$CPSA_{ML}$: A Language and Code Generation Framework for Digital Twin based Monitoring of Mobile Cyber-Physical Systems

Andreas Fend
TU Wien, Business Informatics Group
Vienna, Austria
andreas.fend@tuwien.ac.at

Dominik Bork
TU Wien, Business Informatics Group
Vienna, Austria
dominik.bork@tuwien.ac.at

ABSTRACT

Cyber-physical systems (CPS) are finding increasing use, whether in factories, autonomous vehicles, or smart buildings. Monitoring the execution of CPSs is crucial since CPSs directly influence their physical environment. Like the actual system, the monitoring application must be designed, developed, and tested. Mobile CPSs, in contrast to stationary CPSs, bring the additional requirement that instances can dynamically join, leave, or fail during execution time. This dynamic behavior must also be considered in the monitoring application. This paper presents $CPSA_{ML}$, a language and code generation framework for the model-driven development of mobile CPS systems, including a cockpit application for monitoring and interacting with such a system. The pipeline starts with the formulation of the system and the CPSs it contains at an abstract level by the system architect using a domain-specific modeling language. Next, this model is transformed into SysML 2 for further extension and richer specificity by system engineers on a more technical level. In the last step of the pipeline, the SysML 2 model is used to generate code for the CPS devices, a system-wide digital twin, and the cockpit application mentioned above. This cockpit enables the operator to configure and apply the monitoring and interaction with the system during runtime. We evaluate our $CPSA_{ML}$ language and code generation framework on an Indoor Transport System case study with Roomba vacuum cleaner robots.

CCS CONCEPTS

• **Software and its engineering → System description languages**; **Application specific development environments**.

KEYWORDS

model-driven engineering, cyber-physical systems, multi-paradigm modeling, digital twin

ACM Reference Format:
Andreas Fend and Dominik Bork. 2022. $CPSA_{ML}$: A Language and Code Generation Framework for Digital Twin based Monitoring of Mobile Cyber-Physical Systems. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22 Companion), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3550356.3563134

MODELS '22 Companion, October 23–28, 2022, Montreal, QC, Canada

ACM ISBN 978-1-4503-9467-3/22/10.
https://doi.org/10.1145/3550356.3563134

1 INTRODUCTION

The term Cyber-Physical Systems (CPS) typically refers to engineered, physical, and biological systems monitored and/or controlled by an embedded computational core [3]. "The behaviour of a CPS over time is generally characterised by the evolution of physical quantities, and discrete software and hardware states" [3]. This behavior can be modeled mathematically, but this does not guarantee that the system will always behave according to these models at runtime. For this reason, runtime monitoring of such CPSs is essential for maintenance. A monitoring system is used by an operator to get insights into the system during execution. Thereby, system values, states, and events are visualized in a structured way. These insights enable the operator, if necessary, to influence the execution of the system in order to correct or prevent possible errors.

"Compared with CPS that rely on stationary and huge machines or sensors and emphasize how to utilize cyber components to better master the physical world, mobile CPS concentrate on their mobility" [12]. This also places new demands on the CPS to enable this mobility. Since any cable-based connection is a limitation for a mobile CPS, other technologies have to be used. The power supply usually comes from built-in batteries and communication is usually done through wireless technologies. The use of batteries and wireless communication introduces the problem that the communication to the mobile CPS can be interrupted or even disconnected. This poses the challenge of dynamically adding/removing components to/from the system at runtime.

Although an operator uses the monitoring system, it gets realized by a developer. Developers and operators often have different views on the system, which can lead to complications. To bridge that gap, it is important to involve the operator in the design of the monitoring system. This is where the model engineering aspect comes into play. Model engineering abstracts real world concepts in a way, such that only the relevant properties are left. A model therefore is a simplified representation of reality. This helps planning and designing complex systems, as a model should give a common sense of the underlying system.

In this paper we aim to provide a generic solution for the Model-Driven Development (MDD) of mobile CPS and their monitoring systems. Our approach focuses on two specific stakeholders, *system architects* aiming to design systems, and *operators* aiming to monitor and interact with the system. In this work we want to address both stakeholders at different stages following a multi-paradigm approach which is commonly adopted for MDD in CPSs [2, 9]. We will start with a Domain-Specific Language (DSL), which allows the system architect to model the CPS and its functionalities, sensors, and actuators from a more abstract viewpoint. Subsequently, we apply multiple model transformations and code generation to

automate the realization of a cockpit application which enables monitoring and interacting with the CPS system.

Through the proposed MDD pipeline, we assume that we can better support both the system architect and the operator for the development of a mobile CPS. The system architect only keeps the focus on her interests through the introduced DSL, while the operator is supported by the generated cockpit application. In addition, the used code generation should save a lot of implementation effort for developers. Above all, such a solution offers the advantages of rapid adaptation in the code in the case of changes in monitoring or in the system. For evaluating the feasibility of our approach, we will realize a Indoor Transport System (ITS) (see Section 5).

In the remainder of this paper, Section 2 introduces the relevant foundations before related works are presented in Section 3. Section 4 then comprehensively introduces the MDD pipeline which is evaluated in a case study in Section 5. A critical discussion (Section 6) and concluding remarks (Section 7) wrap up this paper.

2 BACKGROUND

2.1 Model-Driven Development

In Model-Driven Development (MDD), models are treated as "first class citizens" that drive the (software) development. Models conform to a metamodel that describe "the whole class of models that can be represented by that language." [7] Such metamodels specify abstract structures (i.e., the abstract syntax) and constraints that are used to validate models [6]. Since the abstract syntax is not intended for humans, a concrete syntax (textual, graphical, or hybrid) must be defined to enable efficient human comprehension [5, 17]. Regarding the scope of a modeling language, languages with a narrow scope, directed toward specific stakeholders with specific purposes (i.e., Domain-Specific Languages (DSL) can be differentiated from languages with a broad scope, addressing several stakeholders with diverging purposes (i.e., General Purpose Languages (GPL)). In addition to language engineering transformations are essential in MDD. Model-to-model transformations are used to convert models of one language into other models conforming to either the same (i.e., endogenous) or a different modeling language (i.e., exogenous). In model-to-model transformations, the elements of the source metamodel are mapped to the elements of the target metamodel.

2.2 SysML 2

The Systems Modeling Language (SysML) 2 [18, 20] is a language standardized by the OMG for the specification, analysis, design and verification and validation of complex systems that may include hardware, software, information, processes, personnel and facilities. SysML 2 builds on the Kernel Modeling Language (KerML) which is divided into the layers *root*, *core*, and the KerML library. SysML 2 is structured in a modular manner by using *packages* which are specified in addition to the concept of elements in the root layer of KerML. The core layer describes the direct semantic mapping to formal logic. In the KerML library fundamental elements, like data types, functions as well as physical units are defined. In the SysML library, which is based on the KerML library, all elements for the different features of the language are defined, which are divided into *structure modeling* and *behavior modeling*.

2.3 Robot Operating System (ROS)

The Robot Operating System (ROS) is an open source framework for the development of robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms [19]. A system executed in a ROS environment consists of several processes, called *nodes*. Such nodes can be implemented in different languages such as C++, Python, Octave, or Java [11]. The nodes communicate with each other via publish/-subscribe mechanisms on top of the TCP or UDP protocol. Besides unidirectional message exchanges via topics, there are also services. A service works like a remote procedure call. It consumes a request and returns a response. The structure of both messages are as well specified in service definitions. Furthermore, ROS has an extensive naming feature, that allows to group ROS nodes into namespaces and thus also to control the communication of the ROS nodes by manipulating the topic and service names without having to adapt the actual code of the nodes.

3 RELATED WORKS

There are several works from the area of model-driven development of CPS with an emphasis on CPS monitoring.

Berrouyne et. al [4] propose a model-driven approach which uses several DSLs as well as transformations to tackle the issues of heterogeneity and interoperability of the individual devices in the IoT. The ThingML [13] language is used to specify the individual devices in a system whereas State machines are used to describe the behavior of a thing. Additionally, the network is modeled using the CyprIoT language.

Vierhauser et al. [23] deal with the critical concerns of CPS in general and unmanned aerial vehicles (UAVs) in particular. They use the ReMinds framework [25], which was originally developed for the monitoring of systems of systems in the domain of automation software for metallurgical plants, and extend it to meet requirements for the monitoring of mobile CPS, such as the dynamic instantiation of multiple instances, the evaluation of constraints, the simulation of CPSs, and the generation of a monitoring GUI representing the live state of the CPS [16].

Vierhauser et. al [24] use MDE to automatically generate runtime monitoring systems based on the *ModIRMo* model-integrated framework. ModIRMo enables the modeling of the monitored system as a domain model with a UML class diagram. ModIRMo generates Monitoring APIs which are responsible for querying the system values and publishing them to an MQTT message broker. In addition, a minimal Digital Twin is generated, which receives the real-time values and validates them using the VIATRA framework [22].

Iglesias-Urkia et. al [15] present TRILATERAL, a model-based approach to accelerate and simplify the development of Industrial CPS (ICPS). TRILATERAL is based on EMF and applies the IEC 61850 [21] standard for modeling. Users need to first create an Information Model and a Server Model. The former reflects the ICPS elements to be monitored and the Control Block, which defines several interaction aspects while the Server Model specifies the communication. Subsequently, a code generater generates C++ code for the middleware of the ICPS containing interfaces for querying and for controlling ICPS elements.

Table 1: A comparison of the works through defined characteristics.

Approach	Addressed Stakeholder	Problem Domain	Used Modeling Languages/Frameworks	Codegenerator / Target Language	Monitoring Modeling Stage	Constraint Validation	Degree of Action	GUI
CyprIoT [4]	System Engineer, Network Engineer	Interoperability Problem of IoT devices	ThingML, CyprIoT	TH-CGEN / C	-	No	Passive	No
ReMinds [25]	Operator	Monitoring Mobile CPS	Requirements Monitoring Model DSL	Java	Runtime	Yes	Passive	Yes
ModIRMo [24]	System Engineer	Monitoring CPS	Monitoring DSL, UML Class Diagram, VIATRA	EMF / Java	Design-time	Yes	Passive	Planned
TRILATERAL [15]	System Engineer, Electrical Engineer	Industrial CPS Development and IoT Protocols	IEC 61850	EMF, Xtend / Java	Design-time	No	Active	Planned
Interactive Digital Twin Cockpit [10]	System Architect, System Engineer	Digital Twins for Cyber-Physical Production Systems	MontiArc, OCL	MontiGem / Java	Design-time	Yes	Active	Yes
TwinOps [14]	Developer, Operator	DevOps for Model-Based Development of Digital Twins	SysML, AADL	C, Ada	Design-time	Yes	Passive	No
Our Work	System Architect, Operator	Monitoring Mobile CPS	CPSA$_{ML}$, SysML 2	Xtend / Java, Typescript, Python, Dockerfile	Runtime	Yes	Active	Yes

Dalibor et. al [10] present a Model-Driven Architecture for the development of Digital Twins for monitoring and controlling Cyber-Physical Production Systems (CPPS) based on MontiArc [8]. Through MontiGem [1], a web-based CPPS Digital Twin is generated.

Hugues et. al [14] introduce TwinOps, a process that combines DevOps, Digital Twins, and Model-Based Engineering to improve the development of CPS. Several different approaches are proposed, where one of them is applied and evaluated by developing a building monitoring system with multiple sensors. SysML is used to model requirements, use case, and block diagrams. Subsequently, model transformations and code generators are applied to generate a minimal middleware. Monitoring probes are implemented manually to further process the real-time data of the sensors.

Comparison. Table 1 summarizes the comparison of related works using a set of characteristics like the addressed stakeholders, the used modeling languages, and whether an approach is passively monitoring or also actively controlling a CPS.

Synopsis. The related works, underpinned by related surveys [2, 9] show, that often a combination of different modeling languages is used to comprehensively represent a CPS including its monitoring and communication. We believe SysML 2 offers the possibility to model many of these aspects of a CPS. It is possible to formulate the domain model, its behavior and constraints in a structured way. The fact that all concepts can be represented in the same model provides continuous support for the modeler. However, since a system is first planned on an abstract level, this language is not suitable for formulating a coarse draft. We therefore want to provide a language to formulate CPS on an abstract level and then further transform this model into SysML 2. Through code generation we want to automate the middleware for communication between components and a simple digital twin acting as a monitoring application. The generated components should follow a system architecture that fulfills the special requirements of mobile CPS.

The presented works use text-based modeling languages to formulate rules at runtime. These languages are quite complex and do not provide intuitive support for the modeler with respect to the domain model. As mentioned in [23], it is important that rules for CPSs can be adapted at runtime, since their behavior is often only revealed at runtime. Since the monitoring application is used by the operator, we want to design it accordingly to her/his needs and requirements. In doing so, the formulation of rules should be simple and intuitively supported by graphical elements at runtime.

4 APPROACH

We present an MDD pipeline for modeling and developing systems containing several different mobile CPS elements. As a part of this pipeline, code generators are applied to produce code for a distributed system. The architecture of this distributed system aims at overcoming well-known challenges of mobile CPS elements, such as the loose coupling of individual CPSs or the dynamic mounting and unmounting of CPSs at runtime. One of the components in this architecture is the Cockpit application. It is used for monitoring, as well as for interacting with the system and the CPS entities acting in it. In the following, we will discuss in more detail the architecture and the communication schema used, the pipeline including its steps, the transformation between them, the languages used, and the addressed stakeholders.

One of the most important requirements for the CPS system is that it is possible to dynamically add and remove CPS entities from the system at runtime. For this reason, the system architecture must be designed in such a way that this can be done without any difficulties. This also brings the challenge of ensuring that there are no problems if a certain CPS entity is not reachable any longer. In Figure 1, we can see the system architecture of our approach. This consists basically of three layers. For communicating between the layers, message brokers and topic-based messaging are applied to decouple the message exchange from the various components.

The bottom layer contains the (mobile) CPS entities. There can be an arbitrary number of these entities at any point in time. CPS entities publish data messages at periodic intervals. Such messages contain sensor data as well as calculated values or states of the CPS

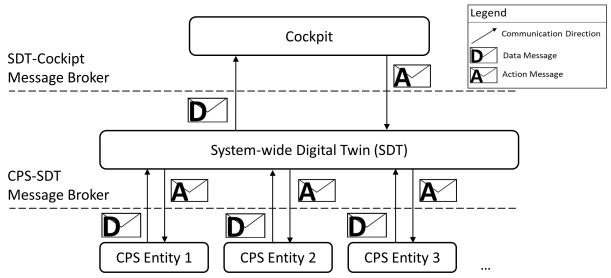

Figure 1: CPS system architecture

entities. Data messages are published to a CPS entity independent, but CPS type specific topic. This allows other components to consume messages from CPS entities of this type without knowing the exact identity of these CPS entities. In addition, CPS entities consume action messages. Action messages describe actions to be performed by the CPS entities. Unlike data messages, action messages are always addressed to a specific CPS entity. If the message is not received, for example, because the CPS entity is not available at that moment, the message expires.

The System-wide Digital Twin (SDT) is located in the middle layer of the architecture. There is only a single instance of it. The SDT subscribes to all CPS topics known in order to be able to receive data messages from every active CPS entity. Thus it holds a global system snapshot, including all components data. One of the SDT's tasks is to detect new CPS entities as well as CPS entities that are no longer accessible. The aggregated system snapshot is as well published in a periodic interval as data message. Additionally, the SDT consumes action messages. The actions described in these messages can either be addressed to specific CPS entities or they can describe CPS entity-independent actions. In both cases, the SDT is responsible for publishing the corresponding action messages to the appropriate CPS entities.

The cockpit consisting of a backend and a frontend is located on the top layer. The backend subscribes to the SDT topic in order to consume data messages describing system snapshots. The cockpit can be configured at runtime by dashboard concepts. Through these configurations, the received data is appropriately prepared and processed before it is then graphically displayed in the frontend. With these dashboard concepts it is possible to display live data in different representations, apply metrics, define rules the snapshots are validated against, describe events that check for certain changes, and actions to trigger predefined actions in the system. When such an action is started in the frontend, the backend publishes the corresponding action message to the SDT.

Note that data messages always flow from bottom up, while action messages always flow from top down. Furthermore, communication between components can only take place between adjacent layers. Different CPS entities, for example, cannot communicate directly with each other. This approach prevents possible conflicts when receiving and processing multiple action messages from different senders.

The goal of our work is to design a method for developing a distributed system with mobile CPS according to the architecture described above, addressing the requirements and needs of identified stakeholders. To this end, we use MDD and present a pipeline

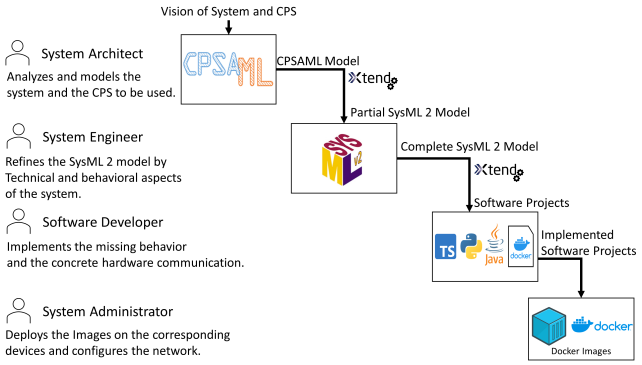

Figure 2: MDD Pipeline

that fulfills this challenge. The pipeline, shown in Figure 2, consists of several steps, each of which involves a certain stakeholder.

The pipeline starts with the system architect that has a certain vision of the system to be developed. This includes the different CPS types to be used, their functionalities, and more complex functionalities in the system that cannot be executed by a single CPS entity. This vision is usually rather abstract and does not yet contain any technical implementation details. For exactly this viewpoint of the system architect we have designed the CPS Architecture Modeling Language ($CPSA_{ML}$) using *Xtext*. $CPSA_{ML}$ allows the system architect to formulate that abstract vision of the system in a textual model. This model is then transformed into a SysML 2 model. SysML 2 offers system engineers the possibility to extend the system to be developed from a far more technical point of view. SysML 2 can be used to define parameters of actions, to integrate types and units, to formulate constraints, or to describe the behavior by means of state machines.

In order to reuse as much information as possible from the SysML 2 model, we use code generators in the pipeline in a next step. These produce code in different languages for the components presented in the architecture. Including source code files in Python, Java, or Typescript, but also other files like Dockerfiles or ROS configuration or message exchange files. This code is maintained in different projects, and is extended by the software developers. The concrete communication with the hardware or the logic of actions to be executed by the system is thereby implemented.

In a final step, the implemented projects are containerized using Docker. This allows the system administrator to deploy and configure the images as easily as possible on the corresponding hosts or devices.

A major contribution in our work is the CPS Architecture modeling language. It fulfills the following six requirements, we derived from the interests of the system architect:

R1 Definition of the types of CPSs occurring in the system.
R2 Multiplicity specification of the CPS types.
R3 Description of the CPS elements functionalities.
R4 Specification of the CPSs hardware components in form of sensors and actuators that interact with the physical environment.
R5 Description of the hardware elements functionalities.
R6 Specification of system-wide functionalities that are not directly related to a specific CPS entity.

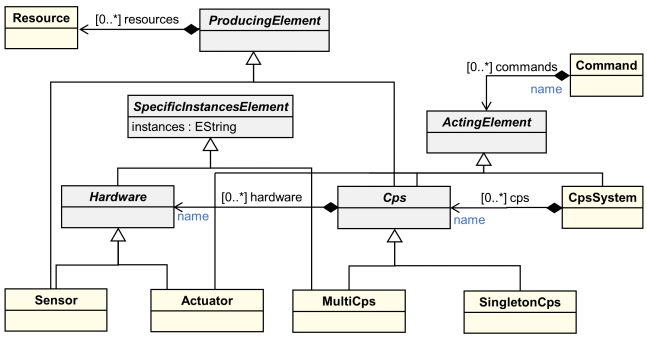

Figure 3: CPSA$_{ML}$ metamodel.

Since the language is only intended to fulfill the above-mentioned requirements, it is kept rather simple, as can be seen from the few concrete classes in the metamodel in Figure 3 (visualized using yellow background color). Each concrete class from the metamodel extends the abstract *DescribedNamedElement* class, which is not included in the figure for reasons of better readability. This class contains the attribute *name* and the optional attribute *description*, that can be used to describe an element in more detail.

The root element of the CPSA$_{ML}$ metamodel is the *CpsSystem*. It contains all CPS types that occur in the system. In addition, CPS entity independent commands are defined here. A *Command* represents an operation that triggers one or more actions in the system or for specific CPS entities. Such a command is not specified in more detail in CPSA$_{ML}$, since this is usually not in the interest of the system architect.

A *Cps* element describes a certain CPS type as well as its multiplicity in the sysem. A distinction is made between *SingletonCps* and *MutliCps*. *SingletonCps* occur only once in the system. *MultiCps* occur either with an exact number or with an arbitrary number in the system. Describing the exact number makes sense, for example, when it is known that exactly two robot arms are used in the system to perform certain tasks, and it is important to distinguish between them. To do this, the exact names of these entities are specified in the *instances* array of the *MultiCps*. For mobile CPSs, it is often not known exactly how many entities will be in use, or this can change frequently at runtime. For this second case, the *instances* array is left empty, meaning that any number of such CPS entities can exist.

The data published by *Cps* are represented as resources. *Resources* are represented in CPSA$_{ML}$ also only very superficially without any type specification. In addition, the commands the CPS entities can process are described in the CPS. Finally, the hardware components used in a CPS type are also specified. Since it is necessary to determine the number of times a certain hardware component is used in the CPS, the hardware class extends the abstract *SpecificInstancesElement* class. However, the use of the *instances* attribute contained therein behaves differently here than it does with the *Cps* class. Since it makes no sense to use an uncertain number of a particular hardware component, an empty *instances* array means that the component is installed only once in the CPS. Otherwise, each installation is noted with the exact identifier in the *instances* array. For hardware, a distinction is made between sensors and actuators. Sensors specify resources that can be used, while actuators specify commands that can be executed.

By satisfying R2, we tried to design the language in a way that the definition and usage of *Cps* or hardware components are not separated to avoid unnecessarily confusing or complex concepts. This resulted in the approach with *SingletonCps* and *MultiCps*, as well as the concept with the *instances* array for *Cps* and hardware components. This also brings drawbacks, e.g., hardware that is used by several different Cps must be specified multiple times.

In the course of our work, we have developed a concrete textual syntax for CPSA$_{ML}$ that aims to allow system architects to use the language as easily as possible. We decided upon a textual rather than a graphical syntax for reasons of a simpler prototyping but nothing would generally stand against a graphical syntax for CPSA$_{ML}$.

4.1 CPSA$_{ML}$ to SysML transformation

The fact that CPSA$_{ML}$ is a rather simple language leads to its limits being reached very early on. For example, it is not possible to formulate data types of resources more precisely or to specify parameters for commands. However, during the development of the system even more technical questions arise, which are no longer in the interest of the system architect but rather in the interest of the software engineer. It is not only about the data types of the resources or the parameters of an action, but to describe the CPS system on a more technical level. Since CPSA$_{ML}$ does not offer this possibility and it does not make sense from our point of view to adapt the language in this way, we have decided to use a transformation to SysML 2.

We decided to transform to SysML 2 because the language, with its extensive and above all very technical concepts, is ideally suited for making further specifications for the CPS system. These include language features such as the separation of definitions and usages, utilization of units and definition of new complex data types, annotations, compositions, actions, state machines, and constraints. Since SysML 2 is a predominantly textual language, we decided to use Xtend for the transformation, to ensure that the output models directly have our preferred structure.

During the transformation, a minimal SysML 2 library provided by us is included, containing attribute definitions for the use of annotations and constraints. These are very useful for certain use cases and facilitate further modeling with SysML 2. In addition, some annotations are essential for further code generation, as they serve as markers.

Basically, the transformation maps each element from the CPSA$_{ML}$ language to one definition and at least one usage in SysML. A visual representation of the transformation can be seen in Figure 4. The blue blocks are representing CPSA$_{ML}$ elements, while the closed arrows with the diamonds show their containments. Red blocks represent SysML 2 definition elements and the open arrows show the usages between them. The striped arrows demonstrate the transformation of the elements.

Resources are mapped to attribute definitions, *Commands* to action definitions. *Hardware* elements are mapped to part definitions, whereby these are tagged with the `@RosNode` annotation. This indicates the code generator later that this part is to be translated into an own ROS node component. The contained *Resources* of the *Sensors* and the contained *Commands* of the *Actuators* are used in the respective part definition as usages of the attributes and the

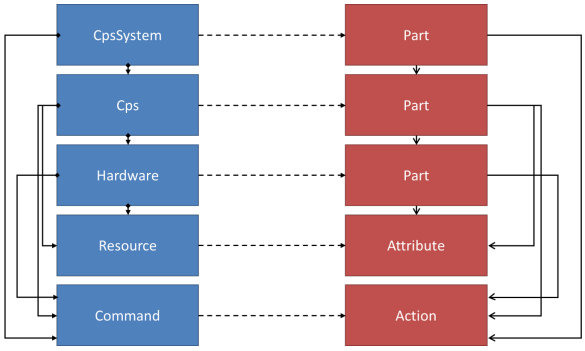

Figure 4: CPSA$_{ML}$ to SysML 2 transformation.

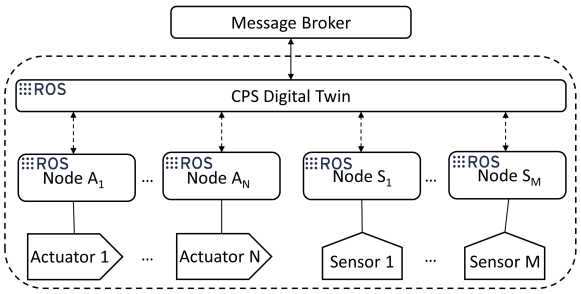

Figure 5: ROS node components of a CPS package

actions. For *Cps* also part definitions are created. These contain the action usages of executable *Commands*, attribute usages for the publishing *Resources*, and part usages for the installed *Hardware* components. In case a *Hardware* component occurs multiple times, indicated by the identifiers in the *instances* array, a part usage is created in the *Cps* part definition for each of these occurrences. In addition, the *Cps* part definition is provided with the `@Cps` annotation. This indicates to the code generator that the complete CPS should be translated into a dedicated ROS package. Finally, the *CpsSystem* root element is mapped to a part definition, provided with the `@System` annotation. In this part definition any CPS part usages are included, depending on their multiplicity. If there is an arbitrary number of a certain CPS, it is declared as an unbounded array usage. In addition, the system-wide *Commands* are used as action usages within that part definition.

It is important that any information from the CPSA$_{ML}$ model is transferred to the SysML 2 model during the transformation, since the CPSA$_{ML}$ model has no further application in our pipeline. The result of the transformation offers a good separation of the different components by the package concept of SysML 2. In order to be able to use the code generator of our pipeline, it is necessary to follow certain guidelines during the modeling. This includes the explicit use of the annotation definitions provided by us according to certain criteria, or also restrictions against action parameters and their typification.

4.2 SysML to ROS transformation

After modeling the system in SysML 2, code generation follows in our pipeline. In this step, ROS packages are generated for the modeled CPSs. Thus, the code for each CPS is in an independent project and can be further implemented, tested, and built separately from the other projects.

For any attributes used in `@RosNode` or `@Cps` annotated Part definitions, ROS message files are created. In addition, ROS service files are created for any actions used in the same part definitions. These files define the data structures of messages, as well as requests and responses for services. When building the ROS package, Python files with the corresponding data classes for messages, requests and responses are generated from these files.

Each `@RosNode` annotated part definition is translated to a ROS Node component. Thereby three Python files are created. A base class, that implements the entire ROS communication, an empty sub class, that extends the base class and must be implemented by the software developers afterwards, and a client class, that can be used by other ROS components in order to communicate with that component easily over method calls. Thus we create a good separation of the actual logic and the communication to other ROS components.

In the base class, a ROS topic is created for each attribute usage. This topic is used to publish the latest value of this attribute at periodic intervals. Additionally, a service is registered for each action usage and an empty handler method is created for that service. This method can be overwritten by the sub class in order to be able to implement the logic for the action.

For each CPS package a local CPS Digital Twin (DT) component is generated, executed as ROS Node. This DT holds the aggregated state of the CPS entity. It is also the only component that communicates to the outside via the message broker. It publishes the latest entity snapshot at periodic intervals. Incoming action messages are processed and the corresponding ROS services are invoked. As with the `@RosNode` annotated part definitions, a base class is generated for the local CPS DT, providing all boilerplate code for external and internal communication, and an empty extending sub class, that must be further implemented after the code generation.

Finally, a launch file is generated for the package, specifying the configuration for the correct execution of the ROS package. To ensure the proper communication between the nodes, it defines which and how many ROS nodes will be started, their names, and namespaces. The architecture of that resulting ROS package can be seen in Figure 5.

4.3 Code Generation

In addition to the ROS packages, the SDT and the cockpit are generated during code generation. The SDT is being realized as a Maven project in Java and is based on the Spring Boot Framework. For each CPS type a message listener is created, listening for incoming data messages from the corresponding CPS entities. Additionally, service classes are created for each CPS type, responsible for publishing action messages. An additional message listener is created for consuming action messages. Such messages are processed and either forwarded to the appropriate CPS entity, or in the case of a system-wide action, the corresponding generated method is called. For each of these system-wide actions, one such stub method is generated in a separate service. The snapshots of the CPS entities are stored in repositories for a limited time. If a snapshot expires before a new snapshot of the same entity is received, the entity is

considered offline. At a periodic interval, the SDT publishes an aggregated snapshot that includes the latest CPS entity snapshots as a data message. Finally, a Dockerfile is generated, for containerizing the SDT as a Docker image. This image can then be easily deployed on a powerful machine, e.g., in the cloud.

In contrast to the SDT, the Cockpit project consists to a significant degree of static code. The main components required by the Cockpit are the data structures of the data messages published by the SDT, the executable actions of the CPS entities and the SDT, as well as the constraints of attributes and action parameters.

4.4 Cockpit Dashboards

The front-end of the cockpit composes a *Config Mode* and an *Operator Mode*. In *Config Mode*, the cockpit can be fully configured at runtime, whereas these configurations are applied in *Operator Mode*. Dashboards are created to configure the cockpit. A dashboard defines a CPS type for that dashboard concepts are configured. Optionally, a dashboard can be assigned to a specific CPS entity. Otherwise, the rendered dashboard includes a dropdown element in *Operator Mode* to switch between all active and offline CPS entities. Dashboard concepts include monitors, events, checks, and actions.

Monitors are used to display live data of the system. The data can be displayed in different ways, like simple text, range or diagram. In addition, operations can be applied to these values, such as aggregation functions on lists. To make the values easier to interpret, metrics can be defined for monitors. Logical expressions are formulated to map the value of the monitor to the colors green, orange, or red. Since we wanted to ease the development of such monitors for operators, we generate also a simple and intuitively usable logic tree-based configuration (see Figure 6) at runtime.

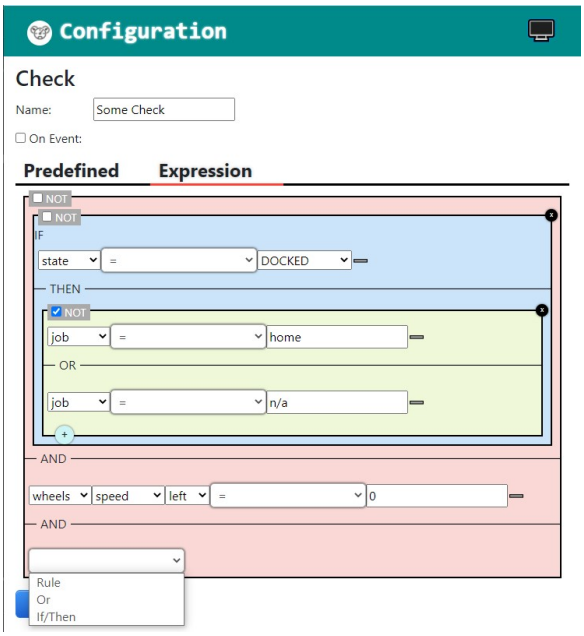

Figure 6: Logic tree-based configuration of monitors at runtime

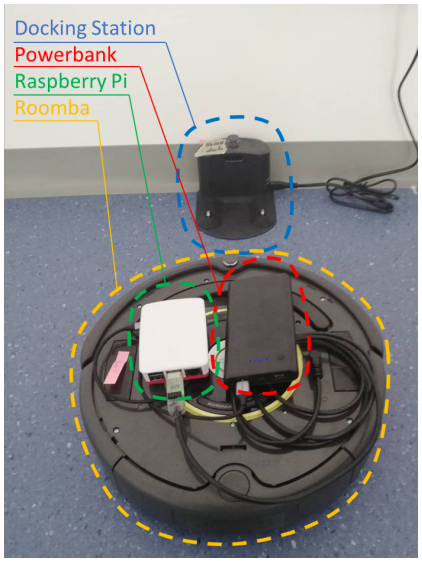

Figure 7: Roomba case study setup

Events are used to compare changes in two successive snapshots. If the condition of the event specification is satisfied, the event is logged together with the snapshot. In *Operator Mode*, an event log is displayed for this purpose. When selecting an event in the event log, the corresponding snapshot is displayed in the dashboard instead of the live data.

Checks formulate rules about assumptions that are made. These are automatically evaluated for each new snapshot. If the rule of a check fails, it will be highlighted in the *Operator Mode* and a notification will be sent to the operator, either via email or SMS.

Actions are tools for executing CPS and system-wide actions. For example, predefined parameter values can be specified. In *Operator Mode*, a block is displayed for each action containing the arguments to be defined as input fields and a button that initiates the execution of the action.

5 CASE STUDY

In our case study we want to realize an Indoor Transport System (ITS). In this ITS, the goal is to transport small items inside a building. We therefore need only one CPS type for this case study that is able to move around in a building and carry additional items. While there are some industrial transport robots that have been developed specifically for this task, we decided to use iRobot Roomba 650 Create1 vacuum cleaner robots, further referred to as Roomba. These are much cheaper, have a large number of different sensors, actuators, as well as a clearly defined API that allows controlling them.

The Roomba is controlled by a Raspberry Pi 4 model B2 that is placed on top of the Roomba and connected via USB. It runs the Raspberry Pi OS3 and has Docker Desktop 3.4.04 installed. Additionally, a powerbank is placed on the Roomba. This serves as power supply for the Raspberry Pi. In Figure 7 we can see the Roomba with the Raspberry Pi and the powerbank.

Using the ITS, it should be possible to manually control a Roomba, have a Roomba perform a list of tasks, and have a specific as well as any idle Roomba navigate to a specific location.

5.1 ITS.cpsaml

In a first step we modeled the ITS using CPSA_{ML}. We defined the system-wide command `Request` and the *MultiCps* Roomba. The Roomba CPS is able to execute the commands `Drive`, `Turn`, and `Stop` for manual controlling, `Stack`, `Start`, and `Clear` for giving the Roomba tasks, execute them and clear them, as well as the commands `Dock` and `Navigate` for driving into the docking station and autonomous navigation.

Listing 1: The ITS modeled using CPSA_{ML}.

```
/*
 * Indoor Transport System
 *
 * CPS system for transporting little objects within
 * a building using mobile roombas.
 */
cps-system ITS:

/*
 * To request an arbitrary idle roomba to
 * a specific place.
 */
command Request


cps Roomba {

    ***
    Manual Driving
    ***
    command Drive
    command Turn
    command Stop // stops all current activites

    ***
    Tasks Driving
    ***
    /*
     * Adds a task to execute on a stack
     */
    command Stack
    command Start // Starts executing the stack
    command Clear // Clears the Stack

    ***
    Autonomous Driving
    ***
    /*
     * Drives the roomba to its docking station
     * for loading the battery.
     */
    command Dock
    command Navigate

    sensor Battery {
        /*
         * The amount of energy
         */
        produces energy
        produces docked // flag if the roomba is in its docking station
    }

    sensor Bumper {
        produces pushed
    }

    actuator Wheel
      instances leftWheel, rightWheel {
        command Drive
        command Stop
    }

}
```

The `Battery` sensor provides information about the power level of the Roomba, as well as whether the battery is charging. The `Bumper` sensor only provides the information if it is pushed. There is a `leftWheel` and a `rightWheel` instance of the `Wheel` actuator. Both can execute the `Drive` and `Stop` commands, where Drive specifies a speed at which the motor should rotate and Stop makes the motor stop. The complete CPSA_{ML} model can be seen in listing 1.

5.2 SysML 2 Modeling

After transforming the CPSA_{ML} model into an initial SysML 2 model, we extended this model by specifying technical details. We first defined custom units, like `RPM` (rounds per minutes), `Second`, `Percentage` and `Degree`, and specified bound constraints for them. Afterwards we added parameters to the already existing actions. We therefore added a `speed` parameter to the `Drive` actions and typed it as `RPM`, an `angle` parameter for the `Turn` action of type `Degree`, a `target` parameter for the action `Navigate` of type `string` and finally the parameters `speed`, `time`, and `angle` of types `RPM`, `Second`, and `Degree` for the `Stack` Action.

Next, we added type information for the published sensor data. The `energy` attribute of the `Battery` is of type `Percentage` while the `docked` attribute is just a simple `Boolean` flag, just like the `pushed` attribute of the `Bumper`. We additionally added a `speed` attribute to the `Wheel` definition, to indicate the current speed of the motor.

Finally, we modeled a state machine for the Roomba, including the states `IDLE`, `MANUAL`, `TASK`, `SEEKING`, `DOCKED`, and `NAVIGATING`, to specifies some high-level behavior.

5.3 Cockpit

After the code generation phase and implementing the missing logic as well as the concrete communication to the hardware components, we built the software, containerized it using the provided Dockerfiles and deployed it on the corresponding machines. As there are functional requirements for ITS, we also formulated some requirements from the operator's point of view.

Thereby we want to see in the cockipt any live data of the Roomba, have a metric displayed for the battery level, and make sure that a Roomba is not driving anymore if it collided with an object. Additionally, some specific events should get recognized and logged, like object collision, the Roomba gets lifted or put down, or the task list or navigation completed successfully.

We tested different scenarios to evaluate if the cockpit with all defined dashboard concepts fulfills the desired functionalities. We have tested scenarios to check if the systems recognizes newly added Roombas as well as Roombas going offline. We also tested scenarios to check, if all the actions are invoked accordingly, both system-wide as well as entity-specific actions. One scenario was about event detection and if they get logged accordingly, and in the last scenario we manipulated the behavior of the Roombas to see weather alerts are risen in case of rules of checks get violated.

In each of those scenarios the ITS worked as expected as well as all the requirements on the Cockpit were fulfilled. Figure 8 shows the Cockpit in *Operator Mode* after applying the **EVENTS** scenario.

On top we can see the two specified dashboards `Roomba` and `ITS`, whereby the first one is selected. As no specific Roomba entity is bound to that dashboard, a dropdown is displayed, listing all Roomba entities. Within the dashboard we can see the live data in the monitors on the left side of the screen, actions to invoke for the Roomba entity in the middle of the screen and the event log as well as the assured checks on the right side of the screen. The `Speed` as well as the `Power` monitors use a range representation, while all the other monitors use simple textual representations. Additionally, the `Power` monitor includes a metric, that is displayed as the horizontal

Figure 8: Roomba dashboard in Operator Mode with the selected R1 entity.

traffic light element. For each action parameter a corresponding input component is used to specify its value before invocation. Numerical parameters have sliders together with a number input field, while textual parameters have a small input text field. Actions without any parameters do not have additional input elements, like the `Stop`, `Dock`, `Start`, and `Clear` action. The checks are listed in the top console. As long as the rule of the check is ensured, the check is displayed green, otherwise it turns red. In the event log we can see all the occurred events for the selected entity. Each log entry contains an icon, indicating the type of event, the timestamp of the event as well as a message.

6 DISCUSSION

We were able to show that the MDD approach presented in our work is applicable to the development of the ITS. Using the CPSA$_{ML}$ language we developed, we were able to express the aspects of the system architect, such as the commands, cps, and hardware parts in a simple textual form. The subsequent transformation to SysML 2 has adopted all the modeled elements of the CPSA$_{ML}$ model, saving a lot of work in SysML 2. SysML 2 turned out to be a very suitable modeling language here. The many different concepts of the language enabled the formulation of structures, behavior, constraints as well as metadata. With the subsequent code generation, over 9000 lines of code, contained in 155 classes, could be generated. The complete middleware for the communication of the different software components was generated thereby. Table 2 shows for each project of the ITS how many lines of code as well as classes were generated and how many lines of code and classes had to be implemented manually afterwards.

By running the above scenarios, we could not only show that the presented architecture is well suited for mobile CPS as well as the special requirement of dynamically adding and removing CPS entities at runtime. We were also able to show that through the different dashboard concepts, any requirements also work in their entirety and deliver the desired result.

Of course, this research is not free from limitations and threats to validity [26], the most severe one is the fact, that we were only able to evaluate our approach in one concrete case, the indoor transportation system. Future research needs to apply the approach in further cases and we believe the modeling languages and code generators are likely to be adpated throughout this process. Still, the case we chose is realistic and uses real sensor data of real mobile CPSs. As such it serves the feasibility study. Given the very positive results with respect to the different evaluation scenarios and the extent to which we were able to automatically generate code, we believe this research establishes meaningful first contributions that future work can build upon.

	Roomba		SDT		Cockpit	
	Generated	Manual	Generated	Manual	Generated	Manual
Classes	17	3	18	0	120	1
LoC	582	359	677	12	7867	36

Table 2: Generated and manually implemented source code.

Another focus of future research is on empirically evaluating the perceived usefulness of the CPSA$_{ML}$ modeling language and the model-driven approach it is integrated in. We want to quantify and qualify the benefits of using the higher-level language CPSA$_{ML}$ instead of directly starting modeling the CPS with SysML.

7 CONCLUSION

Monitoring of mobile CPS is important to ensure their correct behavior and to detect potential errors and apply corrective actions as early as possible. The development of such monitoring applications is not trivial, since the structures and behavior of the CPS must be taken into account, as well as the interests and aspects of the operators using these applications.

In our approach, we use an MDD pipeline to drive the development of such systems using multiple modeling languages, with each language focusing on the requirements and needs of a specific stakeholder. With code generation, we enable the generation of large pieces of code. The thereby completely generated cockpit

combines four important concepts for monitoring and interacting with the system. By applying the pipeline to our use case of the indoor transport system, we were able to show that our approach allows to create a fully operational system and a suitable monitoring application, thereby saving costs by generating code that would otherwise have to be implemented manually.

The source code accompanying this research is available here[1]. We are currently working on integrating sensor data streams from real cyber-physical production systems in the Austrian Center for Digital Production. We want to extend our framework such that is can cope with industrially standardized sensor streams and automatically generate the models and the monitoring dashboard.

ACKNOWLEDGMENTS

This research has been partly funded by the Austrian Research Promotion Agency (FFG) via the Austrian Competence Center for Digital Production (CDP) under the contract number 854187.

REFERENCES

[1] Kai Adam, Judith Michael, Lukas Netz, Bernhard Rumpe, and Simon Varga. 2019. Enterprise Information Systems in Academia and Practice-Lessons learned from a MBSE Project.. In *EMISA Forum: Vol. 39, No. 1*. De Gruyter.

[2] Moussa Amrani, Dominique Blouin, Robert Heinrich, Arend Rensink, Hans Vangheluwe, and Andreas Wortmann. 2021. Multi-paradigm modelling for cyber-physical systems: a descriptive framework. *Softw. Syst. Model.* 20, 3 (2021), 611–639. https://doi.org/10.1007/s10270-021-00876-z

[3] Ezio Bartocci, Jyotirmoy Deshmukh, Alexandre Donzé, Georgios Fainekos, Oded Maler, Dejan Ničković, and Sriram Sankaranarayanan. 2018. Specification-based monitoring of cyber-physical systems: a survey on theory, tools and applications. In *Lectures on Runtime Verification*. Springer, 135–175.

[4] Imad Berrouyne, Mehdi Adda, Jean-Marie Mottu, Jean-Claude Royer, and Massimo Tisi. 2020. A Model-Driven Approach to Unravel the Interoperability Problem of the Internet of Things. In *Advanced Information Networking and Applications*. 1162–1175.

[5] Dominik Bork, Dimitris Karagiannis, and Benedikt Pittl. 2018. Systematic analysis and evaluation of visual conceptual modeling language notations. In *2018 12th International Conference on Research Challenges in Information Science (RCIS)*. IEEE, 1–11.

[6] Dominik Bork, Dimitris Karagiannis, and Benedikt Pittl. 2020. A survey of modeling language specification techniques. *Inf. Syst.* 87 (2020). https://doi.org/10.1016/j.is.2019.101425

[7] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. Model-driven software engineering in practice. *Synthesis lectures on software engineering* 3, 1 (2017), 1–207.

[8] Arvid Butting, Oliver Kautz, Bernhard Rumpe, and Andreas Wortmann. 2017. Architectural programming with montiarcautomaton. *ICSEA 2017* (2017), 224.

[9] Giuseppina Lucia Casalaro, Giulio Cattivera, Federico Ciccozzi, Ivano Malavolta, Andreas Wortmann, and Patrizio Pelliccione. 2022. Model-driven engineering for mobile robotic systems: a systematic mapping study. *Softw. Syst. Model.* 21, 1 (2022), 19–49. https://doi.org/10.1007/s10270-021-00908-8

[10] Manuela Dalibor, Judith Michael, Bernhard Rumpe, Simon Varga, and Andreas Wortmann. 2020. Towards a Model-Driven Architecture for Interactive Digital Twin Cockpits. In *International Conference on Conceptual Modeling*. Springer, 377–387.

[11] Pablo González-Nalda, Ismael Etxeberria-Agiriano, Isidro Calvo, and Mari Carmen Otero. 2017. A modular CPS architecture design based on ROS and Docker. *International Journal on Interactive Design and Manufacturing (IJIDeM)* 11, 4 (2017), 949–955.

[12] Yanxiang Guo, Xiping Hu, Bin Hu, Jun Cheng, Mengchu Zhou, and Ricky YK Kwok. 2017. Mobile cyber physical systems: Current challenges and future networking applications. *IEEE Access* 6 (2017), 12360–12368.

[13] Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif Husa. 2016. ThingML: a language and code generation framework for heterogeneous targets. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*. 125–135.

[14] Jerome Hugues, Anton Hristosov, John J Hudak, and Joe Yankel. 2020. TwinOps-DevOps meets model-based engineering and digital twins for the engineering of CPS. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. 1–5.

[15] Markel Iglesias-Urkia, Aitziber Iglesias, Beatriz López-Davalillo, Santiago Charramendieta, Diego Casado-Mansilla, Goiuria Sagardui, and Aitor Urbieta. 2019. TRILATERAL: A Model-Based Approach for Industrial CPS–Monitoring and Control. In *International Conference on Model-Driven Engineering and Software Development*. Springer, 376–398.

[16] Lisa Maria Kritzinger, Thomas Krismayer, Michael Vierhauser, Rick Rabiser, and Paul Grünbacher. 2017. Visualization support for requirements monitoring in systems of systems. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 889–894.

[17] John Mylopoulos. 1992. Conceptual modelling and Telos. *Conceptual modelling, databases, and CASE: An integrated view of information system development* (1992), 49–68.

[18] Object Management Group. o.J.. SysML V2, OMG SysML. URL: https://www.omgsysml.org/SysML-2.htm. [Accessed: 14.12.2020].

[19] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. Kobe, Japan, 5.

[20] Ed Seidewitz. 2021. Intro to the SysML v2 Language-Textual Notation. https://github.com/Systems-Modeling/SysML-v2-Release/tree/master/doc.

[21] IEC TC. 2003. 57," Communication networks and systems in substations–Part 7-4: Basic communication structure for substation and feeder equipment–Compatible Logical Node Classes and Data Classes. *International Electrotechnical Commission, Geneva, Switzerland, Draft Standard* (2003), 61850–7.

[22] Dániel Varró and András Balogh. 2007. The model transformation language of the VIATRA2 framework. *Science of Computer Programming* 68, 3 (2007), 214–234.

[23] Michael Vierhauser, Jane Cleland-Huang, Sean Bayley, Thomas Krismayer, Rick Rabiser, and Pau Grünbacher. 2018. Monitoring CPS at runtime-A case study in the UAV domain. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 73–80.

[24] Michael Vierhauser, Hussein Marah, Antonio Garmendia, Jane Cleland-Huang, and Manuel Wimmer. 2021. Towards a Model-Integrated Runtime Monitoring Infrastructure for Cyber-Physical Systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results*. 96–100.

[25] Michael Vierhauser, Rick Rabiser, Paul Grünbacher, Klaus Seyerlehner, Stefan Wallner, and Helmut Zeisel. 2016. ReMinds: A flexible runtime monitoring framework for systems of systems. *Journal of Systems and Software* 112 (2016), 123–136.

[26] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[1] Source code repository: https://github.com/me-big-tuwien-ac-at/cpsaml