within the pipeline. Furthermore, we demonstrated how the described pipelines can be stored within a graph database

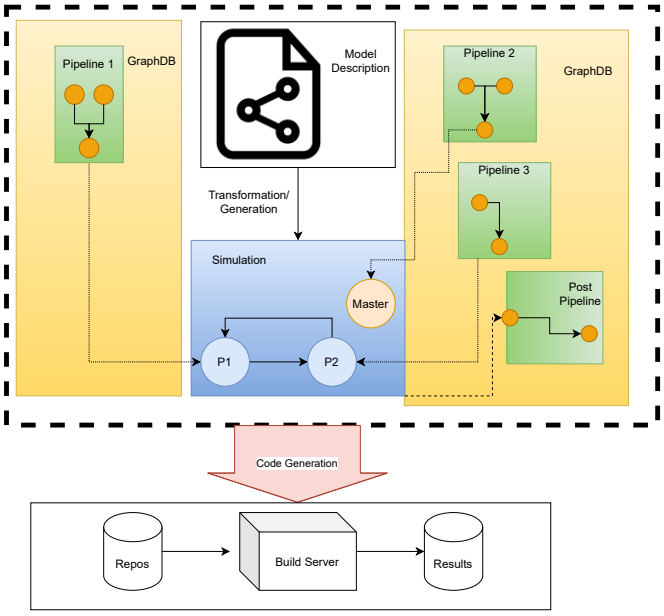

Figure 5. Transformation of Model and Linking of Subgraphs

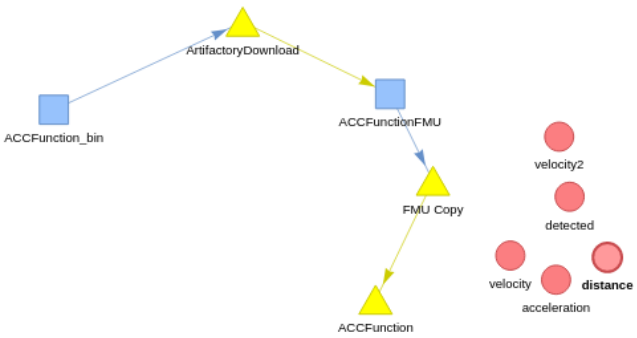

Figure 6. Database View of the ACC Function Build-Graph

and how it can be used to merge existing pipelines to a complete workflow containing all necessary steps. For the simulation part we demonstrated how FMU participants can be converted to DCP participants and how to roll them out automatically via Docker containers within the generated pipeline. Connections between the participants are automatically set with help of the graph data structure and participants are automatically started with help of the provided meta-data. However, it is important to highlight that this was a mere proof-of-concept. There are a lot of aspects regarding proper simulation description which still are open. Currently only a primitive mapping was used but there are several aspects regarding system design and work processes which were not considered yet. Extensions for SysML and SSP are planned to provide a proper tool for systems engineering. Since the proposed graph data structure is very abstract and very flexible there are also quite a few topics which have to be investigated on the implementation side. One topic is the algorithmic analysis of the process graph; partial results were presented on the Grazer Symposium of the Virtual Vehicle (S. H. Reiterer 2020). As already mentioned in Section 3.3 a companion journal paper was recently submitted and accepted (S. Reiterer and Kalab 2021). In this work the algorithmic and modeling aspects of the co-simulation process graph model are described in detail. Furthermore, strategies to optimize the resulting pipelines are investigated. Another big topic is the proper integration of the graph database. This includes versioning of graph-based pipelines within the graph database and managing of configuration parameters and tool variation. Regarding the DCP standard there are several questions remaining like defining a comprehensive standard for the description of the co-simulation process graph elements (node and edge data) to ensure generation of DCP scenario descriptions in a consistent manner. This would not only open the possibility to describe DCP simulations as graphs but would also enable computer aided analysis of the performance of simulations on a large scale or automatically deploy different variations of scenarios.

Acknowledgements

This publication was written at Virtual Vehicle Research GmbH in Graz, Austria. The authors would like to acknowledge the financial support within the COMET K2 Competence Centers for Excellent Technologies from the Austrian Federal Ministry for Climate Action (BMK), the Austrian Federal Ministry for Digital and Economic Affairs (BMDW), the Province of Styria (Dept. 12) and the Styrian Business Promotion Agency (SFG). The Austrian Research Promotion Agency (FFG) has been authorized for the program management. They would furthermore like to express their thanks to Desheng Fu for further input and discussions and Prof. Eugen Brenner and Georg Macher from the Institute of Industrial Informatics of the TU Graz for their support. Also, many thanks to Martin

Krammer from Virtual Vehicle for input and corrections for this work.

References

Bass, Len, Ingo Weber, and Liming Zhu (2015). *DevOps: A software architect's perspective*. Addison-Wesley Professional.

Blochwitz, Torsten et al. (2012). "Functional mockup interface 2.0: The standard for tool independent exchange of simulation models". In: *9th International Modelica Conference*, pp. 173–184. DOI: 10.3384/ecp12076173.

Fernandes, Diogo and Jorge Bernardino (2018). "Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB." In: *DATA*, pp. 373–380.

FMI-Working-Group (2020). *Functional Mock-up Interface for Model Exchange and Co-Simulation*. https://github.com/modelica/fmi-standard/releases/download/v2.0.2/FMI-Specification-2.0.2.pdf. Accessed: 2021-04-12, V 2.0.2.

Hagberg, Aric A., Daniel A. Schult, and Pieter J. Swart (2008). "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, pp. 11–15.

Krammer, Martin and Martin Benedikt (2018). "Configuration of slaves based on the distributed co-simulation protocol". In: *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. IEEE, pp. 195–202.

Krammer, Martin and Torsten Blochwitz (2018). "The Distributed Co-Simulation Protocol for the Integration of Real-Time Systems and Simulation Environments". In: *Proceedings of the 50th Computer Simulation Conference*. DOI: 10.22360/SummerSim.2018.SCSC.001.

Lanusse, Agnes et al. (2009). "Papyrus UML: an open source toolset for MDA". In: *Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*. Citeseer, pp. 1–4.

Modelica Association Project DCP (2019). *DCP Specification Document, Version 1.0*. Linköping, Sweden: Modelica Association. URL: http://www.dcp-standard.org.

Pundsack, Mark (2018). *Out-of-sequence job execution using directed acyclic graphs (DAG) MVC*. https://gitlab.com/gitlab-org/gitlab-foss/issues/47063. Accessed: 2019-11-11.

Reiterer, Stefan and Michael Kalab (2021). "Modelling Deployment Pipelines for Co-Simulations with Graph-Based Metadata". In: *International Journal of Simulation and Process Modelling*. Accepted.

Reiterer, Stefan H. (2020). "Continuous Deployment of ADAS Functions over the Air". In: *13. Grazer Symposium des Virtuellen Fahrzeugs*.

Reiterer, Stefan H. et al. (2020). "Continuous Integration for Vehicle Simulations". In: *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. Vol. 1. IEEE, pp. 1023–1026.

Tick, József (2007). "P-graph-based workflow modelling". In: *Acta Polytechnica Hungarica* 4.1, pp. 75–88.