Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems

Jörg Christian Kirchhof, Judith Michael, Bernhard Rumpe, Simon Varga, Andreas Wortmann
Software Engineering, RWTH Aachen University, Germany, www.se-rwth.de

ABSTRACT

Digital twins emerge in many disciplines to support engineering, monitoring, controlling, and optimizing cyber-physical systems, such as airplanes, cars, factories, medical devices, or ships. There is an increasing demand to create digital twins as representation of cyber-physical systems and their related models, data traces, aggregated data, and services. Despite a plethora of digital twin applications, there are very few systematic methods to facilitate the modeling of digital twins for a given cyber-physical system. Existing methods focus only on the construction of specific digital twin models and do not consider the integration of these models with the observed cyber-physical system. To mitigate this, we present a fully model-driven method to describe the software of the cyber-physical system, its digital twin information system, and their integration. The integration method relies on MontiArc models of the cyber-physical system's architecture and on UML/P class diagrams from which the digital twin information system is generated. We show the practical application and feasibility of our method on an IoT case study. Explicitly modeling the integration of digital twins and cyber-physical systems eliminates repetitive programming activities and can foster the systematic engineering of digital twins.

CCS CONCEPTS

• **Software and its engineering** → *Architecture description languages*; **Integration frameworks**; • **Computer systems organization** → *Embedded and cyber-physical systems.*

KEYWORDS

Model-Driven Software Engineering, Cyber-Physical Systems, Digital Twins, Information Systems, Software Architecture

ACM Reference Format:
Jörg Christian Kirchhof, Judith Michael, Bernhard Rumpe, Simon Varga, Andreas Wortmann . 2020. Model-driven Digital Twin Construction: Synthesizing the Integration of Cyber-Physical Systems with Their Information Systems. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3365438.3410941

MODELS '20, October 18–23, 2020, Virtual Event, Canada

ACM ISBN 978-1-4503-7019-6/20/10…$15.00
https://doi.org/10.1145/3365438.3410941

1 INTRODUCTION

Motivation. There is an increasing demand for the fast and agile creation of digital twins [56, 67], namely digital representations of cyber-physical systems (CPSs), in a variety of disciplines, *e.g.,* marine [33, 39, 70], smart manufacturing [68, 69], avionics [35, 41], building information and energy management [21, 31, 40], automotive [10, 11] or health care [15, 32, 37]. Such a digital twin (DT) comprises models, data traces, (aggregated) data representations, and services to represent, monitor, control, or even optimize the observed CPS. Digital twin information systems (DTISs) with a set of graphical user interfaces provide a convenient and effective way to manage a CPS [34]. The DTIS would be responsible for displaying the data and allowing for interaction with both users and the CPS. The CPS then handles all the cyber-physical elements and shares its data with the DTIS. As such, DTISs can serve as viable bases for representing and monitoring CPSs, *i.e.,* acting as their DTs. Clearly, the DTIS and CPS have to share a great number of interfaces to be able to communicate about data and models.

Open Challenges. Until now, large parts of the connections between such interfaces had to be crafted manually. These implementation tasks do not require high cognitive performance of the developers but are, due to the number of interfaces, time-consuming, and hence, error-prone. As the tasks and the artifacts to be developed are highly repetitive, this is a good candidate for improvements [64]. Following the idea of model centered architecture [43, 44], models can be used for the flexible definition of any kind of system interfaces and communication. Through making these interfaces and their connections explicit in suitable models, creating these repetitive artifacts can be automated. This improves efficiency and consistency in engineering DTs for CPSs. Although model-driven software engineering (MDSE) provides the necessary methods to generate these connections, these methods have not yet been applied to integrated development and connection of DTs to CPSs.

Contribution. In this paper, we address the challenge of reducing the effort for engineering the communication interfaces between cyber-physical systems and digital twins implemented as information systems. To this end, the paper conceives a model-driven method for the integration of CPSs and DTISs using a novel, domain-specific tagging language that decouples the development of both systems. This separates the concerns involved, and many related development tasks can then be fully automated.

Our contribution, hence, consists of

- A development process for the model-driven integration of CPSs and DTISs.
- A model-driven solution for the generative extension of architecture models and class diagrams (CDs) with elements that keep their data synchronized.

- A method for clearly separating business logic and synchronization infrastructure in model-driven systems using DTs.

Structure of the paper. In the following, Section 2 introduces preliminaries on concepts, modeling languages, and tools used in the remainder. Section 3 presents the requirements for our system. Section 4 introduces our running example, the automatic fire extinguishing system, and shows how it can be represented with different types of models. Section 5 presents how to enhance models with further information about component communication and how to generate the synchronization infrastructure between a DTIS and the corresponding CPS. Section 6 shows the application of our method in a case study. Section 7 relates our approach to other approaches and Section 8 discusses it. Section 9 concludes.

2 BACKGROUND

This section introduces our notion of digital twins, the MontiArc architecture description language, which we leverage to model the architecture of CPSs, the MontiGem code generation framework for the efficient engineering of DTISs, and the tagging language framework used to combine the CPSs with the DTISs.

2.1 Digital Twins

DTs are digital representations of cyber-physical assets or processes that enable advanced control, decision making, and optimization. They are used in a variety of domains, including avionics, automotive, and smart manufacturing [12, 26, 68].

While the use of DTs promises to improve the use of CPSs in many ways, intensional definitions of DTs are rare and vague, such as (1) "An always in sync digital model of existing manufacturing cells throughout the life cycle" [66], (2) "[…] virtual product models, which are frequently referred to as DTs" [60], or (3) "[…] a set of virtual information constructs that fully describes a potential or actual physical manufactured product from the micro atomic level to the macro geometrical level" [26]. Such approaches to definitions often use the term model—opposed to the commonly accepted definition of Stachowiak [63]—in a sense that the reduction property (*i.e.,* the model is an abstraction of the original for a specific purpose) cannot be adhered to. Often, these definitions also focus on very specific applications, such as "manufacturing cells" or "product models." Hence, a commonly accepted definition still is lacking.

Based on a joint effort within the German "Internet of Production"[1] cluster of excellence research project of 200 researchers of 25 departments conducting research in artificial intelligence, computer science, innovation research, labor science, mechanical engineering, and production technology [61], we conceived the following definition on the constituents of DTs that is liberated from specific applications, focuses on its contents, and separates data and models:

A DT of a system consists of a set of models of the system, a set of contextual data traces and/or their aggregation and abstraction collected from a system, and a set of services that allow using the data and models purposefully with respects to the original system.

From this, it follows that (1) A DT is not a model itself: instead it comprises models of the system it represents. These can be the engineering models used to build the developed system, models derived from these, or abstractions of the data traces observed by the DT. (2) A DT can be made active by invoking its services, which may comprise databases, user interfaces, analyses, and even the interaction with other systems. (3) A DT leverages its models and data traces to converge the observations coming from the represented system and from itself.

[1]Internet of Production: https://www.iop.rwth-aachen.de/

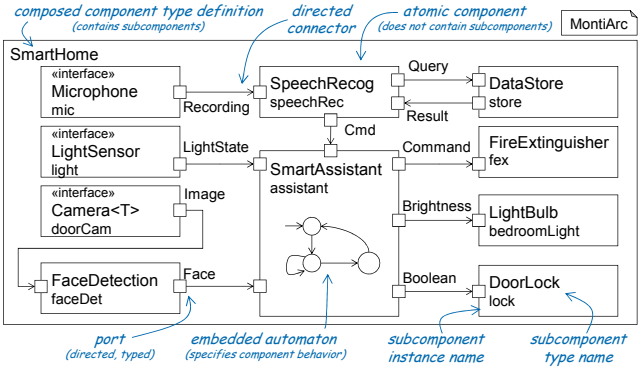


Figure 1: MontiArc architecture of a smart home system

This supports the investigation of a variety of DTs, such as development digital twins used during the development of the (to be) represented system, usage DTs that represent the system as operated, diagnostic DTs that support detailed analysis of the represented system in its context, and many more.

In this respect, our contribution focuses on efficiently modeling DTs comprising data structures representing properties of CPSs by relating interfaces of the CPSs' architecture models to data structure properties. The data structures are used by a DTIS that may aggregate and abstract this data prior to visualization and further use. These DTs offer services through their software architecture as well as through human interaction with the DTIS.

2.2 MontiArc

MontiArc [18, 27] is an extensible [17, 57] architecture description language [45] for the efficient engineering of CPSs. The language comprises modeling elements for atomic and composed component types that exchange messages via the directed and typed ports of their interfaces. Atomic component types yield embedded behavior (*e.g.,* automata) models or general-purpose language (GPL) implementations that define their behavior directly. Composed components are hierarchically composed and yield topologies of subcomponents. Hence, their behavior emerges from their subcomponents' behavior.

Figure 1 illustrates MontiArc's most important modeling elements on the software architecture for a smart home: the system boundaries are defined by the `SmartHome` component type that contains ten subcomponents of different component types, such as the subcomponent `mic` of component type `Microphone`. The subcomponents `mic`, `light`, and `doorCam` sense the smart home's environment and send their data either into post-processing components (such as `speechRec` or `faceDet`) or directly into the central component `assistant`. Based on these inputs, the central controller, *i.e.,* the `assistant`, decides on the next actions and activates the actuators `fex`, `bedroomLight`, and `lock` on the right.

MontiArc supports various features to facilitate architectural programming, such as generic data type parameters for component types, interface components, component type inheritance, component parametrization, injection of component instances into composed components, or dynamic reconfiguration [29]. In Figure 1, the component type `Camera`, *e.g.,* defines a generic data type parameter that can be used to define the data type of its single outgoing port. For the component instance `doorCam`, this parameter bound to the data type `Image`. `Camera`, as well as the component types `Microphone` and `LightSensor`, also is an abstract component type that does not yield an implementation by design. Instead, this type is replaced by a platform-specific component type that extends it and yields a specific implementation before deployment.

MontiArc models can be translated to Java [58] for educational purposes, to Python [4] for service robotics, and to C++ for Internet of Things (IoT) systems. Through modular language engineering, the MontiArc language and its code generation capabilities can be extended with novel language elements and transformations [17].

2.3 MontiGem

MontiGem [6, 23], the generator framework for enterprise information systems, uses models to generate complete (enterprise) information systems [24]. Different UML/P [59] languages, such as CDs and the object constraint language (OCL), are used as sources. Further domain-specific languages (DSLs) are incorporated for code generation such as the GuiDSL, a graphical user interface (GUI) description language. Using these models, MontiGem generates the data structure, database schema, and (web-)pages, including corresponding data views (ViewModels). Together with the basic runtime environment for the frontend, *i.e.,* the user interface, and the backend, *i.e.,* the data processing, of the information system (IS), the generated code forms an executable application which is extendable by handwritten code.

We derive the database schema and data structure in front- and backend from CDs. This ensures consistency between front- and backend by construction. We use OCL as a restriction language on the data structure and generates validators for data inputs. Commands handle the communication between front- and backend and also depend directly on the CD input. Additional structure and behavior commands can be defined. GUI models describe the layout of the generated (web)page, as well as the used ViewModels. Those ViewModels map the data structure to specified GUI models enabling the generation of views with specific extracts from the data. This enables defining the ViewModels in place, where they are to be displayed. To improve usability and speed up the development process, a set of standard GUI models does not need to be defined manually but can be generated based on the domain models (CDs). This provides an overview of all used data classes but still allows for adaption and extension of the (web)pages using handwritten GUI models and/or code. Additionally, we use a tagging language [25, 42] to enrich the domain model described in the CD. This DSL enables the use of different generator configurations, *i.e.,* what should be generated, or adds implementation-specific information to CD or GUI models.

The MontiGem generator framework enables the generation of a complete IS using only domain-specific CDs citeGMN+20 but

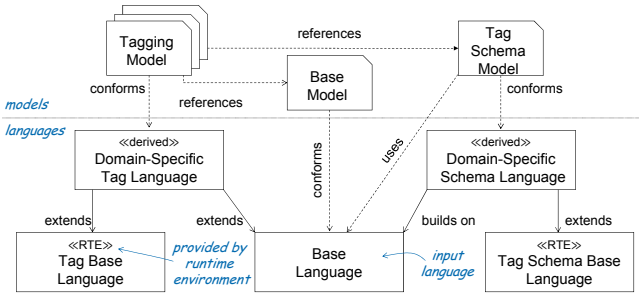


Figure 2: Architecture of the tagging languages based on [25]

allows to use further DSLs for detailed behavior. To allow the inclusion of further DSLs, *e.g.,* behavior and goal models [49] or privacy concepts [48] is in discussion. By now, the resulting IS presents stored data and provides operations to create, edit, or delete data sets. MontiGem is used in several application areas, such as finance cockpits [7, 23], IoT or energy management dashboards. Each specific implementation adapts and extends the generated code with domain-specific logic and additional functionality.

2.4 Tagging

In this work, we use tagging to connect CPSs with their DTs. Tagging [25] is a language engineering technique that enables the non-invasive annotation of existing models of a given base language through models of a domain-specific tagging language automatically derived from the base language. Through this, domain experts can reuse established syntax of the base language in the tagging models for annotating it and do not need to convolute the base model with these annotations. In consequence, this increases the reusability of the base models.

As depicted in Figure 2, tagging is based on a common tag base language (bottom left), which predefines various tag types, and a common tag schema base language (bottom right), which prescribes the structure of tag schemata. Based on these and the base language (bottom middle), the tagging code generators derive a domain-specific schema language and a domain-specific tag language. Models of the former govern the type, number, and shape of tags in conforming tag models. This, *e.g.,* enables annotating the base models with non-functional properties [42] or adding communication information [20]. In general, for a single base language, multiple tag schemata can be defined, and models of the domain-specific tagging language then are validated against the schema they reference. Models of the domain-specific tag language refer to a base model they annotate and to a tag schema model they conform to.

3 REQUIREMENTS

Within the last decade, we have gained experience in various domains including avionics [36, 72], automotive [9, 22], robotics [5, 57], smart homes [50, 65], and manufacturing [47, 48, 71]. These domains are facing the same challenges in creating a connection between a CPS and a DTIS. To automate engineering of these connections, we identified the following requirements based on an analysis of popular IoT tools such as Arduino IoT Cloud, Amazon AWS, or Microsoft Azure (see Section 7 for details):

(R1) *The CPS and the DTIS shall be able to synchronize any data type known to both the CPS and the DTIS.* Until now, integrating CPS and DTIS demands for error-prone handcrafting to map each datatype from one system to the other one from various languages.

(R2) *The communication infrastructure that keeps the CPS and the DTIS synchronized shall be completely generated from corresponding models.* Until now, the integration of both demands repetitive handcrafting and is error-prone.

(R3) *The handwritten artifacts (e.g., models, code) specifying the CPS and DTIS shall not contain information about their integration and the integration of the systems shall not presuppose the content of the handwritten artifacts.* Component developers and system architects of the CPS and frontend and backend developers of the DTIS should be able to work independently on the design of these systems, including modeling aspects. R3 ensures the independent modeling of CPS and DTIS. In addition, R3 ensures that the integration can be applied to legacy artifacts that were not created with DTs in mind.

(R4) *The DTIS shall enable users to manually override the specified behavior of the CPS temporarily or permanently.* This is important to be able to handle exceptional situations. Thus, a user's manual intervention should be possible and override the automatic behavior of the system.

(R5) *The CPS should support heterogeneous platforms as long as they can communicate with the Internet.* To work with platform-independent versions offers hardware flexibility.

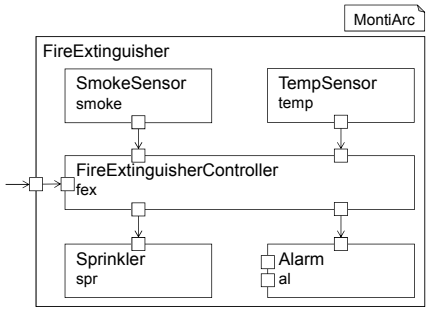
The following sections discuss each of these requirements in detail and show how these requirements are met.

4 EXAMPLE: AUTOMATIC FIRE EXTINGUISHING SYSTEM

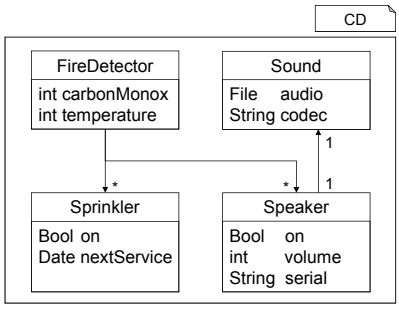
In IoT environments, systems need to interact with the real world and connect to DTISs to receive the goals of their users. In the following, we use an automatic fire extinguisher in a smart home environment (*cf.* Figure 1) as a running example. This example is motivated by Google's fire alarm system *Nest Protect*[2]—though our example is simplified for better comprehension. Our simplified version of the architecture is based on the fire alarm architecture shown in [46]. Figure 3 depicts the models used to specify this automatic fire extinguishing system: (a) the MontiArc architecture of the CPS and (b) a CD describing the DTIS's data structures.

The CPS architecture shows two sensors (top), a central controller (middle), and two actuators (bottom). The sensors measure the carbon monoxide concentration in air and the current temperature. This raw data is sent to the central controller, which then decides whether there is a fire or not. If a fire is detected, the controller can react by turning on the sprinklers or triggering a fire alarm. To do so, the controller sends commands to the actuators via its outgoing ports and the attached connectors. While the sprinkler only needs to be prompted to switch on, the `Alarm` component also requires a sound file with the alarm tone and a volume level at which the alarm should be played. The architect, however, did

[2]Nest Protect: https://store.google.com/product/nest_protect_2nd_gen_specs



(a) Underspecified CPS architecture describing a fire extinguishing system that detects fire based on smoke and temperature sensors and uses this information to trigger an alarm and turn on a sprinkler



(b) Domain model of the DTIS describing the data structure used to monitor the fire extinguishing system in online platform

Figure 3: Automatic fire extinguisher system. The MontiArc model (a) describes the logical software architecture of IoT devices. The CD (b) describes the IS data model.

not specify how this information should be provided. This underspecification is reflected by the two ports on the left side of the `alarm` component that are not connected to another component. Allowing such underspecification is crucial in the development process as it allows to defer design decisions to a later stage of the development process where more information about the system is available. However, to generate code from the architecture model, the gaps resulting from underspecification have to be filled.

The DTIS domain model shows four data classes that might be used in a DT of the CPS. For example, turning up the `volume` of the `Speaker` in the domain model should cause the volume of the real `Alarm` to increase **(R1)**. Similarly, if the temperature sensor detects a temperature change, the temperature information in the DTIS needs to be updated. While the DTIS's domain model represents a view on the same system, the data structure is different, as the DTIS may contain information that is not required by the CPS architecture, omit data used by the CPS, and structure the data differently. For example, the DTIS domain model also includes a `Date nextService` storing the due date of the next required maintenance. Though this might be valuable information to the user who interacts with the DTIS, the sprinklers themselves do not need this information.

The two models are used as input for MontiArc and MontiGem to generate code that is executed on the IoT devices and in the backend of the DTIS. However, these models do not define the interfaces between the CPS architecture and DTIS, *i.e.,* the CPS does not know how to exchange data with the DTIS and vice versa.

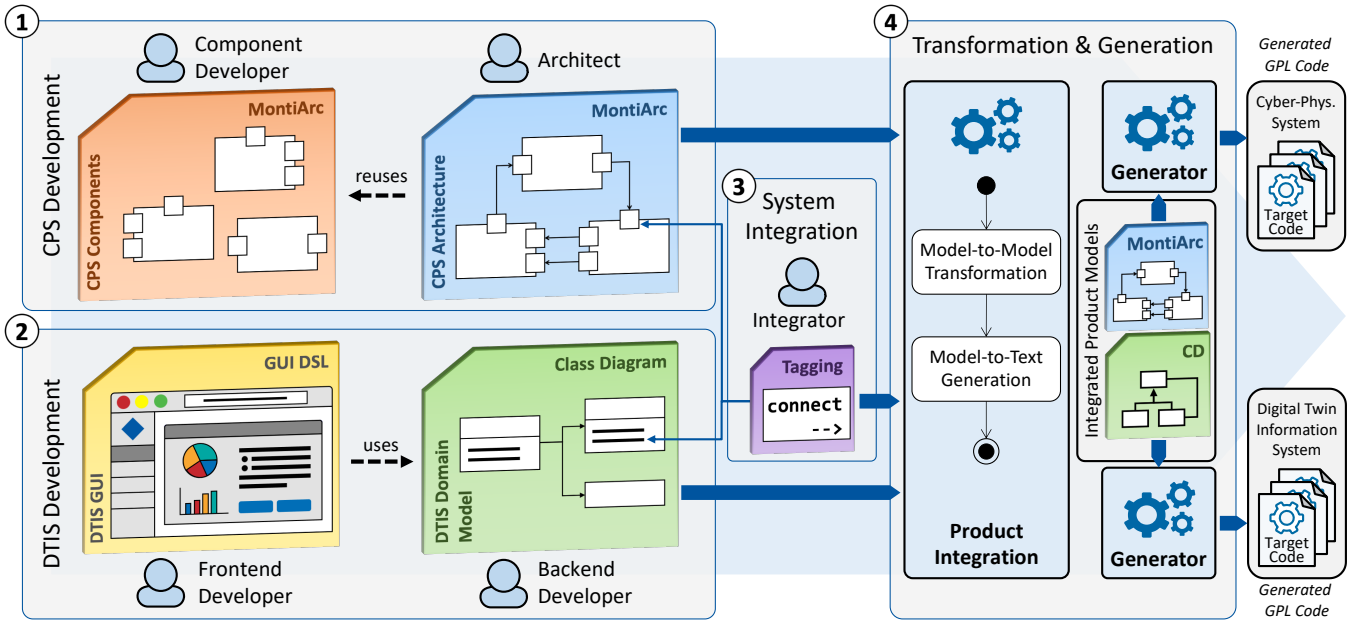


Figure 4: Process overview: The CPS architecture and the IS can be developed independently of one another. The integrator connects the models by tagging elements in the architecture and the IS. Based on these connections, model-to-model transformations create the necessary interfaces between the architecture and the IS.

Implementing connections between both systems is a repetitive and time-consuming task. Clearly, the automatic generation of such interfaces and their automatic integration into existing systems is an attractive option for the development of such systems **(R2)**.

5 INTEGRATING CYBER-PHYSICAL AND INFORMATION SYSTEMS

Our process for developing integrated CPSs and DTISs consists of four activities (*cf.* Figure 4), the first two of which can be performed in parallel: (1) Developing the CPS architecture; (2) Developing the DTIS; (3) Integrating the CPS with the DTIS; and (4) Generating the CPS and DTIS.

The first two steps consist of developing a set of models from which the software running on the CPS's devices and the DTIS can be generated. The two systems can, but do not have to, be developed independently of each other **(R3)**. As the systems may be developed independently of each other, our process can be applied to already existing systems as well as to greenfield, *i.e.,* newly developed, systems including both the development of the CPS and the DTIS. In the third step, the models generated in the first two steps are integrated. The fourth step generates GPL code from the models. This is fully automated.

Step (1): The CPS architecture development starts with developing a set of reusable software components—in our case using the MontiArc architecture description language. Next, the architect connects the components to create an integrated architecture of the CPS. While first developing a set of reusable components independently of the architecture is useful, it is not required to apply our method. It is also possible to start developing architectures for specific products and then later decide which components are worth maintaining independently of the product.

Step (2): The DTIS development includes the development of the front- and backend. The frontend depends on accessing data provided by the backend. Nevertheless, the front- and backend can be developed (partly) in parallel. In our case, the DTIS is developed using MontiGem, *i.e.,* we use class diagrams to describe the domain model, *i.e.,* the data structures used by the backend.

Step (3): Once the CPS architecture and the DTIS have been developed, the integrator tags ports of the CPS architecture with attributes of the domain model and vice versa. This tagging is conceptually based on [25]. Section 5.1 describes this in more detail.

Step (4): Using the tagging created in Step 3, the CPS architecture, and the DTIS's data model as input, a model-to-model transformation extends the CPS and DTIS by the necessary communication and synchronization infrastructure. This step is fully automated **(R2)**. The process for transforming the CPS architecture is described in Section 5.2, and the process for transforming the DTIS is described in Section 5.3. The transformation results in integrated product models using the same MontiArc and class diagram languages that were used by the input models. This allows forwarding the resulting models of the integrated CPS and DTIS to MontiArc's and MontiGem's code generators that produce the necessary GPL code to be executed on the CPS devices and the server that provides the DTIS.

5.1 Tagging CPS Architectures and IS Domain Models

DTs need to stay synchronized with the original system. While the logic that the DTIS may apply to the data of the DT is application-specific, the task of keeping the data values of two systems in sync is repetitive and, therefore, automatable. If both the CPS and the DTIS are developed in a model-driven fashion, tagging can be

utilized to select the model elements of both systems that should stay synchronized.

The tagging model serves two purposes: First, it specifies how to match DTs, *i.e.,* objects instantiated from the domain model, to the physical devices they represent. Secondly, it specifies which ports of the CPS architecture should connect to which attributes of the domain model. Ports are architecture elements that components use to exchange data with other components. Hence, they are ideal candidates for injecting data into or extracting data from the system. Attributes in the domain model are used to store the actual data values. Accordingly, tagging attributes of the domain model and connecting them to ports of the architecture allows specifying which data reflects the state of the CPS. Figure 5 shows such a tagging model.

The two purposes of the tagging model leave two tasks for the integrator (Figure 4) who is responsible for connecting the CPS and the DTIS: (1) Identify which attributes of the domain model are designed to store device identifiers or detect there is no such attribute for a certain device type. (2) Select which attributes of the objects instantiated from the domain model should be synchronized to which ports of the CPS architecture.

For the first task, the integrator needs to find out which attributes of the domain model are intended to store device identifiers of the CPS. If the integrator finds an attribute that stores a device identifier, (s)he specifies it as an identifier in the tagging (ll. 4-5 of Figure 5). Manually specifying the identifier enables the user of the DTIS to create digital twins for future devices. If the user sets the values of these attributes to hardware identifiers of the devices, the system can match the actual devices to their DTs once the devices first go online. Here, objects of the `Speaker` class are identified by the attribute `Speaker.serial`, which stores the serial numbers of the physical devices. Objects of the Sound class are identified by the `serial` attribute of the `Speaker` object, which references the Sound. This is possible as there is exactly one `Speaker` for every Sound (Figure 3(b)).

If the integrator does not find an attribute that stores a device identifier, (s)he can choose to automatically instantiate DTs for devices once they first connect to the DTIS (l. 8 of Figure 5). The `auto identify` keyword is followed by the qualified name of a class from the domain model. This class is then instantiated every time a device with a port that should be synchronized to one of the class's attributes first connects to the DTIS. Each physical device is identified by a unique hardware-specific identifier of the device, *e.g.,* the MAC address of the network interface. The DTIS uses this identifier to connect the ports of the architecture to the new instance of that class, but the identifier will not be part of the data model.

To ensure that any communication with a device is always assigned to the same digital twin, the DTIS must know a permanent device identifier for each device. For the second task, the integrator needs to specify which attributes of the domain model should be synchronized to which ports of the architecture. The remaining lines of Figure 5 (ll. 9-18) show how to connect ports of the CPS architecture to attributes of the domain model of the DTIS and vice versa. If data from the CPS architecture should be sent to the DTIS, the CPS architecture will update the DTIS whenever a new message is sent through the port. For this, the integrator specifies

Tagging Model

```
1  // Objects of the Sound and Speaker classes serve as
2  // digital twins for CPS devices that use the value of
3  // Speaker.serial as identifier
4  identify Sound    by attr Speaker.serial
5  identify Speaker  by attr Speaker.serial
       class from domain model
       class from domain model. Same as before or reachable via to-1-association.
       attribute of the class before the dot. Stores device identifier.

6  // Automatically create and link a digital twin when
7  // a device first connects to the IS
8  auto identify FireDetector
       class name from domain model.

9  // Send data from the CPS architecture to the IS
10 connect port smokeSensor.value
11     --> attr FireDetector.carbonmonoxy
       architecture component instance   outgoing port
12 connect port temperatureSensor.value
13     --> attr FireDetector.temperature
       class name from domain model   attribute name
       "Use messages from temperatureSensor's port 'value' to update the temperature attribute of the FireDetector class"

14 // Send data from the IS to the CPS architecture
15 connect attr Sprinkler.on          --> port sprinkler.on
16 connect attr Speaker.on            --> port alarm.on
17 connect attr Speaker.volume        --> port alarm.volume
   "If 'Speaker.volume' changes, inject the new value into alarm's 'volume' port"
18 connect attr Speaker.sound.audio --> port alarm.sound
       class name from domain model   (association +) attribute   component instance   port
```

Figure 5: Tagging model connecting the architecture and the domain model from Figure 3. Ports of the architecture are mapped to attributes of the domain model and vice versa.

the sending port and the receiving attribute (ll. 9-13). To this effect, a message containing the fully qualified name of the port, the identifier of the device, and the new value is generated and sent to the DTIS. Inversely, if updates of an attribute are to be forwarded to a port of the CPS (ll. 14-18), the DTIS generates such a message whenever the attribute in the DTIS is updated and sends it to the corresponding device.

5.2 Cyber-Physical System Architecture Transformation

Our method leverages model-to-model transformations to extend (possibly underspecified) CPS architectures by components that carry out the communication and synchronization with the DTIS. Figure 6 conceptually depicts this transformation. The input architecture consists of a sensor, a controller, and an actuator. The tagging defines with ports need to send data to or receive data from the DTIS. Accordingly, for each such tag, a component is generated. The transformation distinguishes between three cases: Tagging (1) outgoing ports, (2) incoming ports without connectors, and (3) incoming ports with connectors.

Case (1): The tagged port is an outgoing port that should send data from the architecture to the DTIS. The generated sender component (Figure 7) has an incoming port that takes data of the type given to the component as a generic type parameter **(R1)**.

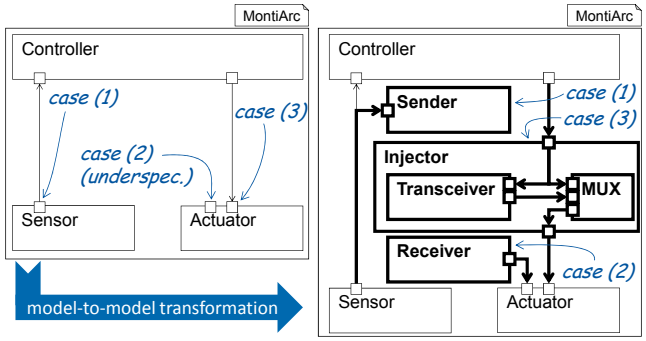


Figure 6: Architecture transformation that inserts components to keep the system synchronized with the information system. Generated elements illustrated in bold. Generic parameters omitted from all generated elements for better readability.

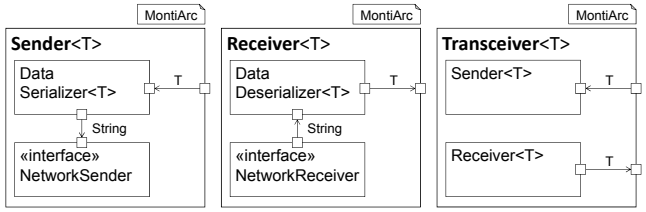


Figure 7: Prototypes for generated composed components that handle the communication with the IS.

The generator uses the type of the tagged port to instantiate the sender component and, thus, ensures matching types. A generated connector then connects the tagged port to the generated component. As soon as the generated sender component receives data, it serializes and forwards the data, together with identifiers of the emitting device and port, to the DTIS. The `Sender` component specifies the `NetworkSender` as an interface component. This component is exchanged during the deployment process by a hardware platform-specific version. This ensures portability across hardware platforms **(R5)** by exchanging the interface components that require platform-specific network functionalities by platform-specific variants during deployment of the architecture.

Case (2): The tagged port is an incoming port without connections. The process for injecting data into the architecture is inverse to the process of extracting data: If the tagged port should process data from the DTIS, the generated receiver component (Figure 7) has an outgoing port with a generic type **(R1)**. The generator again uses the type of the tagged port to instantiate the generated component and, thus, guarantees that the type of the port of the receiver component matches the tagged port. A generated connector connects the outgoing port of the generated receiver component to the tagged (incoming) port. If the `NetworkReceiver` component inside the generated receiver component receives data from the DTIS, it creates a message on its outgoing port that is then deserialized and forwarded to the tagged port.

Case (3): The tagged port is an incoming port with a connector. In this case, a more complex injector component is generated that replaces the connector and synchronizes incoming messages with the DTIS. This injector has two subcomponents: A transceiver and a multiplexer (MUX). The transceiver can be realized by combining

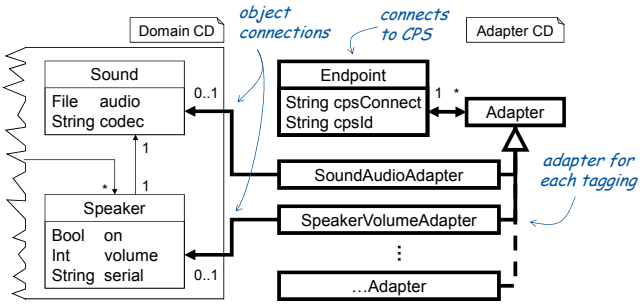


Figure 8: Additional endpoint for the example domain (Figure 3(b)). Each mapping has its own adapter to transform given data. Generated elements illustrated in bold.

the generated sender and receiver components from the previous two cases (Figure 7). This is done by using them as subcomponents of the transceiver, where connectors forward the data to or from the ports of the enclosing transceiver component. The generated sender and receiver keep the device synchronized with the DTIS.

The MUX gives manual decisions of the user priority over automatic decisions by the CPS, as the user's manual intervention expresses the explicit decision to override the automatic behavior of the system **(R4)**. If the MUX receives valid input from both of its incoming ports, the port connected to the DTIS is preferred. The MUX does not accept any value from the original system until the transceiver component explicitly releases the connection by sending an empty message. This is done to prevent immediately overriding the user's messages.

5.3 Information System Transformation

The generated sender and receiver components of the CPS communicate with *endpoints* of the DTIS. Endpoints store the necessary communication-related information about the connection to the CPS (`cpsConnect` in Figure 8), *e.g.*, a socket. Moreover, an endpoint maps a port of the CPS architecture to the respective *adapter* of the DTIS. Adapters are responsible for processing incoming data and monitoring data updates in the DTIS. For each `connect` statement of the tagging, we extend the DTIS with an adapter.

Depending on the tagging, the adapter either processes data received from the CPS (ll. 10-13, Figure 5) or monitors data updates that need to be sent to the CPS (ll. 15-18, Figure 5). An adapter that processes data received from the CPS determines to which object in the data source of the DTIS, *e.g.*, database, the data belongs. Figure 9 describes the process of how an object is loaded. The first step is to find the adapter that knows how to load the data from the data source and which parts (attributes) of the object to load. If the adapter is found, the object can be loaded directly. If the adapter cannot be found, that is, it has not yet been created, it is created together with the object in the DTIS data source and connected to it. Now that the adapter is created, the object can be retrieved. After the loading of the connected data object, the adapter then updates the data source accordingly. An adapter that informs the CPS about data updates listens to changes in the data objects. Changes in data objects can either originate from a user or external sources, *e.g.*, data imports. On every change, the adapter creates a message and forwards the data to the CPS via the endpoint.

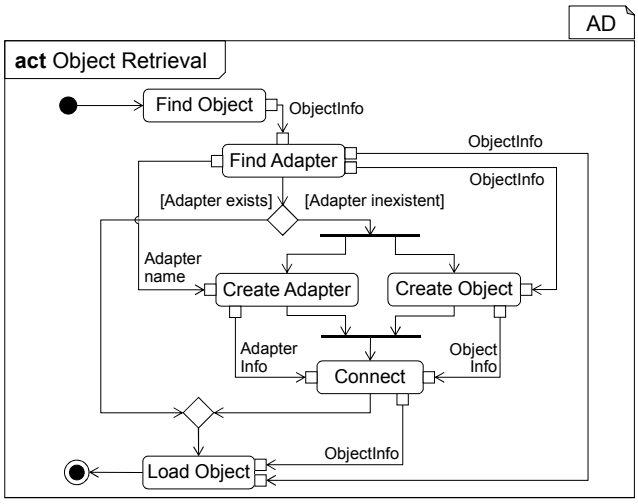


Figure 9: Object retrieval with adapters

Endpoints handle the DTIS's communication with the CPS and can be created either manually by the user or automatically when a CPS device first connects to the DTIS and no corresponding endpoint exists. The connection information of a device (`cpsConnect`) is set when the device first connects to the DTIS. As IoT devices may be mobile and the network topology cannot be assumed to be static, the endpoint updates this information whenever a CPS sends a message to the DTIS. To enable these updates, devices of the CPS include a unique device identifier (`cpsId`), *e.g.,* a name or serial number, in their messages to the DTIS.

This identifier can also be used to link devices to existing data objects when they first connect to the DTIS. To this effect, an attribute in the existing data class can be set in the tagging model (ll. 4-5, Figure 5). This also enables creating DTs of devices even before their counterparts in the CPS first connect to the DTIS. A special variant of this is choosing an attribute of a connected object. This only works for *-to-1-associations as the object has to be uniquely identified. If the identifier of the CPS device is unknown to the DTIS, a new object is created. If the tagging model specifies automatic mapping (l. 8, Figure 5), a new object is created whenever a device first connects.

The application itself does not need to know any of the extensions to the backend data structure, because the data is transformed by the respective adapter to match the internal data structure. The existing data structure or database schema used by the application does not need to be changed; it is only extended to handle the communication (Figure 8). These extensions are non-invasive and do not require the system to know the adapters.

6 CASE STUDY

Our case study uses the fire extinguisher example (Section 4), which is based on Google's *Nest Protect*. The goal is to build an extensible fire extinguisher system that can be monitored and controlled at runtime from an online dashboard. For monitoring the system, sensor values need to be sent to a DTIS. For controlling the system, the state of the actuators and its representation in the DTIS needs to be synchronized and the DTIS should be enabled to influence actuators. From an engineering perspective, the synchronization of the systems should be realizable with as little effort as possible. Since writing high-level communication protocols is a repetitive task, the communication between the DTIS and the CPS should be generated to a large extent instead of handwritten **(R2)**. The CPS was implemented using three Raspberry Pi 4B **(R5)**. Two of them were connected to a gas sensor and a siren, and one was connected to a temperature sensor. All of them executed C++ Code generated from MontiArc models. The sprinklers were only virtually present to prevent damage to our laboratory.

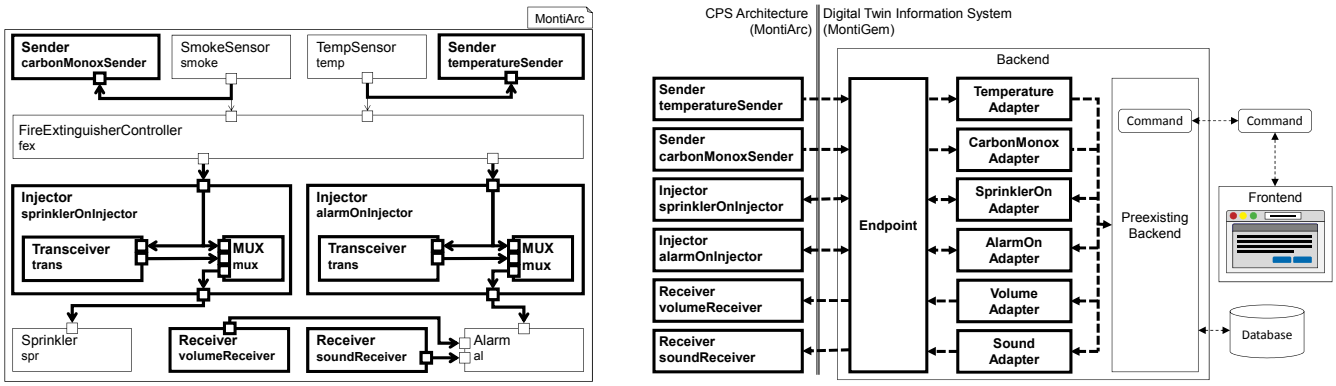
We integrate the architecture model (Figure 3(a)) of the CPS and the domain model (Figure 3(b)) of the DTIS using the tagging model from Figure 5. A major advantage of our approach is that the integration of the two systems only takes nine statements. This not only makes very efficient use of the engineers' time but also enables rapid prototyping of DTs as no time is needed for implementing communication infrastructures to keep the CPS and DTIS in sync.

Using the tagging model, we transform the CPS architecture and extend the DTIS according to this tagging model (*cf.* Section 5.2). Through this, we automatically include appropriate endpoints for the communication with the cyber-physical devices. The resulting architecture (Figure 10(a)) shows generated elements in bold. Clearly, all concepts from the original architecture are still present in the resulting architecture except for the connectors between the controller and the actuators. The two absent connectors are replaced by connectors to the added `Injector` components that tap the data and forward it to the DTIS. The new sender and receiver components connect to the already existing components of the architecture to extract or inject data. Figure 3(a) shows two underspecified ports that leave the decisions open how the `Alarm` component gets the desired volume and the sound played in case of an alarm. The generated `VolumeReceiver` and `SoundReceiver` components eliminate this underspecification. The fact that the original components remain unchanged ensures that the behavior description of the components, which relies on communication via the ports, does not need to be changed.

Figure 10(b) shows the changes in the DTIS based on the specified elements defined by the tagging model (Figure 5). Adapters for each mapping are added to the existing infrastructure of the DTIS. Once again, the additions do not interfere with the system's business logic, but only add the ability to update the internal data objects and send data update from the DTIS to the respective CPS. It is not necessary to modify the system's data- or view-logic. The added parts interact with the preexisting infrastructure. To load data objects which are sent to the CPS, the adapters are used to load the specific data from the database. The adapters then transform the domain data to a format that suits the communication with the CPS. When data is sent from the CPS to the DTIS an adapter transforms the message to an internal data object and stores it in the database.

7 RELATED WORK

Multiple tools support the model-driven engineering of IoT systems. ThingML [28, 51] enables defining devices and their logic using a C-like DSL that is used to generate C, Java, or JavaScript artifacts. Ericsson's Calvin [54] enables defining the architecture of IoT applications in a MontiArc-like syntax using the CalvinScript



(a) Transformed CPS architecture including DT communication infrastructure.

(b) Adapters and DTIS adapter endpoint added to the existing MontiGem DTIS.

Figure 10: Automatic fire extinguisher system. Generated elements are in bold. (Generic type) parameters are omitted for better comprehensibility.

DSL. MDE4IoT [19] uses the Foundational Subset for Executable UML Models (fUML) and the action language for foundational UML (ALF) [53] to describe IoT applications and possible deployments to physical devices. SysML4IoT [30] describes how to develop adaptive IoT systems using a SysML-based DSL. Node-RED [2] provides a graphical editor that allows connecting the in- and outputs of software components. Node-RED comes with a library of predefined components to access, *e.g.,* Twitter or Amazon S3 cloud storage. Similarly, the Ptolomy-based CapeCode [13] offers a user interface for graphically combining software components as reusable building blocks.

While some of these systems enable specifying message exchanges and serialization, they lack mechanisms for automatically synchronizing them to DTISs or defining DTs **(R2)**. Though it might be possible to execute some of the actors defined in the above languages inside the DTIS, none of these languages is designed to define a DTIS. Therefore, synchronization with the DTIS data structures is possible **(R1)**, but requires considerable manual effort **(R2)** and that the DTIS and CPS developers agree on messages or topics for synchronization **(R3)**. Especially, this requires the developer of the CPS architecture to have in-depth knowledge about an already existing DTIS and vice versa. In our approach, only the integrator is required to know both systems and only on a high level of abstraction.

Even with popular IoT solutions, such as IBM's Bluemix Cloud platform, connecting devices as simple as a temperature sensor, can be unnecessarily complicated [38]. Following our method, reading and synchronizing the values of a sensor to the data structure in the DTIS requires only two statements in the model (*cf.* ll. 8-11, Figure 5). Moreover, as all of the above approaches to IoT development platforms are based on components exchanging data with each other, our method could be applied to any of the above systems if the generated sender and receiver components were adapted to the platforms' respective interfaces.

The robot operating system (ROS) [55] serves to develop (distributed) robotic software as collections of loosely connected nodes that perform computations and exchange messages over topics (typed message buses). These topics can exchange messages of complex data types known to participating nodes **(R1)** and a generic communication infrastructure takes care of handling message handling. However, sending and reacting to messages has to be handwritten **(R2)** and requires developers to agree on topics **(R3)**. As such, ROS architectures can already represent small-scale IoT applications. While the communication infrastructure of ROS is generic, the data types that can be communicated are, similar to our approach, defined in (rosmsg) models that resemble C++ structs. From these, platform-specific implementations of the data types are generated. However, ROS does not feature any notion of system representation aside from logging and debugging information that could be considered a representation of the CPS.

AutoFocus 3 [8] is a modeling framework based on the Focus [14] calculus to describe the architectures of embedded systems. As such, it covers modeling from requirements to logical and technical architectures to their deployment. It neither facilitates engineering of DTISs to represent DTs, nor connecting DTs to the modeled CPSs. None of the platforms provide specific infrastructure for users to overwrite values out-of-the-box **(R4)**.

Many popular commercial IoT platforms support the development of digital twins but lack means for the model-driven development and integration of CPSs and DTISs. Examples include the arguably largest cloud providers: Microsoft Azure's "device twin" [3] and Amazon AWS's "device shadow" [1]. Both of them exchange data with the CPSs using a combination of JSON and MQTT to synchronize values known to both the CPS and the cloud **(R1)**. Those messages can also be used for manually overwriting values **(R4)**. Structurally, the DT services offered by Azure and AWS resemble the tripartite division into CPS, DTIS and integration of our development process with one important difference: While AWS and Azure require lots of error-prone low-level programming for communicating with the DTIS and synchronizing values, our model-driven approach can automatically generate this infrastructure **(R2)**. Hence, our approach decouples the business logic of the systems from the communication and synchronization tasks required to create a DTs. Thus, we argue that our systems are easier to understand and maintain as the implementation of the business logic is not cluttered with code needed by the infrastructure **(R3)**. In contrast, the Arduino IoT Cloud takes works at a lower level of abstraction. It allows users to define variables of primitive data

Requirements	This Solution	ThingML [28, 51]	Calvin [54]	MDE4IoT [19]	Node-RED [2]	CapeCode [13]	ROS [55]	AutoFocus 3 [8]	Microsoft Azure	Amazon AWS	Arduino IoT Cloud
	IoT / Architecture Tools										
R1	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	P^1
R2	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
R3	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
R4	✓	✗	✗	✗	✗	✗	✗	✗	✓	✓	P^2
R5	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	P^3

[1] Only primitive data types (optionally with unit)
[2] Users can set values, but devices can immediately reset them
[3] Only Arduino-compatible hardware

Table 1: Comparison to related work. ✓= fulfilled, ✗= not fulfilled, P = partially fulfilled

types online **(R1)** for which it then generates the necessary code to keep them synchronized with the Arduino IoT Cloud **(R2)**. This, however, pollutes the business logic with synchronization-specific code and requires the developers to use the variables defined by the generated code **(R3)**. While all other tools usually only require support for a specific GPL, the Arduino IoT Cloud is limited to Arduino-compatible devices **(R5)**.

While our concept is designed for combining MontiArc and MontiGem, there is no conceptual limitation that forbids adapting the generated MontiArc components to use the interfaces offered by Azure and AWS. Adding support for them only requires implementing cloud-specific `NetworkSender` and `NetworkReceiver` components and adapting the `DataSerializer` to use the JSON structures expected by the respective cloud (*cf.* Figure 7).

8 DISCUSSION

One of the main advantages of our solution is the separation of concerns that is achieved by defining communication and synchronization related structures separately from the business logic of the application. This makes the models easier to understand because developers who encounter them for the first time can concentrate on the business logic without being distracted by the technical details of synchronizing values. Also, this enables generating the necessary infrastructure to keep the CPS synchronized with the DTs. This eliminates a repetitive and error-prone task for the developers.

Our separation of concerns comes at the cost of the integrator needing to have a high-level understanding of models for both the CPS and DTIS. While this is a simple task for small systems, it can quickly become complicated as systems become more complex. Therefore, ideally, the integrator is not a single person, but a group consisting of at least one developer of both the CPS and the DTIS. Commercial solutions like the "spatial graph" used by Microsoft Azure's DT require similar roles. In Azure's spatial graph, each device can contain multiple sensors producing data of a certain type. The devices have to be aware of this information and react to requests created based on the information in the spatial graph. This leads to problems if there is a mismatch between the sensors offered by the actual device and the sensors specified in the spatial graph. Since our solution directly utilizes the models used to create the CPS and DTIS, we can detect potential inconsistencies caused by this integration step automatically before deploying the system.

While we think a model-driven approach to developing CPSs and DTISs offers many advantages [16], we do acknowledge that many real-world systems do not use a model-driven approach [62]. DTISs, in particular, are today often programmed by hand. Therefore, it is necessary to leave open the option of communicating with those systems. By allowing customization of the communication mechanisms through abstract components, our solution can also easily be adapted to communicate with popular commercial solutions like Amazon AWS instead of our MontiGem DTIS. This would be done by providing network components (*cf.* Figure 7) that use the APIs of the commercial or handwritten solutions.

In some situations, however, it might also be useful to convert between the data types offered by the CPS and the data types used by the DTIS. For example, the CPS might process values of a temperature sensor given in Fahrenheit, while the DTIS stores temperature in Celsius. Currently, the data types used by the CPS have to match the data types used by the DTIS. As future work, we plan to allow conversions and transformations that are applied during the synchronization.

Furthermore, the logic of the synchronization can be further investigated. Currently, the CPS gives priority to the messages coming from the DTIS. However, the component that sends a message to the tagged port is not aware of this process and therefore does not change its behavior. Thus, it would immediately overwrite the value set by the user in the DTIS. To prevent this, the user can (temporarily) lock a value in the DTIS. As long as the lock is set, messages from the CPS are then ignored in favor of the last value set in the DTIS for this port. This prevents user-set values from being overwritten by the CPS. This process, however, may not be desired for all use cases. If the CPS should adapt its behavior to match the user-set values, a more complex synchronization is required.

Moreover, our evaluation only shows the general feasibility of the approach. For productive use, further investigations regarding the scalability would be necessary. To ensure scalability on the server side, common load distribution methods can be used. On the CPS device side, the available processing power and network bandwidth limit the number of values synchronized with the DTIS.

9 CONCLUSION

Creating DTs for a system comprises creating models of the system, means to process and represent data received from that system, and connecting the represented system to its DT. The latter usually involves manually programming the connection using a communication framework of choice, such as MQTT [52]. This is tedious, error-prone, and complicates the analysis of connections. Our method to connect DTs with DTISs facilitates their integration and separates concerns in DT development by decoupling the development of the CPS architecture and the DTIS. The generated infrastructure consists of consistent-by-construction interfaces between CPS and DTIS that synchronize both systems and accelerates developing DTs for CPSs. Overall, explicitly modeling the integration can facilitate the systematic engineering of DTs.

REFERENCES

[1] Device Shadow Service for AWS IoT. [Online]. Available: https://docs.aws.amazon.com/iot/latest/developerguide/iot-device-shadows.html. Last checked 28. April 2020

[2] Node-RED—Low-code programming for event-driven applications. [Online]. Available: https://nodered.org. Last checked 28. April 2020

[3] Understand and use device twins in IoT Hub. [Online]. Available: https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-device-twins. Last checked 28. April 2020

[4] Adam, K., Butting, A., Heim, R., Kautz, O., Rumpe, B., Wortmann, A.: Model-Driven Separation of Concerns for Service Robotics. In: Int. Workshop on Domain-Specific Modeling (DSM'16). pp. 22–27. ACM (October 2016)

[5] Adam, K., Butting, A., Kautz, O., Rumpe, B., Wortmann, A.: Executing Robot Task Models in Dynamic Environments. In: Proc. of MODELS 2017. Workshop EXE. CEUR 2019 (September 2017)

[6] Adam, K., Michael, J., Netz, L., Rumpe, B., Varga, S.: Enterprise Information Systems in Academia and Practice: Lessons learned from a MBSE Project. In: 40 Years EMISA: Digital Ecosystems of the Future: Methodology, Techniques and Applications (EMISA'19). LNI, vol. P-304, pp. 59–66. Gesellschaft für Informatik e.V. (May 2020)

[7] Adam, K., Netz, L., Varga, S., Michael, J., Rumpe, B., Heuser, P., Letmathe, P.: Model-Based Generation of Enterprise Information Systems. In: Fellmann, M., Sandkuhl, K. (eds.) Enterprise Modeling and Information Systems Architectures (EMISA'18). CEUR Workshop Proceedings, vol. 2097, pp. 75–79. CEUR-WS.org (May 2018)

[8] Aravantinos, V., Voss, S., Teufl, S., Hölzl, F., Schätz, B.: AutoFOCUS 3: Tooling Concepts for Seamless, Model-based Development of Embedded Systems. ACES-MB&WUCOR@ MoDELS **1508**, 19–26 (2015)

[9] Bertram, V., Maoz, S., Ringert, J.O., Rumpe, B., von Wenckstern, M.: Component and Connector Views in Practice: An Experience Report. In: Conf. on Model Driven Engineering Languages and Systems (MODELS'17). pp. 167–177. IEEE (September 2017)

[10] Biesinger, F., Weyrich, M.: The facets of digital twins in production and the automotive industry. In: 23rd Int. Conference on Mechatronics Technology (ICMT). pp. 1–6. IEEE (2019)

[11] Blech, J.: Towards digital twins for the description of automotive software systems. Electronic Proceedings in Theoretical Computer Science **312**, 20–28 (Jan 2020). https://doi.org/10.4204/eptcs.312.2

[12] Boschert, S., Rosen, R.: Digital twin—the simulation aspect. In: Mechatronic futures, pp. 59–74. Springer (2016)

[13] Brooks, C., Jerad, C., Kim, H., Lee, E.A., Lohstroh, M., Nouvelletz, V., Osyk, B., Weber, M.: A Component Architecture for the Internet of Things. Proc. of the IEEE **106**(9), 1527–1542 (September 2018)

[14] Broy, M., Stølen, K.: Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement. Springer Heidelberg (2001)

[15] Bruynseels, K., Santoni de Sio, F., van den Hoven, J.: Digital twins in health care: Ethical implications of an emerging engineering paradigm. Frontiers in genetics **9**, 31 (2018). https://doi.org/10.3389/fgene.2018.00031

[16] Bucchiarone, A., Cabot, J., Paige, R.F., Pierantonio, A.: Grand challenges in model-driven engineering: an analysis of the state of the research. Software and Systems Modeling **19**(1), 5–13 (2020). https://doi.org/10.1007/s10270-019-00773-6

[17] Butting, A., Haber, A., Hermerschmidt, L., Kautz, O., Rumpe, B., Wortmann, A.: Systematic Language Extension Mechanisms for the MontiArc Architecture Description Language. In: European Conf. on Modelling Foundations and Applications (ECMFA'17). pp. 53–70. LNCS 10376, Springer (July 2017)

[18] Butting, A., Kautz, O., Rumpe, B., Wortmann, A.: Architectural Programming with MontiArcAutomaton. In: In 12th Int. Conf. on Software Engineering Advances (ICSEA'17). pp. 213–218. IARIA XPS Press (May 2017)

[19] Ciccozzi, F., Spalazzese, R.: MDE4IoT: Supporting the Internet of Things with Model-Driven Engineering. In: 10th Int. Symp. on Intelligent Distributed Computing (October 2016)

[20] Dalibor, M., Jansen, N., Kirchhof, J.C., Rumpe, B., Schmalzing, D., Wortmann, A.: Tagging Model Properties for Flexible Communication. In: Proc. of MODELS 2019. Workshop MDE4IoT. pp. 39–46. IEEE (September 2019)

[21] Francisco, A., Mohammadi, N., Taylor, J.: Smart city digital twin-enabled energy management: Toward real-time urban building energy benchmarking. Journal of Management in Engineering **36**(2), 04019045 (2020). https://doi.org/10.1061/(ASCE)ME.1943-5479.0000741

[22] Gatto, N., Kusmenko, E., Rumpe, B.: Modeling Deep Reinforcement Learning Based Architectures for Cyber-Physical Systems. In: Burgueño, L., Pretschner, A., Voss, S., Chaudron, M., Kienzle, J., Völter, M., Gérard, S., Zahedi, M., Bousse, E., Rensink, A., Polack, F., Engels, G., Kappel, G. (eds.) Proc. MODELS 2019. Workshop MDE Intelligence. pp. 196–202 (September 2019)

[23] Gerasimov, A., Heuser, P., Ketteniß, H., Letmathe, P., Michael, J., Netz, L., Rumpe, B., Varga, S.: Generated Enterprise Information Systems: MDSE for Maintainable Co-Development of Frontend and Backend. In: Michael, J., Bork, D. (eds.) Companion Proceedings of Modellierung 2020 Short, Workshop and Tools & Demo Papers. pp. 22–30. CEUR Workshop Proceedings (February 2020)

[24] Gerasimov, A., Michael, J., Netz, L., Rumpe, B., Varga, S.: Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In: Anderson, B., Thatcher, J., Meservy, R. (eds.) 25th Americas Conference on Information Systems (AMCIS 2020). pp. 1–10. AIS Electronic Library (AISeL), Association for Information Systems (AIS) (August 2020)

[25] Greifenberg, T., Look, M., Roidl, S., Rumpe, B.: Engineering Tagging Languages for DSLs. In: Conf. on Model Driven Engineering Languages and Systems (MODELS'15). pp. 34–43. ACM/IEEE (2015)

[26] Grieves, M., Vickers, J.: Digital twin: Mitigating unpredictable, undesirable emergent behavior in complex systems. In: Transdisciplinary perspectives on complex systems, pp. 85–113. Springer (2017)

[27] Haber, A., Ringert, J.O., Rumpe, B.: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University (February 2012)

[28] Harrand, N., Fleurey, F., Morin, B., Husa, K.E.: ThingML: A Language and Code Generation Framework for Heterogeneous Targets. In: Proc. of the ACM/IEEE 19th Int. Conf. on Model Driven Engineering Languages and Systems. pp. 125–135. MODELS '16, ACM, New York, NY, USA (2016)

[29] Heim, R., Kautz, O., Ringert, J.O., Rumpe, B., Wortmann, A.: Retrofitting Controlled Dynamic Reconfiguration into the Architecture Description Language MontiArcAutomaton. In: Software Architecture - 10th European Conf. (ECSA'16). LNCS, vol. 9839, pp. 175–182. Springer (December 2016)

[30] Hussein, M., Li, S., Radermacher, A.: Model-driven Development of Adaptive IoT Systems. In: Proceedings of MODELS 2017. Workshop ModComp. vol. 2019, pp. 17–23. CEUR, Austin, United States (Sep 2017)

[31] Jain, P., Poon, J., Singh, J.P., Spanos, C., Sanders, S.R., Panda, S.K.: A digital twin approach for fault diagnosis in distributed photovoltaic systems. IEEE Transactions on Power Electronics **35**(1), 940–956 (2020)

[32] Jimenez, J., Jahankhani, H., Kendzierskyj, S.: Health Care in the Cyberspace: Medical Cyber-Physical System and Digital Twin Challenges, pp. 79–92. Springer International Publishing (2020). https://doi.org/10.1007/978-3-030-18732-3_6

[33] Johansen, S., Nejad, A.: On digital twin condition monitoring approach for drivetrains in marine applications. International Conference on Offshore Mechanics and Arctic Engineering, vol. Volume 10: Ocean Renewable Energy (06 2019). https://doi.org/10.1115/OMAE2019-95152

[34] Josifovska, K., Yigitbas, E., Engels, G.: A digital twin-based multi-modal ui adaptation framework for assistance systems in industry 4.0. In: Kurosu, M. (ed.) Human-Computer Interaction. Design Practice in Contemporary Societies. pp. 398–409. Springer International Publishing (2019)

[35] Kraft, E.: The air force digital thread/digital twin - life cycle integration and use of computational and experimental knowledge. In: 54th AIAA Aerospace Sciences Meeting. American Institute of Aeronautics and Astronautics (2016). https://doi.org/10.2514/6.2016-0897

[36] Kriebel, S., Raco, D., Rumpe, B., Stüber, S.: Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy's Streams Become Feasible? In: Proc. of the Workshops of the Software Engineering Conf.: Workshop on Avionics Systems and Software Engineering (AvioSE'19). vol. 2308, pp. 87–94. CEUR-WS.org (2019)

[37] Laaki, H., Miche, Y., Tammi, K.: Prototyping a digital twin for real time remote control over mobile networks: Application of remote surgery. IEEE Access **7**, 20325–20336 (2019)

[38] Lekić, M., Gardašević, G.: IoT sensor integration to Node-RED platform. In: 17th Int. Symp. INFOTEH-JAHORINA. pp. 1–5 (March 2018)

[39] Liu, J., Zhou, H., Liu, X., Tian, G., Wu, M., Cao, L., Wang, W.: Dynamic evaluation method of machining process planning based on digital twin. IEEE Access **7**, 19312–19323 (2019)

[40] Lu, Q., Xie, X., Heaton, J., Parlikad, A.K., Schooling, J.: From bim towards digital twin: Strategy and future development for smart asset management. In: Borangiu, T., Trentesaux, D., Leitão, P., Giret Boggino, A., Botti, V. (eds.) Service Oriented, Holonic and Multi-agent Manufacturing Systems for Industry of the Future. pp. 392–404. Springer International Publishing, Cham (2020)

[41] Mandolla, C., Petruzzelli, A.M., Percoco, G., Urbinati, A.: Building a digital twin for additive manufacturing through the exploitation of blockchain: A case analysis of the aircraft industry. Computers in Industry **109**, 134 – 152 (2019). https://doi.org/https://doi.org/10.1016/j.compind.2019.04.011

[42] Maoz, S., Ringert, J.O., Rumpe, B., Wenckstern, M.v.: Consistent Extra-Functional Properties Tagging for Component and Connector Models. In: Workshop on Model-Driven Engineering for Component-Based Software Systems (ModComp'16). CEUR Workshop Proceedings, vol. 1723, pp. 19–24 (October 2016)

[43] Mayr, H.C., Michael, J., Ranasinghe, S., Shekhovtsov, V.A., Steinberger, C.: Model Centered Architecture, pp. 85–104. Springer International Publishing (2017)

[44] Mayr, H.C., Michael, J., Shekhovtsov, V.A., Ranasinghe, S., Steinberger, C.: A Model Centered Perspective on Software-Intensive Systems. In: Fellmann, M., Sandkuhl, K. (eds.) Enterprise Modeling and Information Systems Architectures (EMISA'18). CEUR Workshop Proceedings, vol. 2097, pp. 58–64 (2018)

[45] Medvidovic, N., Taylor, R.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering (2000)

[46] Mercadal, J., Enard, Q., Consel, C., Loriant, N.: A domain-specific approach to architecturing error handling in pervasive computing. SIGPLAN Not. **45**(10), 47–61 (Oct 2010). https://doi.org/10.1145/1932682.1869465

[47] Michael, J., Koschmider, A., Mannhardt, F., Baracaldo, N., Rumpe, B.: User-Centered and Privacy-Driven Process Mining System Design for IoT. In: Cappiello, C., Ruiz, M. (eds.) Proc. of CAiSE Forum 2019: Information Systems Engineering in Responsible Information Systems. pp. 194–206. Springer (2019)

[48] Michael, J., Netz, L., Rumpe, B., Varga, S.: Towards privacy-preserving iot systems using model driven engineering. In: Ferry, N., Cicchetti, A., Ciccozzi, F., Solberg, A., Wimmer, M., Wortmann, A. (eds.) MDE4IoT & ModComp 2019, Model-Driven Engineering for the Internet of Things (MDE4IoT) & Interplay of Model-Driven and Component-Based Software Engineering (ModComp). pp. 15–22. CEUR-WS.org (Sep 2019)

[49] Michael, J., Rumpe, B., Varga, S.: Human behavior, goals and model-driven software engineering for assistive systems. In: Koschmider, A., Michael, J., Thalheim, B. (eds.) Enterprise Modeling and Information Systems Architectures (EMSIA 2020). vol. 2628, pp. 11–18. CEUR Workshop Proceedings (June 2020)

[50] Michael, J., Steinberger, C.: Context modeling for active assistance. In: Cabanillas, C., España, S., Farshidi, S. (eds.) Proc. of the ER Forum 2017 and the ER 2017 Demo Track co-located with the 36th Int. Conference on Conceptual Modelling (ER 2017). pp. 221–234 (2017)

[51] Morin, B., Harrand, N., Fleurey, F.: Model-Based Software Engineering to Tame the IoT Jungle. IEEE Software **34**(1), 30–36 (January 2017)

[52] Naik, N.: Choice of effective messaging protocols for iot systems: Mqtt, coap, amqp and http. In: IEEE Int. Systems Engineering Symposium (ISSE). pp. 1–7. IEEE (2017)

[53] Perseil, I.: Alf formal. Innovations in Systems and Software Engineering **7**(4), 325–326 (2011). https://doi.org/10.1007/s11334-011-0168-x

[54] Persson, P., Angelsmark, O.: Calvin – Merging Cloud and IoT. Procedia Computer Science **52**, 210 – 217 (2015), 6th Int. Conf. on Ambient Systems, Networks and Technologies (ANT)

[55] Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., Ng, A.: ROS: an open-source Robot Operating System. In: ICRA Workshop on Open Source Software (2009)

[56] Rasheed, A., San, O., Kvamsdal, T.: Digital twin: Values, challenges and enablers from a modeling perspective. IEEE Access **8**, 21980–22012 (2020)

[57] Ringert, J.O., Roth, A., Rumpe, B., Wortmann, A.: Language and Code Generator Composition for Model-Driven Engineering of Robotics Component & Connector Systems. Journal of Software Engineering for Robotics (JOSER) **6**(1), 33–57 (2015)

[58] Ringert, J.O., Rumpe, B., Schulze, C., Wortmann, A.: Teaching Agile Model-Driven Engineering for Cyber-Physical Systems. In: Int. Conf. on Software Engineering: Software Engineering and Education Track (ICSE'17). pp. 127–136. IEEE (May 2017)

[59] Rumpe, B.: Modeling with UML: Language, Concepts, Methods. Springer International (July 2016)

[60] Schleich, B., Anwer, N., Mathieu, L., Wartzack, S.: Shaping the digital twin for design and production engineering. CIRP Annals **66**(1), 141–144 (2017)

[61] Schuh, G., Häfner, C., Hopmann, C., Rumpe, B., Brockmann, M., Wortmann, A., Maibaum, J., Dalibor, M., Bibow, P., Sapel, P., et al.: Effizientere produktion mit digitalen schatten. ZWF Zeitschrift für wirtschaftlichen Fabrikbetrieb **115**(special), 105–107 (2020)

[62] Sebastián, G., Gallud, J., Tesoriero, R.: Code generation using model driven architecture: A systematic mapping study. Journal of Computer Languages **56**, 100935 (2020). https://doi.org/https://doi.org/10.1016/j.cola.2019.100935

[63] Stachowiak, H.: Allgemeine Modelltheorie (1973)

[64] Stahl, T., Völter, M.: Model-Driven Software Development: Technology, Engineering, Management. Wiley (2006)

[65] Steinberger, C., Michael, J.: Using Semantic Markup to Boost Context Awareness for Assistive Systems, pp. 227–246. Springer International Publishing (2020)

[66] Talkhestani, B.A., Jazdi, N., Schlögl, W., Weyrich, M.: A concept in synchronization of virtual production system with real factory based on anchor-point method. Procedia CIRP **67**, 13–17 (2018)

[67] Tao, F., Zhang, H., Liu, A., Nee, A.Y.C.: Digital twin in industry: State-of-the-art. IEEE Transactions on Industrial Informatics **15**(4), 2405–2415 (April 2019)

[68] Tao, F., Cheng, J., Qi, Q., Zhang, M., Zhang, H., Sui, F.: Digital twin-driven product design, manufacturing and service with big data. The International Journal of Advanced Manufacturing Technology **94**(9-12), 3563–3576 (2018)

[69] Tao, F., Qi, Q., Wang, L., Nee, A.: Digital twins and cyber–physical systems toward smart manufacturing and industry 4.0: Correlation and comparison. Engineering **5**(4), 653 – 661 (2019)

[70] Taylor, N., Human, C., Kruger, K., Bekker, A., Basson, A.: Comparison of digital twin development in manufacturing and maritime domains. In: Borangiu, T., Trentesaux, D., Leitão, P., Giret Boggino, A., Botti, V. (eds.) Service Oriented, Holonic and Multi-agent Manufacturing Systems for Industry of the Future. pp. 158–170. Springer International Publishing, Cham (2020)

[71] Wortmann, A., Combemale, B., Barais, O.: A Systematic Mapping Study on Modeling for Industry 4.0. In: Conf. on Model Driven Engineering Languages and Systems (MODELS'17). pp. 281–291. IEEE (September 2017)

[72] Zanin, M., Perez, D., Kolovos, D.S., Paige, R.F., Chatterjee, K., Horst, A., Rumpe, B.: On Demand Data Analysis and Filtering for Inaccurate Flight Trajectories. In: Proc. of the SESAR Innovation Days. EUROCONTROL (2011)