IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 13 May 2024, accepted 24 May 2024, date of publication 28 May 2024, date of current version 4 June 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3406510

RESEARCH ARTICLE

From Digital Twins to Digital Twin Prototypes: Concepts, Formalization, and Applications

ALEXANDER BARBIE AND WILHELM HASSELBRING
Software Engineering Group, Kiel University, 24118 Kiel, Germany
Corresponding author: Alexander Barbie (alexanderbarbie@gmx.de)

This work was supported by Deutsche Forschungsgemeinschaft (DFG) through Open Access Publikationskosten.

ABSTRACT The transformation to Industry 4.0 also transforms the processes of developing intelligent manufacturing production systems. Digital twins may be employed to advance the development of these new (embedded) software systems. However, there is no consensual definition of what a digital twin is. In this paper, we provide an overview of the current state of the digital twin concept and formalize the digital twin concept using the Object-Z notation. This formalization includes the concepts of physical twins, digital models, digital templates, digital threads, digital shadows, digital twins, and digital twin prototypes. The relationships between all these concepts are visualized as class diagrams using the Unified Modeling Language. Our digital twin prototype approach supports engineers in the development and automated testing of complex embedded software systems. This approach enables engineers to test embedded software systems in a virtual context without the need of a connection to a physical object. In continuous integration/continuous deployment pipelines, such digital twin prototypes can be used for automated integration testing and, thus, allow for an agile verification and validation process. In this paper, we demonstrate and report on the application and implementation of a digital twin using the example of two real-world field studies (ocean observation systems and smart farming). For independent replication and extension of our approach by other researchers, we provide a laboratory study published open source on GitHub.

INDEX TERMS Cyber-physical systems, embedded software systems, digital twin prototypes, automated testing, continuous integration.

LIST OF ACRONYMS

ARCHES	Autonomous Robotic Networks to Help Modern Societies.
CAD	Computer-aided Design.
CI/CD	Continuous Integration/Continuous Delivery.
CPS	Cyber-physical System.
DT	Digital Twin.
DTP	Digital Twin Prototype.
HIL	Hardware-in-the-Loop.
IIoT	Industrial Internet of Things.
MAPE-K	Monitor-Analyze-Plan-Execute over a shared Knowledge.
MIL	Model-in-the-Loop.
PIL	Processor-in-the-Loop.
PLC	Programmable Logic Controller.
PT	Physical Twin.
SIL	Software-in-the-Loop.
SME	Small and Medium-sized Enterprise.
UML	Unified Modeling Language.

The associate editor coordinating the review of this manuscript and approving it for publication was Muhammad Khalil Afzal.

I. INTRODUCTION

For cyber-physical systems Cyber-physical Systems (CPSs), the Industrial Internet of Things Industrial Internet of Things (IIoT), and Industry 4.0 applications, embedded software is an increasingly crucial asset. With increasing requirements and hence, increasing complexity, new challenges arise for manufacturers and in particular, for the engineers of these systems. While in large software companies, software development is often performed by distributed teams of engineers [1], this is usually different for small and medium-sized

enterprises Small and Medium-sized Enterprises (SMEs) that develop embedded systems [2]. Especially, in SMEs, embedded software still is often developed by the same engineers who also develop the electronics and/or mechanical parts [3].

A. RELEVANCE OF DIGITAL TWIN TECHNOLOGY

With the demand for context-aware, autonomous, and adaptive robotic systems [4], more advanced software engineering methods have to be adopted by the embedded software community. Consequently, the way these systems are developed has to advance. In future development workflows, the embedded software systems will be the center-piece of IIoT applications. To achieve this, the community has to move from expert-centric tools [4] to modular systems, whereby domain experts are enabled to contribute to parts of the system.

B. RELEVANCE OF SOFTWARE QUALITY IN THIS CONTEXT

A survey among 2,000 decision makers about trends and challenges in software engineering found that *quality* is perceived in the software industry as the single most relevant premise for survival [5]. Achieving high software quality encompasses optimizing key attributes such as functional correctness, reliability, usability, user experience, performance efficiency, security, maintainability, scalability, compatibility, and interoperability [6], yet organizations struggle to achieve software quality along with cost and efficiency [7]. They often face hurdles due to technical debt, which arises when opting for quick and easy software solutions over more thorough, sustainable approaches, leading to increased maintenance/scalability challenges and potential performance issues down the line. Technical debt can significantly impede the ability to achieve and maintain software quality because it requires additional resources to rectify [8]. Therefore, alongside adopting agile methodologies and automating processes, managing and minimizing technical debt is crucial for organizations that aim to enhance their software quality and ensure long-term success.

C. RELEVANCE OF TESTING IN THIS CONTEXT

During the development of embedded (software) systems, at some point, thorough and reliable tests are necessary to verify and validate the whole system [9]. A common method to test the control algorithms of an embedded software system is Hardware-in-the-Loop Hardware-in-the-Loop (HIL) testing. An example for HIL testing at large scale is Airbus, which creates iron birds of their aircraft containing the corresponding electronics, hydraulics, and flight controls [10]. However, many SMEs cannot afford such redundant hardware just for the purpose of testing software. Hence, test *automation* is among the most popular topics for testing embedded software [11]. Still, automatic quality assurance is a challenge in this context, since hardware is in the loop.

Many different simulation tools have been proposed, developed, and sold, with the promise of reducing costs and time needed for verification and validation. Yet, none of these tools is able to combine all aspects of modern machines during all steps of the production life-cycle, due to the complexity of systems and the high amount of data being processed. Thus, multidisciplinary simulation concepts are increasingly important with regard to scalable and highly modular production environments enabled by cyber-physical systems [12]. Alongside HIL testing, manufacturers have implemented different automated testing strategies with In-the-Loop simulations to reduce costs, e.g., Software-in-the-Loop Software-in-the-Loop (SIL), Model-in-the-Loop Model-in-the-Loop (MIL), and Processor-in-the-Loop Processor-in-the-Loop (PIL) simulations [13].

D. CONTRIBUTIONS

One promising technique to enhance the overall software quality of embedded systems, is the digital twin Digital Twin (DT) concept. This paper aims to enhance the overall comprehension of the concept of digital twins. The contributions of this paper are:

- Our discussion begins with an examination of related work in Section II, highlighting the lack of consensus on the definition of digital twins.
- Despite numerous research papers proposing new definitions or analyzing existing ones, formal or semi-formal methodologies remain rare. Therefore, this paper focuses on deconstructing the various components of a digital twin, using Object-Z notation for formal specification rather than mere definition. Specifically, we offer formal specifications for the physical twin, digital model, digital template, digital thread, digital shadow, digital twin, and digital twin prototype in Section III.
- These concepts have been applied in real-world contexts, as detailed in Section IV.
- To facilitate independent replication of our methodologies, we include the PiCar-X example, which is available as open source on GitHub.
- The paper concludes with a summary of our findings and directions for future research in Section V.

II. RELATED WORK

Digital twins are not only a growing topic in academia but also in industry, particularly in manufacturing [14]. However, there is still no consensus on the definition of a digital twin, as explained in Section II-A. Most of the research conducted to find a general definition of a digital twin are literature reviews [15], [16], [17], [18] investigating where digital twins are used, which components are part of it, and which level of integration with the CPS exists. In particular, Kritzinger et al. [18] contributed with their literature review to a consensual understanding about which subsystems are part of a digital twin. They consider the digital model, the digital shadow, and the digital twin as three separate levels of integration in the

overall concept of digital twins. In this paper, we extend this work by providing a formalization for all these categories.

With regard to mathematical approaches to formalize the concept of digital twins, there is a lack in research papers. Nevertheless, we discuss two approaches [19], [20] that use semi-formal approaches to define the relationships between the different components of a digital twin in Section II-B.

A. THE EVOLUTION OF THE DIGITAL TWIN CONCEPT

An innovative method for testing and monitoring embedded systems was used for space missions, dating back to the early Apollo missions conducted by the National Aeronautics and Space Administration (NASA). Here, the ''Twin'' concept was initially employed during the Apollo missions in the late 1960s as a safety precaution. If a system on the spacecraft failed during the mission, engineers had no access to the capsule. A failure to fix problems in a timely manner could be catastrophic for the space mission. At the time, computational power was insufficient for complex simulations. Therefore, NASA engineers came up with the idea of building at least two identical space capsules. One was used for the mission while the other remained on Earth, serving as the ''Twin'' for simulation purposes. Changes to the system were first tested on the Twin before the astronauts received the instructions. This approach required both capsules to be maintained exactly the same, including replacing parts on the Twin even if it was not used during a mission. NASA had planned to transfer this approach to the Space Shuttle program but abandoned the idea because of high costs.

Half a century later, with advancements in computational power and improved simulations, NASA's Twin concept has evolved into a digital twin. However, there was a second research threads that contributed to this concept. The second thread originated in the manufacturing industry and dates back to 2002, when Grieves [21] first pitched for the formation of a product lifecycle management (PLM) center at the University of Michigan. The presentation slide, as depicted in Figure 1, had the title ''Conceptual Ideal for PLM'' [22] and sketched the idea of a digital twin and named it ''Mirrored Spaces Model'' back than [21].

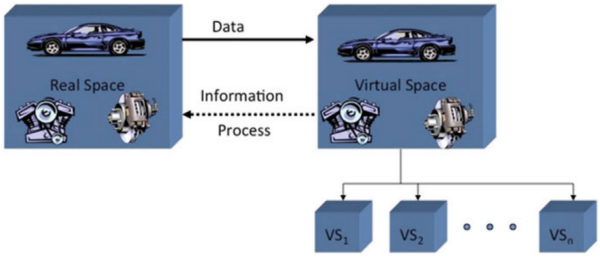

FIGURE 1. A digital twin by Grieves and Vickers [22] consists of the real space (left side), the virtual space (right side), and the link for data flow from real space to virtual space. The opposite direction is done manually by using information to enhance processes (Source: [22]).

Grieves envisioned with the Mirrored Spaces Model already three crucial components of digital twins: the physical space, the virtual space, and the data link between the physical and virtual spaces. Later, in 2016, Grieves and Vickers [22] defined the digital twin as stated in Definition 1:

> *Definition 1 (Digital Twin by Grieves and Vickers [22]):* The Digital Twin is 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. At its optimum, any information that could be obtained from inspecting a physical manufactured product can be obtained from its Digital Twin. Digital Twins are of two types: Digital Twin Prototype and Digital Twin Instance. Digital twin's are operated on in a Digital Twin Environment.

Definition 1 considered the digital twin to be a collection of technologies and distinguished between two types: the Digital Twin Prototype Digital Twin Prototype (DTP) and the Digital Twin Instance. The Digital Twin Prototype is a set of blueprints etc. used to construct or maintain a physical twin Physical Twin (PT). The Digital Twin Instance is the specific instance created after the physical twin has been manufactured and is linked to it throughout its lifecycle. While the vision presented by Grieves and Vickers [22] hinted at possibilities achievable with contemporary technology, the limitations of the technology available in 2002 restricted the implementation of digital twins to a rudimentary level. The computational power was not sufficient for large and complex simulations and the Web 2.0 and hardware virtualization, as used in modern data centers, were in their infancy. Digital twins were seen as a new paradigm for designing, manufacturing, and servicing products. However, the interpretation of a digital twin may varies across different sectors [14]

After their introduction, digital twins experienced a hype phase until around the year 2006. The first hype of digital twins was driven by high hopes in the industry. However, the technology did not live up to the hype, and digital twins became a buzzword in marketing departments rather than a fully realized concept. Newman [23] observed and criticized something similar with regard to microservice architectures. Saracco and Henz [14] emphasized that industry drove the development of digital twins, while academia ignored it. The revival of interest in digital twins in 2016 was due to the maturity of IIoT and CPS technologies, and academia also joined the bandwagon. Digital twins reached the peak of the Gartner Hype Cycle of emerging technologies in 2018 [24]. Furthermore, an increased number of research papers and special issues published by journals can be registered after 2016.

Between 2006 and 2016, different NASA researchers proposed their visions for digital twins [21]. Piascik, Vickers, Lowry, et al. [25] used the term digital twin in their technology roadmap for NASA. However, they described the digital twin concept but did not define digital twins.

The better known digital twin definition was by Glaessgen and Stargel [26] for next-generation fighter aircraft and NASA vehicles, as shown in Definition 2:

> *Definition 2 (Digital Twin by Glaessgen and Stargel [26] (NASA)):* A Digital Twin is an integrated multiphysics, multiscale, probabilistic simulation of an as-built vehicle or system that uses the best available physical models, sensor updates, fleet history, etc., to mirror the life of its corresponding flying twin. The Digital Twin is ultra-realistic and may consider one or more important and interdependent vehicle systems, including airframe, propulsion and energy storage, life support, avionics, thermal protection, etc.

They tailored their vision for the specific use case of spacecraft, satellites, and space exploration, where simulations play a crucial role because of the high cost of hardware and human resources. These simulations are used both in the development phase, which indicates at least a MIL approach, and to monitor the systems during missions. To detect anomalies during flight, they also included a channel for sending sensor data from the physical twins to their corresponding digital twins. Loading these data into the simulation with a realistic model supersedes NASA's Twin approach from the Apollo missions. This is similar to the data link shown in Figure 1, only with far advanced technology and tools. A demonstration of their implementation can be seen in the Perseverance Rover that landed on Mars in 2021 [27].

In parallel to the definition by NASA, Garetti, Rosa, and Terzi [28] defined digital twins for manufacturing, as defined in Definition 3:

> *Definition 3 (Digital Twin by Garetti et al. [28]):* The digital twin consists of a virtual representation of a production system that is able to run on different simulation disciplines that is characterized by the synchronization between the virtual and real system, thanks to sensed data and connected smart devices, mathematical models and real time data elaboration. The topical role within Industry 4.0 manufacturing systems is to exploit these features to forecast and optimize the behaviour of the production system at each life cycle phase in real time.

When the attention on digital twins research was rekindled, academia proposed multiple definitions for the concept [15]. These definitions were influenced by the realistic simulation approach proposed by NASA. Rosen et al. [17] linked the digital twin concept to the Industry 4.0 strategy of the German Platform Industry 4.0 [29]. They illustrated how simulations evolved over time, from mechanics in the 1960s to simulation-based system design and finally to digital twins since 2015. They also highlighted that modularity, autonomy, and connectivity are crucial requirements for digital twins, among other factors.

The definitions provided by Grieves and Vickers [22] and NASA include only an automated connection from the physical twin to its digital twin. Trauer, Schweigert-Recksiek, Engel, et al. [30] conducted an industrial case study to analyze how the industry perceived and defined digital twins between 2002 and 2019. They traced the evolution of digital twins and presented Definition 4 as a result:

> *Definition 4 (Digital Twin by Trauer et al. [30] (2020)):* A Digital Twin is a virtual dynamic representation of a physical system, which is connected to it over the entire life cycle for bidirectional data exchange.

We present Definition 4 here because of the inclusion of the *bidirectional* data exchange from the digital twin to the physical twin. This *bidirectional* interaction allows remote control and operation of the physical twin as well as new opportunities for collaboration between physical twin and digital twin. This poses a challenge for engineers to either develop the software independently for each twin, violating the principle of realistic replication, or to use tools such as Docker to containerize the physical twin's software for use as a digital twin.

Depending on the research field, the industry, and use cases, the term digital twin is often used synonymous with concepts such as *Digital Model*, *Digital Shadow*, and *Digital Thread* [15], [18]. Kritzinger et al. [18] conducted a categorical literature review and analyzed research papers with regard of the proposed concept and how it deviates from a common understanding of the essential parts of digital twins. They classify three subcategories of a digital twin by their level of integration with the physical twin: (i) digital model, (ii) digital shadow, and (iii) digital twin. The differences are depicted in Figure 2.

(i) Figure 2a shows the *digital model*. There is no automated connection between the physical object and the digital model. No automated data exchange is realized. State changes in the physical object do not immediately affect the digital model and vice versa.

(ii) If there is an automated one-way data flow from the physical object to the digital object (see Figure 2b), then this is a *digital shadow*. A change in the state of the physical object leads to a change in the state of the digital shadow, but not vice versa.

(iii) Figure 2c shows a fully integrated *digital twin*. The data flows between the physical twin and the digital twin in both directions were automated. In such a configuration, the digital twin might also act as a control instance of the physical twin. A change in the state of the physical twin directly leads to a change in the state of the digital twin and vice versa.

With the increasing importance of digital twins, the International Organization for Standardization (ISO) also

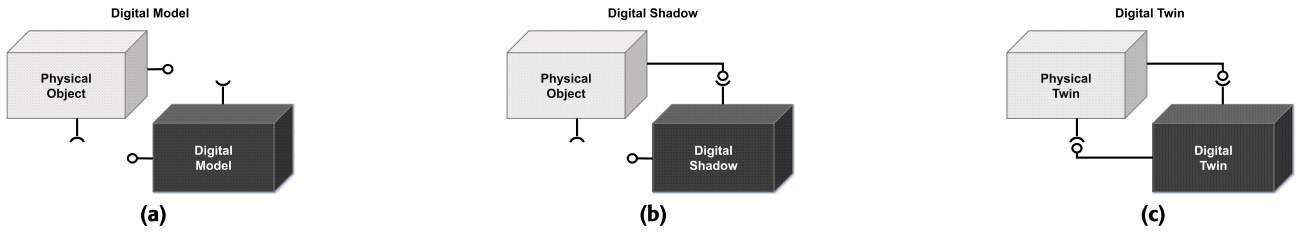

FIGURE 2. Subcategories of digital twins by their level of integration with the physical twins (Source: [18]).

published the ISO 23247 series, defining a framework to support the creation of digital twins of observable manufacturing elements, including personnel, equipment, materials, manufacturing processes, facilities, environment, products, and supporting documents [31]:

> *Definition 5 (Digital Twin by International Organization for Standardization [31] (2021)):* A digital twin assists with detecting anomalies in manufacturing processes to achieve functional objectives such as real-time control, predictive maintenance, in-process adaptation, Big Data analytics, and machine learning. A digital twin monitors its observable manufacturing element by constantly updating relevant operational and environmental data. The visibility into process and execution enabled by a digital twin enhances manufacturing operation and business cooperation.

One aspect of ISO 23247 that immediately catches the eye is the notable omission of bidirectional communication. This standard primarily emphasizes the monitoring aspect of a digital twin. According to the definition provided by Kritzinger et al. [18], ISO 23247 solely encompasses what they term as a *digital shadow*.

Since 2018, IIoT platforms transitioned from basic data hubs to digital twin platforms. Lehner et al. [33] evaluated the digital twin platforms provided by Amazon Web Services, Microsoft Azure, and the Eclipse ecosystem and showed that they fulfill many requirements, yet not all key requirements. Features like bidirectional synchronization between physical and digital twins require additional coding, and automation protocols are not covered yet. According to the categorization of the integration level of digital twins [18], these platforms only help to establish a so-called *digital shadow* [18]. Modern simulation tools, such as AutoDesk, aPriori, or Ansys, use IIoT platforms to feed the simulation with data and enable the integration of automation protocols. Often, they are promoted with the promise of a digital twin. However, similar to cloud providers, these tools also just help to establish a digital shadow. The simulation of a physical twin still does not cover the entire embedded software system that runs on the physical twin and also lacks the ability of proper bidirectional synchronization between physical twin and digital twin. Table 1 classifies the presented digital twin definitions into their primary use cases.

B. CONCEPTUAL MODELS TO DEFINE DIGITAL TWINS

The presented research projects and papers leave plenty of space for interpretation of the digital twin concept. This is one reason, why there are so many definitions of digital twins.

Yue [19] present a semi-formal approach using Unified Modeling Language (UML) class diagrams to define the physical twin, the digital twin, and their relationships by the example of an automated warehouse system. Figure 3 depicts these relationships. The physical twin and digital twin exchange data via the *PT-To-DT-Connection* and *DT-To-PT-Connection*. A state change in one twin, triggers the change of the state of its counterpart. Furthermore, they payed attention to two aspects, which are often not considered explicitly: fidelity and the twinning rate. Fidelity considers the accuracy and the level of abstraction of the digital twin and the twinning rate is the interval physical twin and digital twin synchronize their states. However, the semi-formal approach by Yue, Arcaini, and Ali [19] has its flaws. Although they considered the digital model as part of the digital twin, it is not explicitly mentioned in the general overview in Figure 3. Moreover, the digital shadow was ignored completely.

Becker et al. [20] present in their conceptual model of digital shadows for cyber-physical systems in a similar approach using also UML class diagrams to show the relationships, but solely for the digital shadow. The focus of the digital shadow is on single assets and their information flow from the physical twin to the digital shadow. They also emphasize that an asset's corresponding model is part of the digital shadow and that models can be of different natures/types.

A formal mathematical approach, yet very abstract, of the relationships between physical twins, digital shadow, and digital twin was presented by Lv [34]. A limitation of their approach is that it still offers a lot of space for interpretation and the mathematical notation is peculiar.

In this paper, we merge and extend the relationship diagrams of Yue et al. [19] and Becker et al. [20] by also including the digital model and digital shadow to give a full overview of the digital twin concept. In addition, we present the formalization of a digital twin software architecture using the Object-Z notation.

C. CONTINUOUS TWINNING

In the development phase of cyber-physical systems, HIL testing is still the common approach. The pressure to

TABLE 1. Classification of the presented digital twin definitions into their primary use case.

Primary Use Case	Definitions from Related Work
A snapshot of a mathematical model or computer-aided design used primarily for initial simulations.	[26], [18]
Monitors the corresponding physical twin in (near) real-time and internally updates an instance of the digital model.	[17], [28], [31], [18]
Operates and monitors the corresponding physical twin in (near) real-time and internally updates an instance of the digital model.	[22], [30], [32], [18]
Bundles all information, the digital model, and the embedded software system required to build the physical twin.	[21] (Digital Twin Prototype Definition)

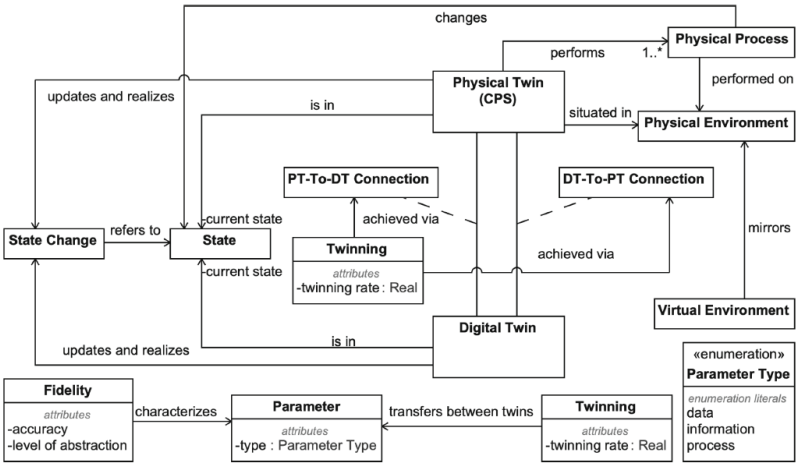

FIGURE 3. Semi-formal description of the relationships between physical twin, digital twin, their connections, and environments as described by Yue, Arcaini, and Ali [19].

reduce costs [7] led to many different approaches to switch from HIL to SIL. To date, for most industrial applications, sensors and actuators are connected via input/output ports to programmable logic controllers Programmable Logic Controllers (PLCs). Although new wireless communication technologies and more powerful and efficient single-board computers open up the embedded community for cheaper and faster development processes, the predominance of PLCs will hold for years. It is quite common to use PLCs in a HIL setup, where the PLC is connected to a simulation [35]. Engineers can program the PLC, and the simulation delivers a virtual context with simulated sensors/actuators to the PLC. As only one engineer can work on an HIL system at the same time, SIL approaches become more and more popular to enable collaboration between engineers. Yu et al. [35] demonstrated that a software PLC in a SIL context can be realized with Docker and other tools.

Quality assurance of embedded systems is regulated with standards and norms to ensure robust testing and to prevent malfunctions that might pose a risk to the safety of individuals who work with or use these systems [2]. The aviation industry is renowned for its strict and stringent testing procedures, contributing to the fact that aircraft are the safest mode of transportation, statistically. This was not the case half a century ago, as standards and procedures have evolved through various experimentation with different testing strategies.

The digital twin prototype approach presented in this paper, enables engineers to produce the first minimum viable product with the first implemented device driver and emulator. Thanks to the proposed modular architecture, all additional nodes and emulators can be developed and added iteratively. Putting all modules in a source code management system allows all developers to use the digital twin prototype and enhance the entire system incrementally, without the need to connect to the hardware of the digital twin. As a bonus, this also enables automated SIL testing in continuous integration/continuous delivery Continuous Integration/Continuous Delivery (CI/CD) pipelines.

By following CI/CD workflows, the development of embedded software systems becomes an agile and incremental process. Beginning with a prototype of a device driver for a single piece of hardware, to entire production plants, to smart factories, agile software development is enabled. This not only improves the software quality and shortens the release cycles, it also allows additional stakeholders to participate in a feedback loop in the development process from the first minimum viable product. Adjusting software requirements

or fixing design flaws can be done during development. With this method, digital twins evolve continuously in small incremental steps, rather than in major releases. Nakagawa et al. [36] envision and call this approach *Continuous Twinning*.

III. THE DIGITAL TWIN CONCEPT - A FORMALIZATION

As Grieves [37] elaborates, there is a flaw in the categorization of the digital twin definition by Kritzinger et al. [18]. Their framework, which divides digital twins into three subcategories, paradoxically includes a digital twin as one of its own subcategories, resulting in the potential for endless recursion. Despite Kritzinger et al. [18] effectively illustrating the distinctions among these proposed subcategories, as seen in Figure 2, their definition still permits the categorization of a digital model or digital shadow as a digital twin. This ambiguity complicates the understanding of what constitutes digital twins in contrast to digital models or digital shadows. Consequently, there is a risk that customers might expect to purchase a digital twin, as advertised by vendors, only to receive a digital model instead. However, we do not share the recommendation to ignore the difference between a digital shadow and a digital twin with Grieves [37]. To enhance the clarity around the concepts and relationships between physical twins, digital models, digital shadows, digital threads, digital twin prototypes, digital templates, and digital twins, we formally specify the digital twin concept as follows. Similar to Hasselbring [38], we propose a three-level interleaving of formality in the specification:

1) informal prose explanation and illustrations with examples;
2) semi-formal object-oriented modeling with the UML;
3) rigorous formal specification with Object-Z.

Object-Z [39] is a formal specification notation used to describe the behavior of software systems. It extends the Z notation [40] and enables the incorporation of object-oriented concepts, such as classes, objects, inheritance, and polymorphism, into specifications. Additionally, Object-Z allows for the specification of operations that can be performed on objects, along with constraints on attribute values and relationships between objects, all expressed in a mathematical notation. The following specification has been checked using a type checker provided by the Community Z Tools Project [41].

The formal specification is exemplified through an embedded software system comprising a sensor, an actuator that also serves as a data transmitter, and an embedded control system connected to both. This control system manages data and command exchange between these components. All example components are very basic and are only meant to demonstrate the core ideas. A real system would be more complex, including more third-party dependencies, tools, and frameworks.

A. THE PHYSICAL TWIN

The digital twin concept starts with the physical twin:

> *Definition 6 (Physical Twin):* A physical twin is a real-world physical system-of-systems or product. It comprises sensing or actuation capabilities driven by embedded software.

Figure 4 illustrates the deployment diagram of the proposed simple embedded system. In this example, the sensor is connected to the controller via an RS232 interface, and the transmitter is connected via Ethernet. All data collected from the sensor is processed by the controller logic and subsequently sent to an external source via the transmitter. Commands to modify the sensor's behavior are received by the transmitter and forwarded to the sensor through the control logic.

Consider both devices as black boxes that maintain a list of accepted commands, a method for executing tasks based on the commands and returning a result, and functions for sending and receiving data. Additionally, a device driver holds a corresponding list of commands that can be sent to the devices. The lists on the device and the device driver are identical, and the device driver handles command transmission and response reception.

The UML class diagram in Figure 5 depicts the various classes that form the embedded control system. To align with the clean code principles, abstract classes *Device* and *DeviceDriver* are introduced first. Sensors and actuators are considered as devices and thus inherit from *Device*, as depicted on the left side of Figure 5. All devices are connected to the embedded control system.

The crucial elements of embedded software systems are the connections between the control systems and the sensors/actuators. In this example, the connections are established using different *PROTOCOL* types (TCP or RS232) to facilitate communication between *Device* and *DeviceDriver*. Specifically, *SensorDriver* inherits from *DeviceDriver* and employs a *RS232Connection* to establish a connection with a *Sensor*. Similarly, *Transmitter* and *TransmitterDriver* (which also inherits from *DeviceDriver*) establish a connection using *TCPConnection*. While a *Device* is treated as an external component running on the device, a corresponding *DeviceDriver* is an integral part of the embedded control system. A *Device* consists of two main components: a *Connection* object and a set of accepted commands (*commandList*). The *Connection* object manages the data exchange between a *Device* and a *DeviceDriver*. The *ExecuteCommand* function represents the execution of a task after a command has been sent to the *Device*. It expects a *COMMAND* object sent by the *DeviceDriver* and returns a *RESPONSE* object. The *Send* and *Receive* functions utilize the corresponding functions provided by the contained *Connection*. To facilitate the exchange of data from a sensor to another process, such as the control logic, *EventHandler* objects are introduced. It can be assumed that these *EventHandler* objects are implemented in a manner similar to the Observer pattern, which also encompasses publish/subscribe architectures.

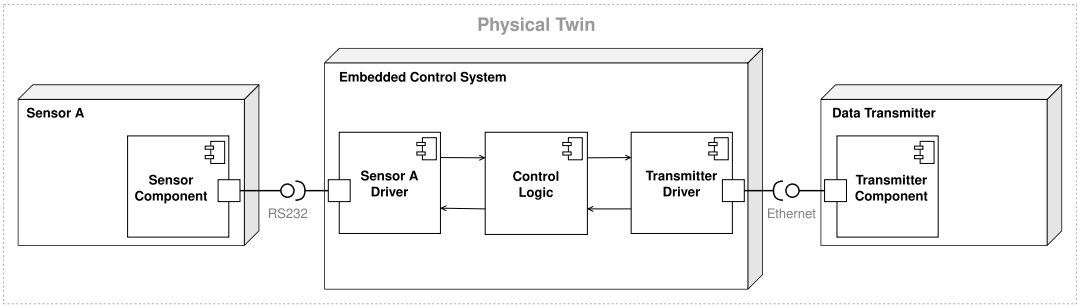

FIGURE 4. The deployment diagram of an embedded system comprising a sensor, a data transmitter and the embedded control system both are connected to. The sensor is connected via RS232 and the transmitter via Ethernet.

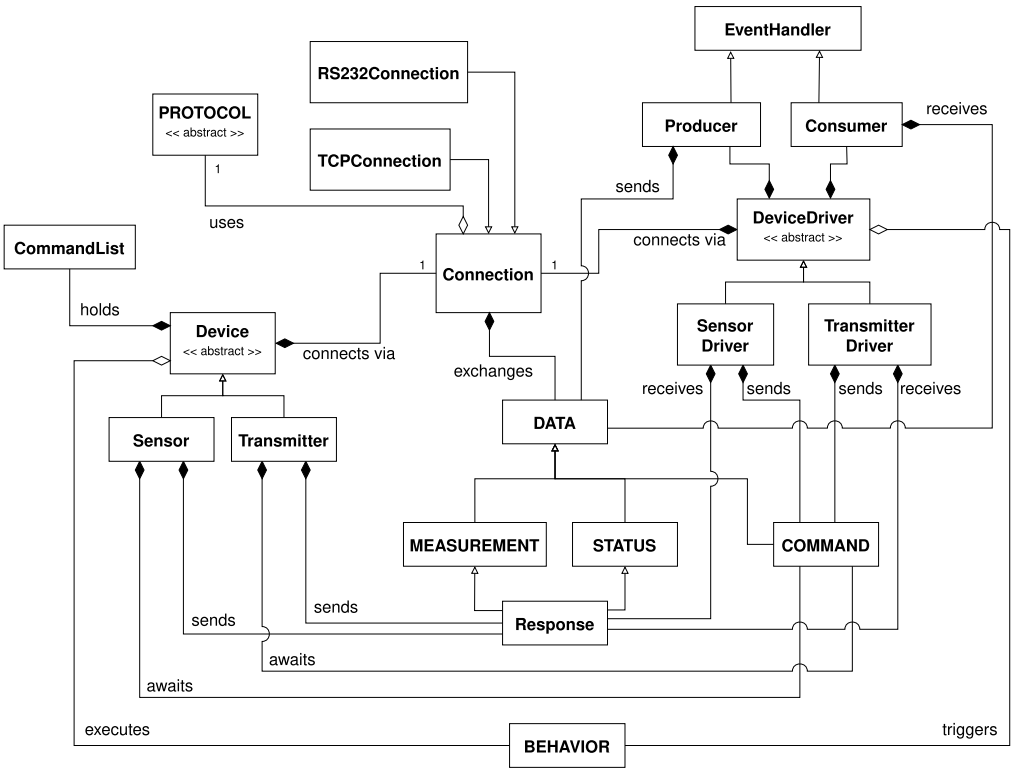

FIGURE 5. UML class diagram of a physical twin.

In this setup, all events received from the *Sensor* are emitted to all listeners through a *Producer*, and processes receive these events by including a *Consumer*.

1) OBJECT-Z FORMALIZATION

The specification of this simple embedded system follows a bottom-up approach. The deployment diagram, as depicted in Figure 4, can be defined using the Object-Z notation. To achieve this, some basic type definitions are introduced:

$$[PROTOCOL, EVENT]$$

PROTOCOL represents the communication protocols utilized between the devices and the control system, whereas *EVENT* is the type employed for data exchange between processes. Basic type definitions introduce new types in Z and Object-Z. Such internal structure is considered irrelevant for the specification. In this particular specification, any details that are not architecturally relevant are abstracted in this way. The various *PROTOCOL* types used in the schema architecture are subsequently defined through an axiomatic definition. In this context, *TCP* and *RS232* are established as values of type *PROTOCOL*:

$$TCP, RS232 : PROTOCOL$$

Up until this point, only basic types have been introduced. However, because Object-Z is object-oriented, objects are also created. In this context, the parent class is denoted as *DATA*, which will later be specialized through inheritance into classes specific to the various data types:

DATA

$$data : \text{seq}\{0, 1\}$$

Communication between devices is represented by a sequence of bits. Given that standard data types such as integers, floats, or strings are irrelevant for the specification, only a bit representation is utilized. As both a device and its corresponding device driver exchange either *RESPONSE* or *COMMAND*, the corresponding schemes inherit from the *DATA* class. In this context, *RESPONSE* can represent either *MEASUREMENT* or *STATUS*:

COMMAND
DATA

STATUS
DATA

MEASUREMENT
DATA

RESPONSE
MEASUREMENT
STATUS

Once the data types have been formalized, the various components and their connections can be configured. Initially, the abstract *Connection* class can be defined as follows:

Connection

$$\lceil (\textit{INIT}, \textit{Read}, \textit{Write}) \rceil$$

$$\begin{aligned} &\textit{type} : \textit{PROTOCOL} \\ &\textit{dataStream} : \text{seq}\, \textit{DATA} \end{aligned}$$

INIT

$$\textit{dataStream} = \langle\rangle$$

Write

$$\begin{aligned} &\Delta(\textit{dataStream}) \\ &\textit{value}? : \textit{DATA} \\ \hline &\textit{dataStream}' = \textit{dataStream} \frown \langle \textit{value}? \rangle \end{aligned}$$

Read

$$\begin{aligned} &\Delta(\textit{dataStream}) \\ &\textit{value}! : \textit{DATA} \\ \hline &\textit{dataStream} = \langle \textit{value}! \rangle \frown \textit{dataStream}' \end{aligned}$$

The symbol ? denotes input parameters and ! denotes outputs [39].

A *Connection* possesses a type and manages bit sequences, represented as a stream (*dataStream*). The *Write* function appends bit sequences to the stream, whereas the *Read* function extracts them by reading bits from it. The specific implementations, *RS232Connection* and *TCPConnection*, are named after the types they set for the *Connection* object from which they inherit:

RS232Connection
Connection

$$\textit{type} = \textit{RS232}$$

TCPConnection
Connection

$$\textit{type} = \textit{TCP}$$

A *Device* comprises a *Connection* object and a set of accepted commands (*commandList*). The *Connection* object is responsible for managing the data exchange between a *Device* and a *DeviceDriver*. The *ExecuteCommand* function represents the execution of a task following the transmission of a command to the *Device*. It expects a *COMMAND* object sent by the *DeviceDriver* and returns a *RESPONSE* object. The *Read* and *Write* functions make use of the corresponding functions provided by the contained *Connection*:

Device

$$\lceil (\textit{INIT}, \textit{Send}, \textit{Receive}, \textit{commandList}) \rceil$$

$$\begin{aligned} &\textit{connection} : \downarrow \textit{Connection} \textcircled{c} \\ &\textit{commandList} : \mathbb{P}\, \textit{COMMAND} \\ \hline &\textit{connection} \notin \textit{Connection} \\ &\#\textit{commandList} > 0 \end{aligned}$$

The symbol $\downarrow$ denotes the union of Connection with all sub-types. Connection is abstract. Thus, the Connection has to be a sub-type that implements it.
The symbol $\textcircled{c}$ denotes object containment [39].

INIT

$$\textit{connection}.\textit{INIT}$$

ExecuteCommand

$$\begin{aligned} &\textit{command}? : \textit{COMMAND} \\ &\textit{result}! : \downarrow \textit{DATA} \\ \hline &\textit{command}? \in \textit{commandList} \end{aligned}$$

$$\textit{Send} \mathrel{\widehat=} \textit{connection}.\textit{Write}$$
$$\textit{Receive} \mathrel{\widehat=} \textit{connection}.\textit{Read} \fatsemi \textit{ExecuteCommand} \fatsemi \textit{Send}$$

The symbol $\fatsemi$ denotes a sequential composition.

Similar to the *Device* class, the *DeviceDriver* class also contains a *Connection* object, a set of commands, a set of known behaviors, and a function that maps a behavior to the corresponding command that can be sent to the *Device*:

DeviceDriver

$\lceil(\textit{INIT}, \textit{Send}, \textit{Receive}, \textit{commandList}, \textit{emitter}, \textit{consumer})$

$\textit{connection} : \downarrow \textit{Connection}©$
$\textit{commandList} : \mathbb{P}\, \textit{COMMAND}$
$\textit{emitter} : \textit{Producer}©$
$\textit{consumer} : \textit{Consumer}©$

$\textit{connection} \notin \textit{Connection}$

INIT

$\textit{connection}.\textit{INIT} \land \textit{consumer}.\textit{INIT}$

$\textit{Send} \mathrel{\widehat=} \textit{consumer}.\textit{Consume} \fatsemi \textit{connection}.\textit{Write}$
$\textit{Receive} \mathrel{\widehat=} \textit{connection}.\textit{Read} \fatsemi \textit{emitter}.\textit{Emit}$

Assume that for this example the *DeviceDriver* fully implements all interactions with the *Device* and hence, the *commandList* for both instances is equal. The *Receive* and *Send* functions in this class also utilize the *Connection*'s *Read* and *Write* functions. Any further implementations beyond this scope are not relevant to our specification.

Data exchange between different processes, such as the *DeviceDriver* and the *ControlLogic*, occurs through *EventHandler*s:

EventHandler

$\lceil(\textit{event})$

$\textit{event} : \textit{EVENT}$

Each *EventHandler* registers for a specific *EVENT*, which can represent, for example, a simple response from the *Device*. In this example, the *EventHandler* is an abstract class, and *Producer* and *Consumer* are the specific implementations. Assuming both register for the same *EVENT*, like "NEWDATA," a *Producer* can emit new events, and the *Consumer* receives and handles all incoming events. It is important to note that this relationship is not one-to-one but rather one-to-many, allowing an indefinite number of *Consumer*s to listen to the same *Producer*. The main function of a *Producer* is the *Emit* function, which is called with a passed *DATA* object, and then all *Consumer*s are notified:

Producer

$\lceil(\textit{INIT}, \textit{event}, \textit{Emit})$
EventHandler

Emit

$\textit{occuredEvent}? : \downarrow \textit{DATA}$
$\textit{eventToEmit}! : \downarrow \textit{DATA}$

$\textit{eventToEmit}! = \textit{occuredEvent}?$

A *Consumer* registers via the *Observe* function to an *EVENT*, listens only to the emitted events, and handles them in a queue. The *Consume* function always returns the first element in the queue:

Consumer

$\lceil(\textit{INIT}, \textit{event}, \textit{queue}, \textit{Observe}, \textit{Consume})$
EventHandler

$\textit{queue} : \mathbb{P} \downarrow \textit{DATA}$

INIT

$\textit{queue} = \varnothing$

Observe

$\Delta(\textit{queue})$
$\textit{item}? : \downarrow \textit{DATA}$

$\textit{queue}' = \textit{queue} \cup \{\textit{item}?\}$

Consume

$\Delta(\textit{queue})$
$\textit{item}! : \downarrow \textit{DATA}$

$\#\textit{queue} > 0$
$\textit{item}! \in \textit{queue}$
$\textit{queue}' = \textit{queue} \setminus \{\textit{item}!\}$

After introducing the basic classes, the logic of the embedded control system can be defined. The *DeviceDriver* manages all communication between the control system and the *Device*, with communication being established through the *Connection* class. In this scenario, assume this *DeviceDriver* is straightforward and serves as a relay between the control logic and the device.

The *Consumer* handles all incoming *DATA* from the control logic and forwards it to the device. When responses are received from the device, the *emitter* forwards these responses to all listeners. In Object-Z, the symbol "$\fatsemi$" signifies sequential execution. Consequently, in the *Send* function, the process begins by receiving an incoming event by invoking *consumer.Consume*. Only after this call's result is obtained, it is passed to the *Connection*, which then sends the command to the device. Conversely, incoming responses from the device are acquired from the connection using *connection.Read* and subsequently emitted to all listeners through *emitter.Emit*.

Now that the abstract classes for *Device*, *Connection*, and *DeviceDriver* have been established, we can proceed to define the concrete classes for the sensor, named *Sensor*, and its corresponding device driver, *SensorDriver*, as depicted in Figure 5. In this particular example, *Sensor* and *SensorDriver* are interconnected using a *RS232Connection*. The outcome of an executed command is categorized as a *RESPONSE*, which can represent either a *MEASUREMENT* or a *STATUS* object.

The remaining functions within these specific classes remain consistent with those in the abstract parent classes *Device* and *DeviceDriver*:

Sensor
$\lceil(\textit{INIT}, \textit{Send}, \textit{Receive}, \textit{commandList})$
Device

$\textit{connection} : \textit{RS232Connection}$©

ExecuteCommand
$\textit{command}? : \textit{COMMAND}$
$\textit{result}! : \textit{RESPONSE}$

$\textit{command}? \in \textit{commandList}$

A *SensorDriver* inherits the *EventHandler*s from its parent class:

SensorDriver
$\lceil(\textit{INIT}, \textit{Send}, \textit{Receive}, \textit{commandList})$
DeviceDriver

$\textit{connection} : \textit{RS232Connection}$©

In this example, all incoming commands are dispatched by the control logic, consumed by the driver, and subsequently forwarded to the sensor via the connection. Vice versa, all responses from the sensor are emitted as events by the corresponding producer and can be listened to by all consumers. The essence of this specification lies in the communication between a device and its device driver, which is captured by the *Communication* schema. In this instance, the device is a *Sensor*, and the driver is a *SensorDriver*. Both the device and the device driver share the same *commandsList* and are connected through an *RS232Connection*. In Object-Z, the symbol "$\|$" denotes the execution of functions in parallel [39]. As such, the *ReadFromDevice* operation exemplifies the *Sensor* transmitting data while the corresponding *SensorDriver* reads it. Conversely, *ReadFromDriver* represents the reverse scenario, with communication from the *SensorDriver* to the *Sensor*:

Communication
$\textit{device} : \textit{Sensor}$
$\textit{driver} : \textit{SensorDriver}$

$\forall x : \textit{device.commandList}$
$\quad \bullet\ x \in \textit{driver.commandList}$
$\forall x : \textit{driver.commandList}$
$\quad \bullet\ x \in \textit{device.commandList}$

$\textit{ReadFromDevice} \mathrel{\widehat{=}} \textit{device.Send} \parallel \textit{driver.Receive}$
$\textit{ReadFromDriver} \mathrel{\widehat{=}} \textit{driver.Send} \parallel \textit{device.Receive}$

The *Transmitter* class is akin to the *Sensor* class in several ways. It handles incoming commands and provides responses in return. However, since the *Transmitter* is an actuator, it does not return measurements but instead sends data using another communication protocol, such as LoRaWAN. It is important to note that this communication differs from the *Communication* schema described earlier. Additionally, the *Connection* object solely represents the connection between the *Device* and *DeviceDriver* and does not pertain to the communication between two transmitters:

Transmitter
$\lceil(\textit{INIT}, \textit{Send}, \textit{Receive}, \textit{commandList})$
Device

$\textit{connection} : \textit{TCPConnection}$©

ExecuteCommand
$\textit{command}? : \textit{COMMAND}$
$\textit{result}! : \textit{RESPONSE}$

$\textit{command}? \in \textit{commandList}$

Similar to the *SensorDriver*, the *TransmitterDriver* represents only a data relay between device and control logic:

TransmitterDriver
$\lceil(\textit{INIT}, \textit{Send}, \textit{Receive}, \textit{commandList})$
DeviceDriver

$\textit{connection} : \textit{TCPConnection}$©

The details of the control system are not within the scope of this specification. The control logic for an embedded system is often some form of a state machine. State machines fully automate a system but do not adapt to new or changed processes on the fly. Modern Industry 4.0 applications, incorporate autonomous behavior, extracted or learned from gathered data and thus, include architectures different from state machines. Furthermore, the orchestration of processes, including different commands to different sensor and actuators, can be quite complex. However, for this example, the only function of the *ControlLogic* class is to execute the commands received from the transmitter and return the responses from the sensor:

The incoming commands contain the value that sets the sampling rate of the sensor. To configure the period, the function *sendCmd* processes events sequentially from the transmitter queue. For each event, the *SetPeriod* function is called to set the sampling rate. The newly configured period is then sent as a command to the sensor, which adjusts its sampling rate accordingly. This message exchange is logged into a list called *dataLog*. All events originating from the sensor are handled by *sendRsp* and are sent to the transmitter

ControlLogic

$sensor : Consumer\textcircled{C}$
$transmitter : Consumer\textcircled{C}$
$response : Producer\textcircled{C}$
$command : Producer\textcircled{C}$
$period : \mathbb{Z}$
$dataLog : \mathbb{P}\downarrow DATA$

INIT

$period = 0$
$dataLog = \varnothing$
$sensor.INIT \land transmitter.INIT$

SetPeriod

$\Delta(period)$
$newPeriod? : \mathbb{Z}$

$period' = newPeriod?$

LogData

$\Delta(dataLog)$
$newSensorData? : \downarrow DATA$

$period > 0$
$dataLog' = dataLog \cup \{newSensorData?\}$

$sendCmd \mathrel{\widehat=} \fatsemi data : transmitter.queue$
$\bullet\ transmitter.Consume \fatsemi SetPeriod$
$\fatsemi command.Emit \parallel LogData$
$sendRsp \mathrel{\widehat=} \fatsemi data : sensor.queue$
$\bullet\ sensor.Consume \fatsemi response.Emit$
$\parallel LogData$

without any alterations. Once again, the message exchange is recorded in the data list through the *LogData* command.

With all required classes defined, the schema of the *EmbeddedControlSystem* in Figure 4 can be defined as follows:

EmbeddedControlSystem

$sensorDriver : DeviceDriver\textcircled{C}$
$transmitterDriver : DeviceDriver\textcircled{C}$
$controlLogic : ControlLogic\textcircled{C}$

Finally, the union of the devices and the schema *EmbeddedControlSystem* forms the *Physical Twin*:

PhysicalTwin

$sensor : Device\textcircled{C}$
$transmitter : Device\textcircled{C}$
$ecs : EmbeddedControlSystem\textcircled{C}$

B. THE DIGITAL MODEL

Modeling and simulation are powerful methods that are utilized in various fields to evaluate complex systems, processes, and knowledge. They empower researchers, engineers, and decision-makers to examine real-world phenomena within controlled and virtual environments. This, in turn, enables them to make informed decisions and gain insights into the system under investigation. At the core of modeling lies the concept of mathematical modeling, which plays a pivotal role in formally capturing the essence of the system.

Mathematical models are representations of real-world systems that sue mathematical equations, relationships, and logical structures. They provide a mean to describe and quantify the behavior of a system. While mathematical models are not confined to any specific domain, in this work, we concentrate on their application in the engineering domain.

Before the advent of computers, machines were constructed primarily on drawing boards. This paradigm shifted with the introduction of computer-aided designs Computer-aided Designs (CADs), enabling the creation of 2D and 3D models that could be easily shared and replicated with others. Over the past decades, advancements in tooling and computational power have facilitated the substitution of real prototypes with virtual prototypes. This transition significantly reduced the number of design cycles and lowered the design costs. When components of a system are governed by mathematical relationships, virtual prototypes can be rigorously tested in simulations under a wide range of conditions. This allows for the evaluation of potential design weaknesses, providing immediate feedback on design decisions.

The *Digital Model* serves as a central component of a digital twin. However, most definitions merely mention digital models, assuming that researchers share a common understanding of what a model entails. This often leads to the assumption that a CAD model constitutes the entirety of a digital model, while a simulation is considered something more than a digital model, despite both being forms of mathematical models. Hence, we define a digital model as follows:

> *Definition 7 (Digital Model):* A digital model describes an object, a process, or a complex aggregation. The description is either a mathematical or a computer-aided design.

This definition encompasses various aspects of digital modeling, including the use of CAD as the foundational model for system design, its utilization within simulation tools involving complex processes, and even purely mathematical models.

1) INTRODUCING THE STATE MACHINE EXAMPLE

Although the physical twin is defined as including (autonomous) behaviors instead of a state machine, this

example can also be implemented as a finite state machine, where one can model its different states as follows [42]:

$$\mathcal{M} = (Q, \sum, \delta, q_0, F) \tag{1}$$

A finite state machine $\mathcal{M}$ can be represented by a 5-tuple M, which consists of a finite set of states Q, a finite set of input symbols known as the alphabet $\sum$, a transition function *delta* defined as $\delta : Q \times \sum \rightarrow Q$, an initial or starting state $q_0 \in Q$, and a set of accept states $F \subseteq Q$. The creation of state machines, often done using tools such as LabView, remains a common approach employed by engineers for programming machines. This practice falls within the scope of the provided definition of a digital model.

The state machine of the embedded control system can be defined as follows:

- $Q = \{STANDBY, ACTIVE, OFF\}$
- $q_0 = STANDBY$
- $\sum = \mathbb{Z}$
- $\delta : Q \times \sum \rightarrow Q$

The corresponding UML state diagram is presented in Figure 6. Upon initiation, the initial state is *STANDBY*, with the corresponding period value for the sensor's sampler rate set to 0, indicating that no samples are taken at this point. If a command with a value $x \in \sum$, where $x > 0$, is issued, the state machine transitions to the *ACTIVE* state. Conversely, if a command with a value $x = 0$ is received, the state reverts to *STANDBY*. For values of $x < 0$, the state of the system changes to *OFF*.

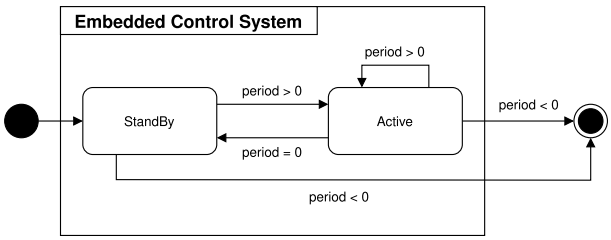

FIGURE 6. UML state machine diagram of the embedded control system formalized for the physical twin.

2) OBJECT-Z FORMALIZATION

This state machine can also be specified in Object-Z. First, the class diagram is displayed in Figure 7. *STATE* is the parent class:

STATE

$\upharpoonright(execute)$

execute

The *execute* method will be internally overwritten by the child states. For this example, the specific code that is executed is irrelevant. The states in which the state machine can be in are defined as subclasses:

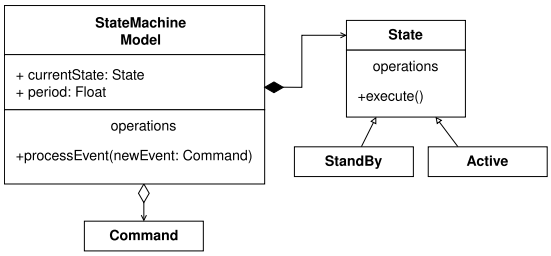

FIGURE 7. UML class diagram for the state machine.

ACTIVE

STATE

STANDBY

STATE

OFF

STATE

The *EventStateMachine* encapsulates the logic responsible for state changes upon receiving *COMMAND* events and maintains both a *STATE* (*state* is also the variable) and a *period*, which is a number. Initially, the *period* is set to 0, corresponding to the initial *state* set as *STANDBY*. The *ProcessEvent* function is responsible for modifying the state of the state machine in response to incoming events.

EventStateMachine

$\upharpoonright(\text{INIT}, ProcessEvent, state)$

$state : \downarrow STATE$

$period : \mathbb{Z}$

INIT

$period = 0$

ProcessEvent

$\Delta(state)$

$newEvent? : COMMAND$

$newState! : \downarrow STATE$

$state' = newState!$

It is important to note that, at this stage, the *EventStateMachine* has no connection to the physical twin. All modifications and updates are made manually, and there is no automatic synchronization between the digital model and physical twin. The schema for the digital model than includes the state machine:

DigitalModel

$\upharpoonright(\text{INIT}, ProcessEvent)$

$stateMachine : EventStateMachine©$

INIT

$stateMachine.\text{INIT}$

$ProcessEvent \mathrel{\widehat{=}} stateMachine.ProcessEvent$

C. THE DIGITAL TEMPLATE

In their initial definition of digital twins, Grieves and Vickers [22] view the digital twin as a collection of information necessary for constructing and monitoring the physical object. Specifically, the digital twin prototype can be regarded as a virtualized set of blueprints, bills of materials, technical manuals, and similar documentation. When combined with a digital model, which can be used to extract all the information needed to create blueprints and bills of materials, it can indeed be employed to construct and maintain the physical twin. However, this approach does not completely virtualize the physical twin, as later demonstrated by the example of the OSI Model in Figure 17 on Page 75357. Thus, the early interpretation of this definition does not fully realize a digital twin of a physical twin.

To encompass all available materials for constructing and maintaining the physical twin, including the software running the physical twin and the digital model, these components can be bundled together into a comprehensive package. We refer to this bundle as the Digital Template:

> *Definition 8 (Digital Template):* A digital template serves as a framework that can be tailored or populated with specific information to generate the physical twin. It encompasses the software operating the physical twin, its digital model, and all essential information needed to construct and sustain the physical twin, such as blueprints, bills of materials, technical manuals, and similar documentation.

Grieves and Vickers [22] initially defined a digital template as a digital twin prototype. However, in a recent paper, Grieves [43] elaborates on the definition of a digital twin prototype. The digital twin prototype encompasses all products that can be manufactured, including their variants. This concept evolves over time, from conception to the creation of the initial manufactured article [43]. We still consider that early iterations of the digital twin prototype are only a digital template. Nonetheless, upon full development, they may also incorporate the digital twin prototype definition presented later in this work.

1) OBJECT-Z FORMALIZATION

The UML class diagram of a digital template is depicted in Figure 8. The digital template includes all documents that either describe the physical twin or are required to build it. Furthermore, it includes the digital model from which the real system is derived and the software that operates the physical twin later. For an Object-Z formalization, the general class *Document* is defined as follows:

Document

Specific types inherit from the *Document* class:

BluePrint
Document

TechnicalManual
Document

BillOfMaterials
Document

Generic
Document

The schema for the digital template includes all the documents, the embedded control software, and the digital model:

DigitalTemplate

$documents : \mathbb{P}\, Document$
$ecs : EmbeddedControlSystem$
$digitalModel : DigitalModel$

D. THE DIGITAL THREAD

With the development of cyber-physical systems, machines have begun interacting with servers tasked with monitoring and controlling them. This paradigm also applies to digital twins. In this context, the communication channel facilitating such interaction is referred to as a digital thread. Taking inspiration from Leiva [44], we define the digital thread as follows:

> *Definition 9 (Digital Thread):* The digital thread refers to the communication framework that allows a connected data flow and integrated view of the physical twin's data and operations throughout its life-cycle.

Data accumulated from physical objects can only be preserved if these objects have an interface for storing the generated data. Similar to the definitions of a digital twin, there is currently no universally accepted and standardized solution for digital threads because of their diverse applications across various domains. Furthermore, it is crucial to understand that the digital thread encompasses more than just the communication protocol. It also involves applications and functionalities that assist in tasks such as monitoring, analysis, planning, and execution. These applications have the capacity to incorporate and share knowledge derived from the digital template and the gathered data, preserving the physical twin's evolution over time [32].

1) OBJECT-Z FORMALIZATION

The UML class diagram for a digital thread between the previously formalized physical twin and a digital twin, which will be defined later in this paper, is illustrated in Figure 9. The *DigitalThread* exists of a *PTtoDTConnection* that sends measurement and status messages (see the *RESPONSES* Object-Z class) and the *DTtoPTConnection*, which sends commands to the physical twin. To send data, a *TransmitterDriver* is used to establish a *Connection*. Note that this connection is not between a *DeviceDriver*

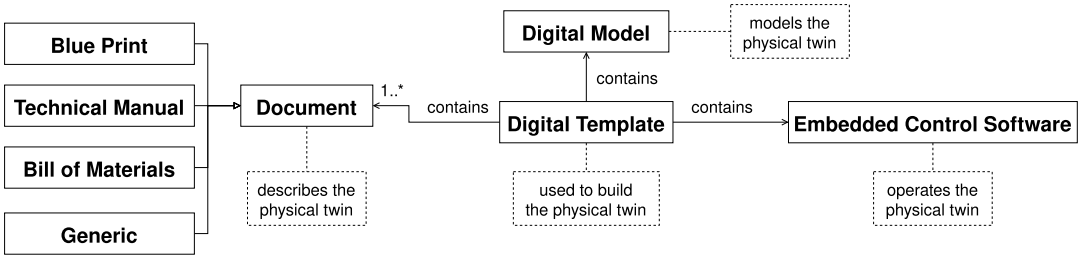

FIGURE 8. Relationship diagram of the digital template concept.

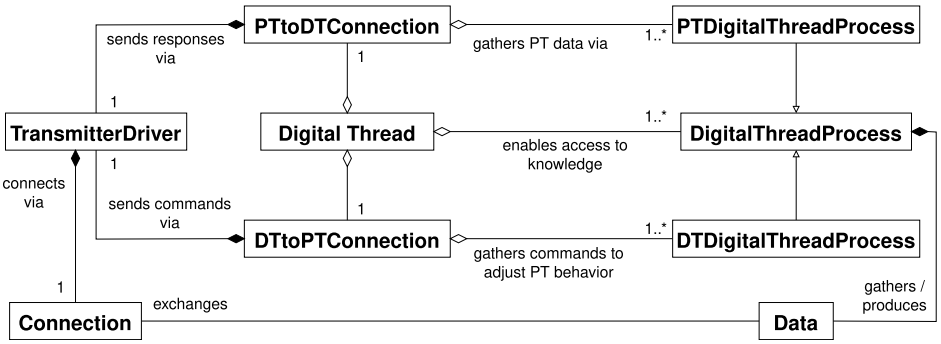

FIGURE 9. UML class diagram of a digital thread.

and a *Device* but between two transmitters, e.g. using the LoRaWAN protocol.

Both connection types gather data from processes (*DigitalThreadProcess*). In general, these processes can differ for each digital thread. Referencing our example again, the *ControlLogic* represents a *PTDigitalThreadProcess*, since it forwards all sensor messages to the transmitter, which then can transmit the data to the digital twin. On the digital twin's side, the *DTDigitalThreadProcesses* can include many different kinds of processes. However, at least one process is included: the process that decides which command is sent to the physical twin to adjust its sampling rate.

Since the digital thread is meant to show the evolution of the physical twin over its life-cycle, all the gathered data has to be stored in some form of a database. Hence, the database is a *DigitalThreadProcess*, which is part of the digital thread.

Formalizing this with Object-Z, we first define the *DigitalThreadProcess*:

DigitalThreadProcess

$$knowledge : \mathbb{P}\, DATA$$

A *DigitalThreadProcess* has a set of *DATA* that can be shared with the corresponding twin counterpart. *PTDigitalThreadProcess* and *DTDigitalThreadProcess* are derived classes:

PTDigitalThreadProcess ≙ *DigitalThreadProcess*

DTDigitalThreadProcess ≙ *DigitalThreadProcess*

These data are sent via the *Connection* in the *PTtoDTConnection*, which is defined as follows:

PTtoDTConnection

$$connection : TransmitterDriver\,©$$
$$collectFrom : \mathbb{P}\, PTDigitalThreadProcess$$

The counterpart is the *DTtoPTConnection*:

DTtoPTConnection

$$connection : TransmitterDriver\,©$$
$$collectFrom : \mathbb{P}\, DTDigitalThreadProcess$$

Both classes form the *DigitalThread*:

DigitalThread

$$ptData : PTtoDTConnection$$
$$dtData : DTtoPTConnection$$

E. THE DIGITAL SHADOW

To fully harness the potential of the digital thread, a process situated at either end of the digital thread must consolidate all the disparate elements into a platform that users can utilize to gain insights into the current state of the physical twin.

In the context of the digital twin concept, this role is fulfilled by the digital shadow. The term "digital shadow" emerged approximately a decade ago in discussions surrounding the privacy implications of "Digital Footprints", wherein numerous traces of digital actions are left behind and potentially misused by mischievous actors [45]. Comparatively, while a digital footprint encompasses the entirety of an individual's online activities, a digital shadow in the context of the digital twin concept specifically focuses on capturing and analyzing data related to the behavior, performance, and environment of physical objects or systems in the digital realm. The digital shadow is defined as follows:

> *Definition 10 (Digital Shadow):* A digital shadow is the sum of all the data that are gathered by an embedded system from sensing, processing, or actuating. The connection from a physical twin to its digital shadow is automated. Changes on the physical twin are reflected to the digital shadow automatically. Vice versa, the digital shadow does not change the state of the physical twin.

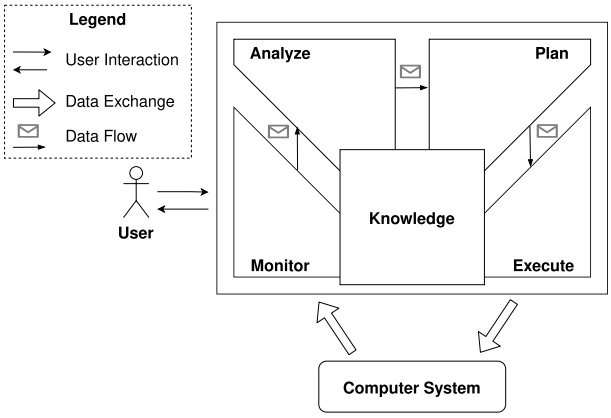

FIGURE 10. MAPE-K reference model for cyber-physical systems.

Because digital shadows represent the predominant state of most commercial digital twin products [33], extensive research is also being conducted on their development. Popular are model-driven approaches [46], [47], [48], [49], [50]. Michael and Wortman [50] describes an model-driven approach using the Monitor-Analyze-Plan-Execute over a shared Knowledge (MAPE-K) (Monitor-Analyze-Plan-Execute over a shared Knowledge) reference model. MAPE-K extends IBM's MAPE framework, with "K" signifying "Knowledge." The MAPE-K reference model provides a framework for automation of processes and the control loop for managing and optimizing computer systems. The different stages, see Figure 10, Monitor, Analyze, Plan, and Execute represent a specific stage or function within an autonomic computing system:

- **Monitor**: This is the first stage of the framework. In this phase, the system continuously collects data and monitors its own performance and the surrounding environment. This can involve data from various sensors, actuators, or monitoring tools that gather information about the system's behavior, resource utilization, and external conditions.
- **Analyze**: To gain insights into the system's behavior and performance, the data collected through monitoring, gets analyzed. The goal is to identify patterns, anomalies, and potential issues, and hence, to understand the current state of the system.
- **Plan**: Based on the analysis of the system's current state, the system formulates a plan for actions to be taken. This plan may involve adjustments, optimizations, or corrective measures aimed at improving the system performance, resource allocation, or other relevant parameters.
- **Execute**: In the last phase, the system performs the actions defined in the planning stage. These actions can be automatic or semi-automatic depending on the level of autonomy and control designed into the system. The system implements the planned changes to achieve the desired state.
- **Knowledge:** This component is critical for learning and adaptation. It involves maintaining a repository of historical data, models, policies, and best practices. The system uses this knowledge to make more informed decisions in subsequent iterations of the MAPE-K loop. Over times, the system becomes better at self-optimization and self-management by learning from past experiences.

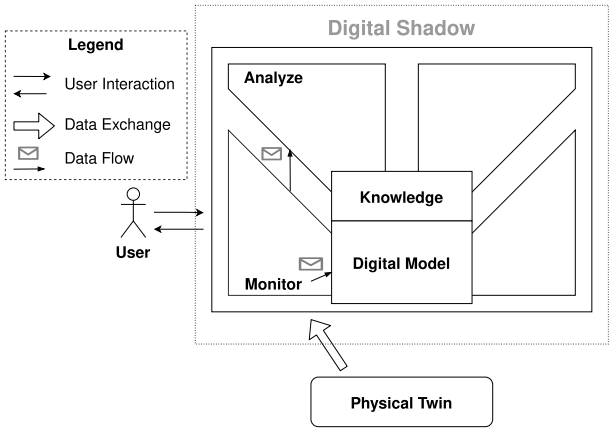

FIGURE 11. A digital shadow realized with the MAPE-K reference model. The Plan and Execution stages are not included, since there is also no data exchange from the Execution stage to the physical twin.

Since machines controlled by external computers/servers already exist in the form of CPSs, it is essential to clarify the distinction between a digital shadow and a CPS. As illustrated in Figure 11, the digital model has the same level of importance as that of Knowledge. However, a CPS does not necessarily have to include a model of the connected machine, and even if it does, this model may not always be

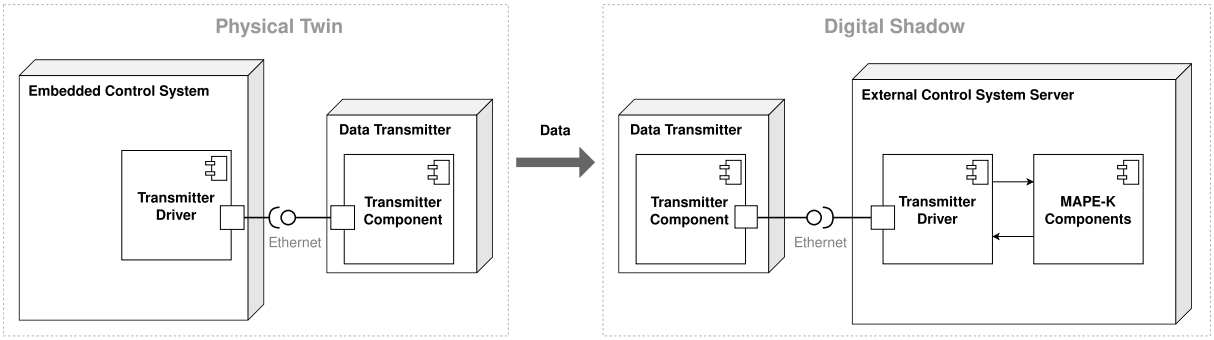

FIGURE 12. The digital shadow is deployed separately from the physical twin. The automated communication is unidirectional from the physical twin to the digital shadow. Status changes and all other data are sent by the physical twin and received by the digital shadow via transmitters. The digital shadow can reuse the transmitter driver from the physical twin. The logic inside the digital shadow is based on the MAPE-K model.

up-to-date. By contrast, for a digital shadow, this scenario is different. In the monitoring stage, all received data automatically update the digital model.

Another distinction is that a CPS can be used to operate the physical object directly. In contrast, a digital shadow's sole purpose is to monitor the physical twin and provide data for analysis, thus enabling insight into the received data. Consequently, the Planning and Execution stages of the MAPE-K model are not inherent components of the digital shadow. While they can be incorporated, the automated change of state in the physical object is not a function of the digital shadow.

1) OBJECT-Z FORMALIZATION

The configuration of the digital shadow for the physical twin, is illustrated in Figure 12. It is noteworthy that some parts of the physical twin are not depicted in the figure. The digital shadow operates on a server that establishes a network connection to the physical twin, either through a cable or wireless. In this example, we assume a wireless connection between the physical twin and its digital shadow.

The UML class diagram in Figure 13 is reduced to the two new classes for the *Monitor* and *Analyze* stages. All other classes and relationships are identical to the UML class diagram of the physical twin in Figure 5 on Page 75344. A direct association between the classes is not required, as they exchange data via an Observer pattern using the event handlers. Software packages to enhance these two classes, are again ignored in this example.

A digital shadow specification with Object-Z can be done as follows. The *Transmitter* and its operation are managed by the corresponding *TransmitterDriver*, both of which can be reused from the Object-Z formalization provided earlier for the physical twin. Additionally, all exchanged messages and the *EventHandler* can also be reused. What remains to be specified are the *Monitor* and *Analyze* stages of the MAPE-K reference model. The monitoring class *Monitor* is a *DTDigitalThreadProcess* and comprises two separate consumers: one for statuses and another for measurements:

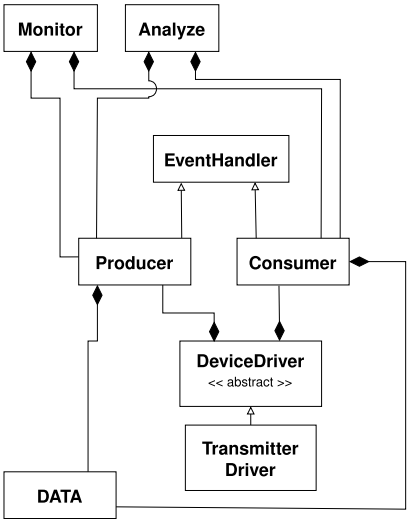

FIGURE 13. Reduced UML class diagram of the digital shadow. The MAPE-K stages *Monitor* and *Analyze* are included, all other classes and relationships are identical to the UML class diagram of the physical twin in Figure 5 on Page 75344.

$$
\begin{array}{l}
\textit{Monitor} \\
\lceil(\textit{INIT}) \\
\textit{DTDigitalThreadProcess} \\
\\
\quad \textit{statuses} : \textit{Consumer}\,\textcircled{C} \\
\quad \textit{measurements} : \textit{Consumer}\,\textcircled{C} \\
\quad \textit{emitter} : \textit{Producer}\,\textcircled{C} \\
\quad \textit{digitalModel} : \textit{DigitalModel} \\
\\
\quad \textit{INIT} \\
\quad \textit{statuses}.\textit{INIT} \wedge \textit{measurements}.\textit{INIT} \\
\\
\textit{handleState} \mathrel{\widehat=} \fatsemi \textit{status} : \textit{statuses.queue} \\
\qquad \bullet\ \textit{statuses.Consume} \\
\qquad \fatsemi \textit{digitalModel.ProcessEvent} \fatsemi \textit{emitter.Emit} \\
\textit{handleMeasurements} \mathrel{\widehat=} \fatsemi \textit{data} : \textit{measurements.queue} \\
\qquad \bullet\ \textit{measurements.Consume} \fatsemi \textit{emitter.Emit}
\end{array}
$$

Any status changes occurring in the physical twin are emitted as STATUS events, whereas all measurements are emitted as *MEASUREMENT* events. An emitter-producer is responsible for transmitting all consumed events to any registered listener. The most crucial component here is the *digitalModel*, which is an object of the previously specified *EventStateMachine*. All status changes are handled by the *handleState* function, which reads all the *STATUS* messages from the queue and forwards them to the digital model (state machine) for event processing. Subsequently, the result of the state machine's operation is emitted to all registered listeners. Because measurements do not affect the state machine's state, they are individually read from the queue via the *handleMeasurements* function and immediately relayed to all registered listeners. One such listener could be a database (part of the Knowledge state) responsible for storing all the data. It is worth noting that the *digitalModel* could also be a separate process that registers as a listener and consumes *STATUS* messages. In this example, the direct reference in the *Monitor* class was used for better demonstration purposes.

The *Analyze* stage is also a *DTDigitalThreadProcess* and can be a (semi-)automated stage of the MAPE-K model in the context of the digital shadow. In this particular example, the *Analyze* stage serves a singular purpose, which is to verify whether the received state from the physical twin aligns with the state of the digital model or not. The results of this comparison can then be emitted to all registered listeners. One potential listener could be a service responsible for notifying a user if any disparities in states were detected. Nonetheless, independent from the MAPE-K model, the analysis from the monitored events could also be performed manually by a user, as no further stage is following:

Analyze

$\restriction(\textsc{Init})$

DTDigitalThreadProcess

consumer : *Consumer*Ⓒ
emitter : *Producer*Ⓒ
digitalModel : *DigitalModel*

Init

consumer.*Init*

CheckStatuses

ptState? : *STATUS*
equal! : $\mathbb{B}$

compare $\widehat{=}$ ⨟*message* : *consumer*.*queue*
• *consumer*.*Consume* ⨟ *CheckStatuses*
⨟*emitter*.*Emit*

With these processes, the *DigitalShadow* schema can be defined. Since the MAPE-K example is only used for a better visualization of the concept, we use a more generic schema definition for the digital shadow:

DigitalShadow

digitalModel : *DigitalModel*Ⓒ
DThreadProcesses : $\mathbb{P}$ *DTDigitalThreadProcess*
DTtoPTConnection : *DTtoPTConnection*Ⓒ

Please note that no data are sent from the digital shadow to the physical twin. The *DTtoPTConnection* solely receives data from the physical twin.

F. THE DIGITAL TWIN

After defining and specifying the digital thread and digital shadow, the subsequent step is to comprehensively define the digital twin. The model-driven approach using the MAPE-K reference model is well suited to more concretely illustrate the differences between a digital shadow and a digital twin. The digital twin expands upon the digital shadow by enabling the automatic synchronization of all alterations made to the digital model with the corresponding physical twin. This means that any changes made to the physical twin are mirrored in the digital twin, and vice versa. Ultimately, the digital twin evolves into a complete replica of the physical twin. To formulate this definition, we draw upon the digital twin definitions proposed by Saracco [32] and Trauer et al [30]:

> *Definition 11 (Digital Twin):* A digital twin is a digital model of a real entity, the physical twin. It is both a digital shadow reflecting the status/operation of its physical twin, and a digital thread, recording the evolution of the physical twin over time. The digital twin is connected to the physical twin over the entire life cycle for automated bidirectional data exchange, i.e. changes made to the digital twin lead to adapted behavior of the physical twin and vice-versa.

Extending the system utilized in this example results in the addition of an extra communication channel from the digital twin to the physical twin, as illustrated in Figure 14. In the previously shown Figure 12 on Page 75353, the digital shadow only facilitates communication from the physical twin to the digital shadow. Now, all modifications within the digital model are also transmitted from the digital twin to the physical twin.

Moreover, the MAPE-K model must be adapted to accommodate the digital twin. The *Monitor* and *Analyze* stages in this new model are identical to those in the digital

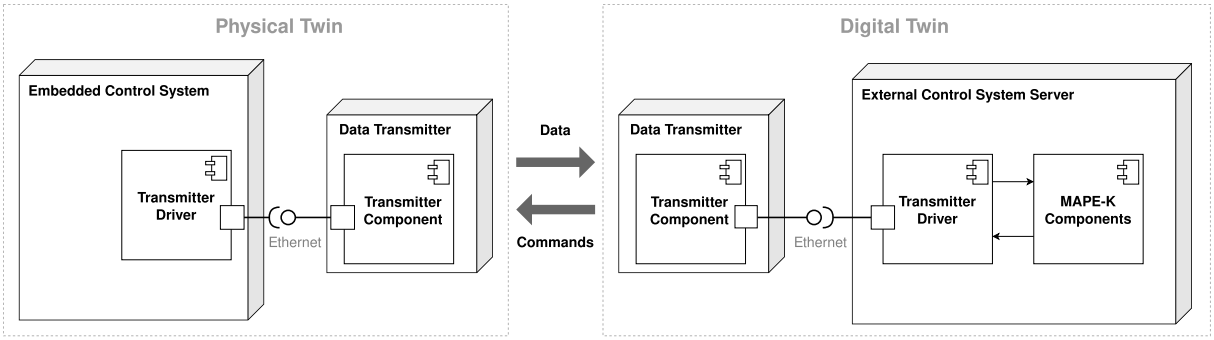

FIGURE 14. The digital twin extends the digital shadow in a way, that the communication between physical twin and digital twin is bidirectional. Additional to communication from the physical twin to the digital twin, all changes in the digital twin are automatically sent to the physical twin.

shadow, as shown in Figure 15. The *Plan* stage takes the analysis results and formulates an execution scenario for the *Execution* stage if changes to the physical twin are necessary. The key distinction from the original MAPE-K reference model lies in the digital twin, where the *Execution* stage interacts with the digital model. Only if a positive result is returned, the command is sent to the physical twin. Consequently, the digital model serves as the final control instance, and all incoming and outgoing changes are verified against the digital model.

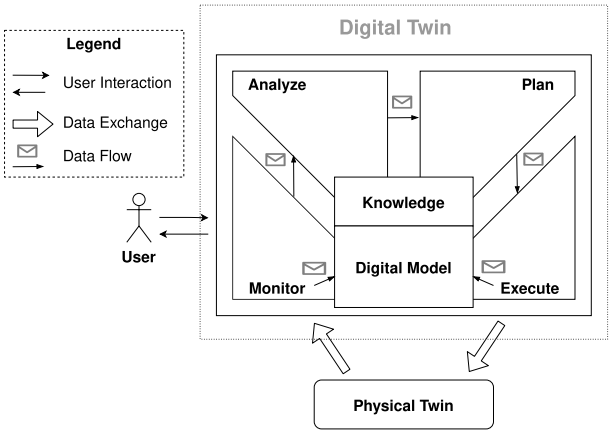

FIGURE 15. A digital twin realized with the MAPE-K reference model. The status change of the digital model and the corresponding data exchange from the Execution stage to the physical twin is fully automated.

1) OBJECT-Z FORMALIZATION

The Object-Z formalization of the digital twin can be built upon the digital shadow, incorporating two additional stages of MAPE-K as mentioned previously. Figure 16 includes only the MAPE-K relevant classes *Monitor, Analyze, Plan, Execute*, and the *EventHandlers*. All other classes are identical to the UML class diagram of the digital shadow in Figure 13. First, the *Plan* class is introduced:

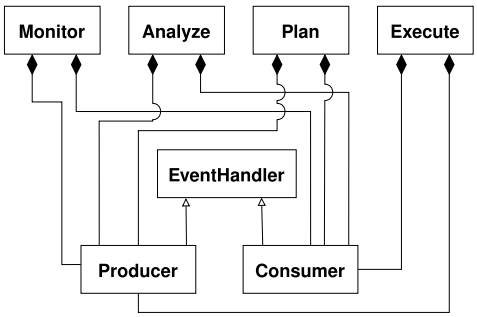

FIGURE 16. UML class diagram of the digital twin, including only the MAPE-K relevant classes Monitor, Analyze, Plan, Execute, and the EventHandlers used for data exchange. All other classes are identical to the UML class diagram of the digital shadow in Figure 13.

$$
\begin{array}{l}
\textit{Plan} \\
\hline
\uparrow(\textit{INIT}) \\
\textit{DTDigitalThreadProcess} \\
\\
\hline
\textit{consumer} : \textit{Consumer}\,\textcircled{C} \\
\textit{emitter} : \textit{Producer}\,\textcircled{C} \\
\textit{digitalModel} : \textit{DigitalModel} \\
\hline
\\
\textit{INIT} \\
\hline
\textit{consumer}.\textit{INIT} \\
\hline
\\
\textit{Planning} \\
\hline
\textit{data?} : \downarrow \textit{DATA} \\
\textit{result!} : \textit{COMMAND} \\
\hline
\\
\textit{plan} \mathrel{\widehat=} \fatsemi \textit{message} : \textit{consumer}.\textit{queue} \\
\qquad \bullet\ \textit{consumer}.\textit{Consume} \fatsemi \textit{Planning} \\
\qquad \fatsemi \textit{emitter}.\textit{Emit}
\end{array}
$$

This class is also a *DTDigitalThreadProcess* and includes a *Consumer* component to receive data from the *Analyze* stage. All results generated during the planning stage are emitted via the *Producer*. Similar to the other stages, the *Plan* stage has direct access to the *digitalModel*. However, in this example, no specific access details are provided.

The primary objective of this stage is to formulate a plan outlining which part of the physical twin's software needs modification, and how those modifications should be implemented. This task is executed through the *plan* function. All incoming data are consumed and subsequently passed to the *Planning* function. The resulting plan is then emitted to all registered listeners.

The last *DTDigitalThreadProcess* is the *Execute* class, which is kept straightforward as well. It receives all plans from the previous stage through the *execute* function. The commands are validated against the *digitalModel*, and the outcome is sent to the physical twin. The *transmitter* producer emits the command as an event to the *TransmitterDriver*, which subsequently consumes this command and transmits it to the physical twin:

Execute

$\restriction(\textsc{Init})$

DTDigitalThreadProcess

$plans : Consumer©$
$transmitter : Producer©$
$digitalModel : DigitalModel$

Init

$plans.\textsc{Init}$

ChangeState

$command? : COMMAND$
$newState! : \downarrow STATE$

$execute \mathrel{\widehat=} \fatsemi plan : plans.queue \bullet plans.Consume$
$\fatsemi ChangeState \fatsemi transmitter.Emit$

Please note that the concrete implementation of the digital model in this context is not critical. The digital model could exist as a separate process that receives events through consumers and provides responses via producers. Alternatively, it could collect all events from the *Execute* stage and independently transmit the results to the transmitter. There are numerous ways to realize this concept. However, the fundamental idea remains constant: changes to the digital model automatically trigger changes in the state of the physical twin without requiring any user intervention.

Similar to the digital shadow, we again define a generic schema *DigitalTwin* without the MAPE-K processes:

DigitalTwin

$digitalModel : DigitalModel©$
$DThreadProcesses : \mathbb{P}\, DTDigitalThreadProcess$
$DTtoPTConnection : DTtoPTConnection©$

The schemes *DigitalShadow* and *DigitalTwin* appear similar in this Object-Z formalization. The distinct difference is that the digital twin can send state changes automatically to the physical twin and thus, more *DTDigitalThreadProcess* are involved. In a mathematical representation, the difference is more obvious. Therefore, we compare the *DTDigitalThreadProcess* sets:

$$\begin{aligned} DSProcesses &= DThreadProcesses \in DigitalShadow \\ DTProcesses &= DThreadProcesses \in DigitalTwin \end{aligned} \tag{2}$$

The digital twin fully includes the digital shadow processes and extends them by including additional processes to send data back to the corresponding physical twin. Therefore applies:

$$DSProcesses \subsetneq DTProcesses$$

Following, the cardinality of the intersection of both *DTDigitalThreadProcess* sets is less than the cardinality of the digital twin's *DTProcesses*:

$$|\, DSProcesses \cap DTProcesses \,| \;<\; |\, DTProcesses \,| \tag{3}$$

At least one process is not a part of the intersection, and this process is the process that automatically sends changes to the physical twin.

G. THE DIGITAL TWIN PROTOTYPE

Today's existing modeling and simulation tools can rapidly create an abstract digital twin of a single component or process, and publish/subscribe architectures allow all messages between the processes to be captured and sent to a database or an IIoT platform. However, complex Industry 4.0 applications require the integration of multiple sensors and actuators into a larger system, posing a challenge with no simple solution yet. The embedded community still uses various industrial interfaces and communication protocols such as ProfiBus, ProfiNet, ModBus, CANOpen, OPC-UA, or MQTT, to name a few. Some are proprietary, making integration difficult, for instance, ProfiBus and ProfiNet.

Robust software testing for communication protocols is challenging because of the difficulty in emulating or simulating them. Software engineers frequently use mock-up functions in unit tests to bypass the need of establishing connections between processes or dependencies. This provides rapid feedback to developers through automated, repeatable tests executed quickly as part of the development workflow. However, even robust unit testing with comprehensive edge case coverage is insufficient. Therefore, some approaches use simulation tools that replace the communication protocols between hardware components with software interfaces. For Industry 4.0 applications, both approaches are inadequate, as insufficient testing can jeopardize the safety of human operators. Despite this, simulation tools are crucial for the development of Industry 4.0 applications as a source of data for sensors and actuators.

The software part of the connection can be formalized as shown in the *Communication* schema of the physical twin. The physical part, however, where the data are sent between *Device* and *DeviceDriver*, cannot be replaced in the same way. Hence, this approach still involves real

hardware in the development loop. During the development and testing, the *Connection* object is the central piece. Without a counterpart, no command is executed and no data are exchanged. Thus, engineers always require the hardware connected to the embedded software system that they develop and test. Replacing the *Connection* with a software mockup to circumvent HIL would result in a different *Connection* object that than used by the original *SensorDriver*. Thus, the configuration during development would differ from that of the real counter part it is deployed on later. Furthermore, not all communication protocols used in industry are properly mockable. This can be demonstrated by the example of ModBus and OPC-UA applications on the OSI-Model shown in Figure 17. Unlike Ethernet-based communication protocols that implement and cover all layers of the OSI-Model, communication protocols based on serial connections, such as ModBus or CANOpen, are placed on the model's 7th layer, the *Application Layer*. No additional host layers exist. Sending/receiving data are handled immediately by the *Data Link* and *Physical Layers*. This means that the physical hardware handles the necessary actions required for data exchange. Mocking these layers is difficult. On the other hand, communication protocols based on TCP, such as OPC-UA, can easily be mocked by opening a socket on the TCP layer and connecting another device to it. However, this is not true for serial protocols. On connection, the driver attempts to establish a connection to another device via RS232. As no device is connected, this would fail, and a connection error would be thrown.

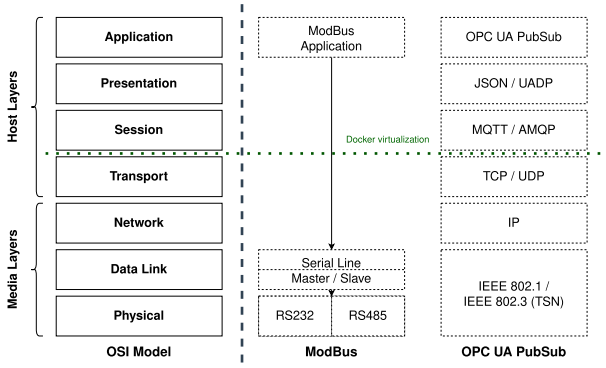

FIGURE 17. RS232 applications and communication can be visualized on the OSI layered model. The application on Layer 7 is directly connected to the RS232 API and driver (Layer 2) that uses the physical connection (Layer 1) to transmit data to other RS232 interfaces.

Replacing the entire physical twin during development and testing, which includes the hardware interfaces, leads to a fully virtual representation of the physical twin, and engineers do not necessarily need the hardware anymore for development. This is the main difference to the digital twin prototype definitions by Grieves and Vickers [22] and Grieves [43]. We define the digital twin prototype as follows:

> *Definition 12 (Digital Twin Prototype):* A Digital Twin Prototype is the software prototype of a physical twin. The configurations are equal, yet the connected sensors/actuators are emulated. To simulate the behavior of the physical twin, the emulators use existing recordings of sensors and actuators. For continuous integration testing, the DTP can be connected to its corresponding digital twin, without the availability of the physical twin.

1) OBJECT-Z FORMALIZATION

To reduce the dependency of the embedded software system on the hardware during development and testing, communication protocols such as RS232 need to stay on the host layers of the OSI-Model without the need of changing the original connection properties of a device driver. This circumvents the layers that include the hardware. However, rerouting the connection disconnects the device from the device driver. Rerouting only works if another process exists at the other end of the connection. So far, there is none. This is why not only the connection but also the device has to be emulated. To begin, the emulated connection is defined first. The Object-Z formalization for *EmulatedConnection* is as follows:

EmulatedConnection

$\lceil (\mathit{INIT}, \mathit{Read}, \mathit{Send}, \mathit{EmulateWrite}, \mathit{EmulateRead}) \rceil$

$\mathit{Connection}$

$\mathit{originalProtocol} : \downarrow \mathit{Connection}$

$\mathit{originalProtocol} \notin \mathit{Connection}$

INIT

$\mathit{type} = \mathit{TCP}$

EmulateWrite

$\mathit{data}? : \downarrow \mathit{DATA}$

$\mathit{forwardData}! : \downarrow \mathit{DATA}$

$\mathit{forwardData}! = \mathit{data}?$

EmulateRead

$\mathit{data}? : \downarrow \mathit{DATA}$

$\mathit{forwardData}! : \downarrow \mathit{DATA}$

$\mathit{forwardData}! = \mathit{data}?$

The *EmulatedConnection* object inherits from the abstract *Connection* class, and thus has all its properties and functions. This is shown in the OSI-Model in Figure 17. The safe way to stay in the host layers is to route all other communication protocols to *TCP* and from there back to the original protocol. Hence, the *EmulatedConnection* does not replace the connection objects of *Device* and *DeviceDriver*. Instead, it is an independent additional connection that provides

interfaces for a device emulator and a device driver to connect to with their original protocols. The *EmulatedConnection* then uses TCP and forwards all incoming data via the function *EmulateRead* and all outgoing data via the function *EmulateWrite* between the emulated device and device driver.

How can this be realized without reconfiguring the device or the device driver? Simply by using tools such as *socat* (SOcket CAT) [51]. *Socat* is a command-line utility that allows for bidirectional data transfer between two endpoints, typically over a network or through pipes. It is similar to the more well-known tool *netcat* but with support for multiple connection types and protocols (TCP, UDP, SSL, PTY, etc.). With two virtual serial ports (client and server) via *socat* for the emulator and the device driver, a connection can be established without the need to change the configuration. In the background, *socat* forwards the data between the ports via a TCP connection.

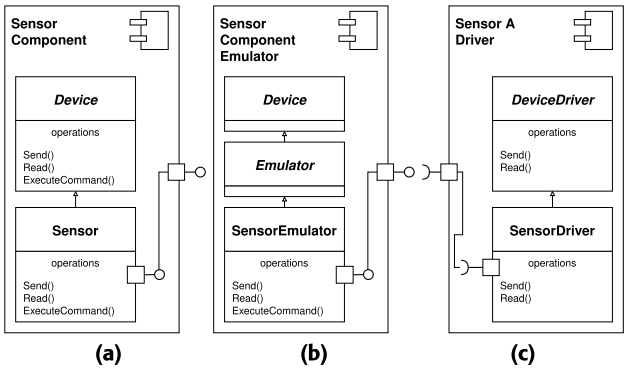

FIGURE 18. UML component diagrams for sensor and emulator components. The real SensorComponent in (a) can be replaced by an EmulatedSensorComponent (b) and the SensorDriver (c) cannot distinguish whether it is connect to the real sensor in (a) or the emulated on in (b).

A device emulator for a sensor could be similar to that shown in Figure 18. Similar to the real sensor, the *SensorEmulator* inherits all properties and functions from the generic *Device* class. There is only one difference: instead of executing a command and responding with the real result, the emulator uses virtual context for the response. Virtual context can be a list of previously recorded data from the real device or context provided by a simulation. In this example, we assume that the virtual context consists of previously recorded data with the real device. Formalizing the emulated device and connection with Object-Z requires the definition of another data subtype first. Since the sensor responds to commands with a *RESPONSE* type, a subtype of *RESPONSE* named *RECORDING* can be defined:

RECORDING
RESPONSE

The abstract class *Emulator* inherits all properties and functions from the abstract class *Device*, and *SensorEmulator* inherits from *Emulator*:

Emulator
Device

Although it may seem more obvious to inherit from *Sensor*, the emulator cannot inherit its properties and functions from there. Most devices are a black box for the developer, and vendors provide only a technical manual and support to interact with the device. Thus, an emulator only mimics the behavior of its real counterpart and provides its API with corresponding return values. However, this is sufficient to replace the real device with an emulator for development and testing. A developer is mostly interested in the connection and data exchange part, not the internal behavior of a connected device.

For abstraction reasons, the *Sensor* object in this example was very simple. That is why the *SensorEmulator* can also inherit all properties from *Emulator* and change the *ExecuteCommand* function to always return *RESPONSE* objects from the *virtualContext* set:

SensorEmulator
Emulator

$$virtualContext : \mathbb{P}\, RECORDING$$

ExecuteCommand

$$\Delta(virtualContext)$$
$$command? : COMMAND$$
$$result! : RECORDING$$

$$command? \in commandList$$
$$result! \in virtualContext$$
$$virtualContext' = virtualContext \setminus \{result!\}$$

The *SensorDriver* remains as it is and does not need any changes. The communication between an emulator and the *SensorDriver* can be specified as follows using *EmulatedCommunication*:

EmulatedCommunication

$$emulator : SensorEmulator$$
$$driver : SensorDriver$$
$$connection : EmulatedConnection$$

$$\forall x : emulator.commandList \bullet x \in driver.commandList$$
$$\forall x : driver.commandList \bullet x \in emulator.commandList$$

$$ToDrv \mathrel{\widehat=} emulator.Send \fatsemi connection.EmulateWrite \parallel connection.EmulateRead \fatsemi driver.Read$$
$$ToDev \mathrel{\widehat=} driver.Send \fatsemi connection.EmulateWrite \parallel connection.EmulateRead \fatsemi emulator.Read$$

The *EmulatedCommunication* object now includes an additional *Connection* object in the form of *EmulatedConnection*. The communication from the emulator to the device

driver, labeled as *ToDrv*, is now a composition of the connections from the device to the *EmulatedConnection*. From there, the data are sent to the device driver, where the *EmulatedConnection* receives it and forwards it to the connection defined by the device driver. The *EmulatedConnection* is not part of either the device/emulator or the device driver. Therefore, in this example, the *SensorDriver* cannot differentiate between whether it is connected to a real device or an emulator, which is the goal of our approach.

H. SUMMARY OF THE DIGITAL TWIN CONCEPT

The relationships between the different concepts are illustrated in the UML diagram in Figure 19. We extended the semi-formal approaches by Yue [19] and Becker et al. [20] for the digital shadow and the digital twin. A *Physical Twin* performs actions using real *Devices* in a *Physical Environment*. The *Physical Environment* is not a real class, but the real world context in which the *Device* operates. Changing behaviors lead to changes in the current *State* of the *Physical Twin*. Hence, the *Physical Twin* updates its *State* and sends the change of *State* via the *Digital Thread* to the *Digital Shadow*. Yue, Arcaini, and Ali [19] named this *Twinning*, see Figure 3 on Page 75342. Different to the formalization byYue, Arcaini, and Ali [19], the *Physical Twin* is not directly connected to the *Digital Twin* but via the *Digital Shadow*, which is included by the *Digital Twin*. In our Object-Z formalization of the *Digital Shadow* and *Digital Twin*, we illustrated the difference utilizing the MAPE-K model and showed that the *Digital Shadow* does not send any data to the *Physical Twin*. All *State* changes are received by the *Digital Shadow*, which then changes the *Digital Model*. Only the *Digital Twin* updates *State* changes similar to the change of *State* of the *Physical Twin*. Instead of *Physical Process*, the digital twin uses the *Digital Model*, which operates in a *Virtual Environment*, to change the *Physical Twin's State*. The special feature of the *Digital Twin Prototype* is that it is operated by the same *Embedded Control System* as the *Physical Twin*. Its software does not even recognize, whether *Physical Hardware* or *Emulated Hardware* is used. Hence, during the development phase, the *Digital Twin Prototype* can replace the *Physical Twin*. A *Digital Twin Prototype* executes commands on *Emulated Hardware* in a *Virtual Environment*. The *Virtual Environment* should mirror the real world, which can be realized via a *Simulation*. Notice that the *Digital Model* used by the *Digital Twin Prototype* is a different instance than the *Digital Model* updated by the *Digital Shadow*. To describe and construct the *Physical Twin* its *Digital Template* can be used, as it includes the *Digital Model* and the *Embedded Control Software*.

The formal specifications of the digital shadow and digital twin were achieved through a model-driven approach employing the MAPE-K reference model. Nonetheless, these methodologies have inherent drawbacks. A significant concern is the disconnect between the development of the physical twin and the digital twin. Developing the digital twin requires additional software development and ongoing maintenance efforts. Furthermore, the control logic of the physical twin must be either developed anew or emulated to ensure accurate planning and execution within the automated control loop. As the complexity of the physical twin increases, this process can become labor-intensive and contribute to accruing technical debt [8].

Fully automated testing of embedded control software on the physical twin often proves impractical due to the requirement of the corresponding hardware, resulting in potentially additional costs for spare hardware solely for the HIL test environment. Consequently, the physical twin is typically assessed using a manual HIL approach. Conversely, a digital twin, implemented with a model-driven approach, could undergo fully automated testing without necessitating the physical twin's hardware. However, alterations in the physical twin's behavior mandate corresponding adjustments in the digital twin prior to any software updates. If only the physical twin can be thoroughly tested including the real hardware, manual testing of the interaction between the physical twin and digital twin becomes necessary. In manual testing, additional time is required from testing engineers, involving tasks such as setting up the test bed and executing all tests. Moreover, the test execution may differs with each round of testing owing to human error. With automated testing, the virtual test bed setup does not require any manpower, and the test execution is always identical; however, it demands increased computational resources. This challenge prompted the introduction of the digital twin prototype.

The digital twin prototype employs the same embedded control software as the corresponding physical twin, with the addition that the sensor/actuators are emulated and the communication protocols between sensors/actuators, and the embedded control system are also virtualized. This configuration transforms the digital twin prototype into a virtual test bed of the physical twin, shifting development from an HIL approach to a SIL approach. Engineers can utilize the digital twin prototype to implement and evaluate new software features without the necessity of manual steps, such as setting up the real test bed, deploying the new version, starting the system, and executing the new code. By requiring fewer hardware resources solely for development purposes, costs are reduced, and it is also more sustainable with regard to material consumption. Moreover, the entire embedded control system of the physical twin can be integrated into automated test pipelines without the need for spare hardware.

The digital twin prototype can also be used to develop the digital shadow or digital twin. In our formal specification, we have already repurposed the physical twin's class *TransmitterDriver* for the digital shadow, a process applicable to all control logic aspects of the physical twin. The integration of the embedded control system with a digital model, such as in a simulation, is already inherent in the digital twin prototype. Initializing the digital twin prototype without sensors/actuators and connecting it to a receiving transmitter already constitutes a digital shadow of

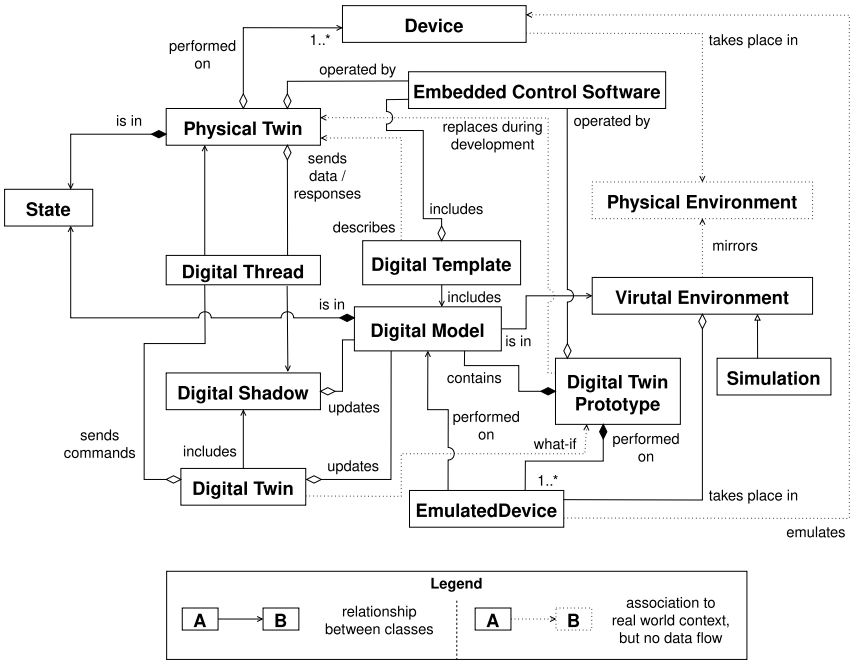

FIGURE 19. Relationships between physical twin, digital model, digital template, digital shadow, digital twin, and digital twin prototype.

TABLE 2. Classification of the concepts and definitions presented in Table 1 on Page 75342 with our formalized digital twin concept.

Concept	Primary Use Case	Definitions from Related Work
Digital Model	A snapshot of a mathematical model or computer-aided design used primarily for initial simulations.	[26], [18] (Digital Model Definition)
Digital Shadow	Monitors the corresponding physical twin in (near) real-time and internally updates an instance of the digital model.	[17], [28], [31], [18] (Digital Shadow Definition)
Digital Twin	Operates and monitors the corresponding physical twin in (near) real-time and internally updates an instance of the digital model.	[22], [30], [32], [18] (Digital Twin Definition)
Digital Template	Bundles all information, the digital model, and the embedded software system required to build the physical twin.	[21] (Digital Twin Prototype Definition)
Digital Twin Prototype	Contains the physical twin's embedded software system but uses emulators instead of real sensors/actuators. Replaces the physical twin during development and in CI/CD pipelines.	-

the physical twin. By substituting the physical twin with the digital twin prototype in the test pipelines and initiating a second, reduced digital twin prototype as the receiving end in the form of a digital shadow, automated testing of the physical twin and digital shadow interaction becomes possible. This capability can also be extended to the digital twin, allowing the digital twin prototype to evaluate commands intended for the physical twin and identify potential software errors before they affect the system. Reusing the embedded control system for the digital shadow/digital twin reduces the need for parallel control logic implementation between the physical twin and digital twin. New software modules are only developed to enhance user interaction with the physical twin and to gain deeper insights into the system. This approach of reusing software modules from physical twin development for a digital twin was also described by Heithoff et al. [52] as a means to improve the sustainability of embedded control software.

A digital twin prototype not only streamlines the software development process of a physical twin but also serves various analytical purposes. For example, it can be employed for evaluating "what-if" scenarios [32] or for conducting system examination and risk management using the HAZOP (Hazard and Operability) method. HAZOP involves examining the system or process in detail, considering every conceivable deviation from the intended operation or design caused by various factors, such as human error, equipment failure, or external events [53]. For each potential deviation, possible causes, consequences, and safeguards to mitigate the risk are identified. By furnishing a detailed, dynamic, and data-driven understanding of the physical twin, the digital twin prototype facilitates a more effective identification, analysis, and mitigation of risks, ultimately enhancing the safety and reliability of the embedded system.

To conclude, Table 2 extends Table 1 to provide an overview of the various concepts defined and discussed throughout this paper, offering a guideline on the contexts in which they are used. We reclassify the definitions presented in Table 1 into our formalized digital twin framework.

IV. APPLICATION OF THIS CONCEPT

In the following, two projects are presented, where the previous definitions and methods have already been applied in real life contexts. The validation of research results and the reproducibility of experiments are integral aspects of good scientific practice [54]. However, replicating the conducted field experiments from our Autonomous Robotic Networks to Help Modern Societies (ARCHES) demonstration mission or the SilageControl case study using similar hardware can be expensive. To facilitate independent replication of the digital twin prototype approach by engineers and other researchers, we have developed a digital twin prototype using cost-effective hardware, specifically a PiCar-X by SunFounder [55]. This digital twin prototype is based on the ARCHES Digital Twin Framework [56] and is publicly available on GitHub [57]. More comprehensive details regarding the PiCar-X digital twin prototype will be presented in a separate publication.

A. FIELD EXPERIMENT WITH UNDERWATER OCEAN OBSERVATION SYSTEMS

The digital twin prototype approach was developed for a network of ocean observation systems and tested during the research cruise AL547 with RV ALKOR (October 20-31, 2020) of the Helmholtz Future Project ARCHES (Autonomous Robotic Networks to Help Modern Societies) [58]. In ARCHES, with a consortium of partners from the Alfred-Wegener-Institute Helmholtz Centre for Polar and Marine Research, the German Aerospace Center, the Karlsruhe Institute of Technology, and the GEOMAR Helmholtz Centre for Ocean Research Kiel, several digital twin prototypes for ocean observation systems were developed. The major aim of this project was to implement robotic sensing networks, which are able to autonomously respond to changes in the environment by adopting its measurement strategy, in both space and in the deep sea.

The ocean observation network realized within the project ARCHES comprised five ocean observation systems, constructed at the Alfred-Wegener-Institute and GEOMAR. One of the challenges was the geographical distance between these two institutes, compounded by the fact that each system differed in construction, payload, and software. These systems are costly, prompting research institutes to maximize their usage across numerous missions. Traditionally, researchers designed experiments and engineers implemented the necessary system logic. Once deployed underwater, the systems operated autonomously for up to a year without direct intervention, and researchers could only evaluate the results after retrieval. This approach was primarily due to the limited battery life and the challenges of underwater wireless communication, as electromagnetic waves are absorbed quickly in water. As a result, acoustic communication, which only recently became reliable and still suffers from low bandwidth, was utilized [58]. However, the high costs associated with both the ocean observation systems and the research cruises make minimizing the risk of experiment failure a high priority. Consequently, robust testing of hardware and software has become increasingly critical.

The goal of ARCHES extended beyond simply developing an ocean observation system that could be remotely monitored. The project also aimed to facilitate the execution and modification of individual experiments directly from the research vessel. Initially, the participating systems in the network lacked a communication link with other systems or the research vessel at the ocean surface. With ARCHES, this was set to change as all systems were to be equipped with acoustic modems for data exchange. Rather than completely rewriting the existing software to integrate communication across all systems, we opted to install RaspberryPis between the acoustic modem and the existing controllers of the ocean observation systems. This strategy allowed us to reuse not only the device driver for the acoustic modems - which were identical across all systems - but also the entire communication stack, thereby forming a digital thread.

We implemented a microservice architecture using the middleware Robot Operating System (ROS) [59] and encapsulated all microservices in Docker. This approach also enabled us to treat the existing software on the ocean observation systems as a connected device, requiring us to develop only specific device drivers for each system. Given the challenge of not having access to the actual systems, we adopted the digital twin prototype approach. We virtualized the physical connections, including RS232 and RS485, using socat [51], and developed an emulator for the existing software on the ocean observation system. The use of digital twin prototypes allowed for the development and testing of scenarios before missions began. Furthermore, we integrated automated testing into CI/CD pipelines within GitLab to ensure high software quality. The development process of these digital twin prototypes is detailed further in the work by Barbie et al. [60].

To enable researchers to observe and control experiments in near real-time, it was necessary to have software on the research vessel that could receive and analyze data. In the context of cyber-physical systems, user interfaces are often developed to monitor and operate these systems, but this would have required additional effort and a new software system. Although feasible, this method overlooked a critical issue: detecting hardware or specific software failures in an underwater ocean observation system. The limited bandwidth underwater makes sending error messages impractical, as they must be sent in small chunks, consuming too much energy. Our solution involved using a digital twin for each physical twin underwater. With digital twin prototypes already developed, we only had to ensure that all data, status changes, and control operations occurring on the underwater systems were mirrored to the corresponding digital twin on the research vessel. The publish/subscribe mechanism of ROS allowed us to tap into all publishers, ensuring that the same messages sent to the physical twin

were also received by the digital twin. This setup meant that if a message caused an error in a ROS process on the physical twin, the same message would similarly affect the digital twin, since their software configurations were identical. This way, the intended message containing the data would be sent, and any potential errors would be logged in the digital twin.

To further reduce energy consumption for data exchange, all ROS messages were converted into a binary serialization using Apache Avro [61] before being sent, and then deserialized back into ROS messages at their destination. Therefore, the digital twin prototypes served a dual purpose: they allowed us to maintain digital shadows of the physical twins underwater without additional software development, and they enabled us to test commands for correct behavior before sending them to the underwater systems. In effect, the digital twin prototypes functioned as actual digital twins, eliminating the need for further software development. The digital thread we developed facilitated uniform data exchange across all systems.

The digital twin prototype approach was tested during the ARCHES demonstration mission in the Baltic Sea. Various scenarios were executed for evaluation, including monitoring the ongoing experiments of individual systems, changing the status of the entire network from the research vessel, and facilitating collaboration between a physical twin and its digital twin in a potential experiment. A detailed field report on this demonstration mission was published in Barbie et al. [58]. Moreover, Figure 20 features a screenshot from a video capturing the moment when two ocean observation systems received a broadcast from the research vessel. This video is accessible in our GitHub repository [57], [58]. The ARCHES Digital Twin Framework, developed during the project, was released as open-source software [56] as well. This framework was used to establish a digital thread between all physical twins and their corresponding digital twins. Throughout the mission, all messages exchanged between the digital twins and physical twins were recorded. These records can now be utilized by the sensor and actuator emulators to replay the interactions within the CI/CD pipelines during testing. Leveraging real data from real-life contexts provides a ground truth in our data, enhancing the quality of the automated tests.

B. CASE STUDY WITH SMART FARMING APPLICATIONS

As the digitalization of agricultural processes promotes the use of digital twins for various use cases [63], we also report in Barbie [62] on a case study that experimented with the digital twin prototype approach for a smart farming application utilizing methods and ideas we developed during the project ARCHES. In this paper, we only provide a brief overview.

The smart farming project SilageControl, with a consortium of the Silolytics GmbH, Blunk GmbH, and Kiel University, used digital twins to adopt the digital twin prototype approach for development and maintenance. The major goal of SilageControl is to improve the process of

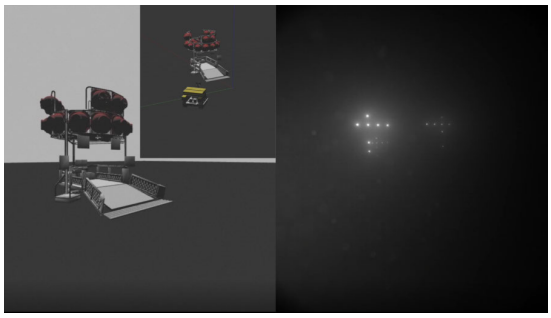

FIGURE 20. On the left side of the video, the digital twins of Mansio - a stationary sensor hub - and Viator, a crawler equipped with a camera, are displayed. In the captured scenario, we broadcast a command to the network, altering the behavior of all systems. The video captures Mansio activating its lights and Viator slowly moving backward. A split-screen format is used to show that the digital twins replicate the actions of their physical counterparts with a small delay [58], [62].

silage making, i.e. the fermentation of grass or corn in silage heaps. In order to avoid mold formation, the harvested crop is compacted using heavyweight tractors. As displayed in Figure 21, these tractors are equipped with a sensor bar, which includes GPS sensors, an inertial measurement unit, and a LiDAR sensor. In combination, the sensors enable the continuous and accurate representation of the tractor's position / orientation and the shape and volume of the silage heap.

Silolytics faces several challenges similar to those we encountered in ARCHES. For instance, the equipment is expensive, and as a small company, they lack the resources to purchase hardware solely for testing purposes. Since silage production is seasonal, the hardware is in use in the field during these times and cannot be accessed by engineers for further development. Additionally, they encounter issues such as the GPS sensor not functioning correctly indoors, necessitating outdoor testing [62]. They adapt the digital twin prototype approach to improve the sensor platform independent from the current season. The first field experiments were conducted from May to October 2022. During this period, sensor data was be recorded to further improve the accuracy of the physical models and create scenarios for the automated testing of future features. The scenarios derived from real field experiments once again offer a basis of ground truth for the tests conducted during development. The digital twin prototype in SilageControl is connected to a GAZEBO [64] simulation. Our PiCar-X example can be used to explore this setup [57], [62].

C. OTHER USE CASES, RISKS AND ETHICAL CONCERNS

In this paper, we have defined and described the digital twin concept from a manufacturing perspective. However, research on digital twins is also being conducted in other fields. Examples include the Digital Twin of the Earth project [65] and digital twins of humans [66]. These research areas introduce new perspectives to the digital twin concept and thus employ various definitions, which often resemble those used in the manufacturing sector. However, this also

(a) Sensor bar in lab environment

(b) Sensor bar mounted on a tractor

FIGURE 21. Sensor bar which monitors the process of silage making.

results in these concepts becoming mixed, similar to what was described in Section II, with each claiming to develop a digital twin. That is why it is necessary to distinguish between digital models, digital shadows, digital twins, and digital twin prototypes, not only for technical but also maybe ethical reasons.

For instance, a digital twin as defined in this paper, would mean for the project Digital Twin of the Earth that changes to the digital twin automatically result in adjustments to the physical system, thus directly influencing our planet. This could involve actions similar to geoengineering. However, the actual focus of the project is on improving climate models, analyzing the resulting data, and subsequently providing consulting and recommendations for action to governments and societies [65]. Therefore, this merely represents a digital model or - depending on the number of available sensors around the world - a digital shadow. From a risk and ethical perspective, this has very different implications compared to, for instance, geoengineering.

Similarly, this applies to the digital twin of a human. Does a status change in the digital twin directly influence the well-being or actions of the physical counterpart (the human), or does the digital twin in this context serve only to collect and better analyze data about the person, and to more readily provide this information during medical emergencies? Collecting comprehensive data about a person naturally raises ethical questions and risks. How is this data secured, who has access to it, and what happens if authoritarian states use it to suppress citizens or if mischievous actors misuse it?

Ethical questions and risks also arise in the manufacturing domain. On the one side, for use cases including manufacturing plants or unmanned vehicles, digital twins solve several challenges enterprises face and open new opportunities. On the other side, for use cases involving the cooperation with or transport of humans, the employment of digital twins can pose a security risk. For instance, autonomous passenger air craft with a remote digital twin on the ground can be hijacked by mischievous actors. The same holds true for cars, ships, and other means of transport.

We do not intend to take sides on these issues, as they are matters that societies, associations, and governments need to discuss and resolve. Nonetheless, with the increasing significance of digital twins, the topic of risks and ethical implications is also receiving more attention in research [67], [68].

V. CONCLUSION AND FUTURE WORK

Digital twins find applications across all layers in Industry 4.0 scenarios. However, there is confusion in the definitions of digital models, digital shadows, digital twins, digital twin prototypes, and the digital thread. While many studies attempt to list and categorize these differences, a formal description has been lacking. Therefore, in our digital twin concept, we formally specified the various components, ranging from the physical twin to the digital twin, culminating in a fully virtualized digital twin prototype capable of substituting the physical twin during development. We extended the digital twin concept by the *digital template*. A digital template describes the physical twin and is used to build it. It includes the digital model of the physical twin, describing documents, and the embedded control software operating it. To emphasize the distinctions among these different facets of the digital twin from a software engineering standpoint, we provide an Object-Z formalization for each component.

Furthermore, we have provided real-world application examples to illustrate the practical context. A proof of concept for the formal specifications was demonstrated in a demonstration mission showcasing the viability of digital twins in ocean observation systems. Moreover, we offered insight into how this approach could be employed in the SilageControl smart farming project, which aims to enhance the silage-making process through the development of a sensing platform. The digital twin prototype approach is not limited to the presented use cases but can be utilized for all embedded software systems regardless of their application scenario.

The usage of digital twin prototypes transforms the way how embedded software systems are developed. By starting with the emulation of hardware - sensor by sensor, actuator by actuator, and communication protocol by communication protocol - the development of embedded software systems becomes an iterative process. Furthermore, the integration of a fully operational digital twin prototype heralds a shift towards collaborative efforts between engineers and domain experts, regardless of their physical location or connection to the hardware. This approach diverges from others, such as model-driven approaches. Rather than relying solely on mathematical models and simulations with software modules that emulate or implement the behavior of the physical twin, a digital twin prototype incorporates the embedded control software of the physical twin. This enables the reuse of control logic by the digital twin, reducing the need for parallel

development between the physical twin and the digital twin, and consequently mitigating potential technical debt.

In addition to reducing the time required for testing by switching from HIL to SIL testing with digital twin prototypes, this approach also avoids expenses for redundant hardware and paves the way for more efficient and sustainable development workflows that are otherwise difficult to implement for embedded software systems. Digital twins become a key enabler for fully automated integration testing of embedded software systems in CI/CD pipelines. While building, testing, and releasing of software is possible for embedded software just like in other fields of software engineering, integration testing with hardware interaction is expensive, due to the HIL testing, and is often done manually. Thus, the integration tests are a bottleneck in the verification and validation activities, and hence, the release of new software.

In summary, digital twins have the potential to enhance the quality of embedded software systems, concurrently reducing costs and accelerating development speed. These benefits align with the challenges we elaborated in the introduction and were cited by both Ebert [7] and Ozkaya [5], who identified these challenges to achieve quality while managing costs and efficiency.

Nevertheless, the digital twin community still has a lot of home work. The lack of a consensual definition of digital twins leads to much room for interpretation what a digital twin is. Instead of introducing abstract approaches that are described using an attached case study, researchers should focus more on formal approaches to demonstrate and distinguish different approaches. This still may leads to many different digital twin definitions, but at least the community is able to consolidate similar approaches and has a starting point to discuss differences, flaws, or benefits of different approaches. With the introduction of virtualization tools, such as Docker, and open platforms, such as GitHub, the distribution of code and tools to replicate results of a research study or experiment with an approach became easy and has no costs attached.

REFERENCES

[1] V. Jackson, A. van der Hoek, R. Prikladnicki, and C. Ebert, "Collaboration tools for developers," *IEEE Softw.*, vol. 39, no. 2, pp. 7–15, Mar. 2022.

[2] A. Barbie and W. Hasselbring, "Embedded software development with digital twins: Specific requirements for small and medium-sized enterprises," in *Proc. IEEE Smart World Congr. (SWC)*, Aug. 2023, pp. 1–6.

[3] W. Hasselbring, S. Henning, B. Latte, A. Möbius, T. Richter, S. Schalk, and M. Wojcieszak, "Industrial DevOps," in *Proc. IEEE Int. Conf. Softw. Archit. Companion (ICSA-C)*, Mar. 2019, pp. 123–126.

[4] K. Hribernik, G. Cabri, F. Mandreoli, and G. Mentzas, "Autonomous, context-aware, adaptive digital twins—State of the art and roadmap," *Comput. Ind.*, vol. 133, Dec. 2021, Art. no. 103508.

[5] C. Ebert, "50 years of software engineering: Progress and perils," *IEEE Softw.*, vol. 35, no. 5, pp. 94–101, Sep. 2018.

[6] *Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models*, ISO/IEC Standard 25010:2011(en), International Organization for Standardization, Geneva, Switzerland, 2011. [Online]. Available: https://www.iso.org/obp/ui/en/#iso: std: iso-iec:25010: ed-1: v1: en

[7] I. Ozkaya, "Can we really achieve software quality?" *IEEE Softw.*, vol. 38, no. 3, pp. 3–6, May 2021.

[8] C. Lilienthal, *Sustainable Software Architecture: Analyze and Reduce Technical Debt*. Heidelberg, Germany: dpunkt.verlag, Jun. 2019.

[9] M. Ibrahim, A. Rassõlkin, T. Vaimann, and A. Kallaste, "Overview on digital twin for autonomous electrical vehicles propulsion drive system," *Sustainability*, vol. 14, no. 2, p. 601, Jan. 2022.

[10] P. Goupil, "AIRBUS state of the art and practices on FDI and FTC," *IFAC Proc. Volumes*, vol. 42, no. 8, pp. 564–572, 2009.

[11] V. Garousi, M. Felderer, Ç. M. Karapiçak, and U. Yilmaz, "What we know about testing embedded software," *IEEE Softw.*, vol. 35, no. 4, pp. 62–69, Jul. 2018.

[12] S. Weyer, T. Meyer, M. Ohmer, D. Gorecky, and D. Zühlke, "Future modeling and simulation of CPS-based factories: An example from the automotive industry," *IFAC-PapersOnLine*, vol. 49, no. 31, pp. 97–102, 2016.

[13] E. Bringmann and A. Kr, "Model-based testing of automotive systems," in *Proc. Int. Conf. Softw. Test., Verification, Validation*, Apr. 2008, pp. 485–493.

[14] R. Saracco and P. Henz, "Special issue on digital twins—Part 1," *IEEE Internet Comput.*, vol. 26, no. 3, pp. 5–6, May 2022.

[15] E. Negri, L. Fumagalli, and M. Macchi, "A review of the roles of digital twin in CPS-based production systems," *Proc. Manuf.*, vol. 11, pp. 939–948, Jan. 2017.

[16] F. Tao, Q. Qi, L. Wang, and A. Y. C. Nee, "Digital twins and cyber–physical systems toward smart manufacturing and Industry 4.0: Correlation and comparison," *Engineering*, vol. 5, no. 4, pp. 653–661, Aug. 2019.

[17] R. Rosen, G. von Wichert, G. Lo, and K. D. Bettenhausen, "About the importance of autonomy and digital twins for the future of manufacturing," *IFAC-PapersOnLine*, vol. 48, no. 3, pp. 567–572, 2015.

[18] W. Kritzinger, M. Karner, G. Traar, J. Henjes, and W. Sihn, "Digital twin in manufacturing: A categorical literature review and classification," *IFAC-PapersOnLine*, vol. 51, no. 11, pp. 1016–1022, 2018.

[19] T. Yue, P. Arcaini, and S. Ali, "Understanding digital twins for cyber-physical systems: A conceptual model," in *Leveraging Applications of Formal Methods, Verification and Validation: Tools and Trends*. Basel, Switzerland: Springer, 2021, pp. 54–71.

[20] F. Becker, P. Bibow, M. Dalibor, A. Gannouni, V. Hahn, C. Hopmann, M. Jarke, I. Koren, M. Kröger, J. Lipp, J. Maibaum, J. Michael, B. Rumpe, P. Sapel, N. Schäfer, G. J. Schmitz, G. Schuh, and A. Wortmann, "A conceptual model for digital shadows in industry and its application," in *Conceptual Modeling*. Basel, Switzerland: Springer, 2021, pp. 271–281.

[21] M. Grieves, "Origins of the digital twin concept," Tech. Rep., 2016. Accessed: May 26, 2024. [Online]. Available: https://www.researchgate.net/publication/307509727_Origins_of_the_Digital_Twin_Concept

[22] M. Grieves and J. Vickers, "Digital twin: Mitigating unpredictable, undesirable emergent behavior in complex systems," in *Transdisciplinary Perspectives on Complex Systems*. Basel, Switzerland: Springer, 2016, pp. 85–113.

[23] S. Newman, *Building Microservices*. Sebastopol, CA, USA: O'Reilly Media, 2021.

[24] Gartner. (2018). *Gartner Hype Cycle*. [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2018-08-20-gartner-identifies-five-emerging-technology-trends-that-will-blur-the-lines-between-human-and-machine

[25] B. Piascik, J. Vickers, D. Lowry, S. Scotti, J. Stewart, and A. Calomino, "Materials, structures, mechanical systems, and manufacturing roadmap," NASA, Washington, DC, USA, Tech. Rep., 2012, pp. 2–12.

[26] E. Glaessgen and D. Stargel, "The digital twin paradigm for future NASA and U.S. air force vehicles," in *Proc. 53rd AIAA/ASME/ASCE/AHS/ASC Struct., Struct. Dyn. Mater. Conf.*, Apr. 2012, p. 1818.

[27] National Aeronautics and Space Administration. (2021). *Perseverance Rover Multimedia*. [Online]. Available: https://mars.nasa.gov/mars2020/multimedia/videos/

[28] M. Garetti, P. Rosa, and S. Terzi, "Life cycle simulation for the design of product-service systems," *Comput. Ind.*, vol. 63, no. 4, pp. 361–369, May 2012.

[29] Plattform Industrie 4.0. *Infographic RAMI 4.0*. Accessed: May 26, 2024. [Online]. Available: https://www.plattform-i40.de/PI40/Redaktion/DE/Infografiken/referenzarchitekturmodell-4-0.html

[30] J. Trauer, S. Schweigert-Recksiek, C. Engel, K. Spreitzer, and M. Zimmermann, "What is a digital twin? Definitions ans ingisghts from an industrial case study in technical production development," in *Proc. Design Conf.*, vol. 1. Cambridge, U.K.: Cambridge Univ. Press, May 2020, pp. 757–766.

[31] *Automation Systems and Integration—Digital Twin Framework for Manufacturing—Part 2: Reference Architecture*, ISO Standard 23247-2:2021(en), International Organization for Standardization, Geneva, Switzerland, 2021. [Online]. Available: https://www.iso.org/obp/ui/en/#iso:std:iso:23247:-2:ed-1:v1:en

[32] R. Saracco, "Digital twins: Bridging physical space and cyberspace," *Computer*, vol. 52, no. 12, pp. 58–64, Dec. 2019.

[33] D. Lehner, J. Pfeiffer, E.-F. Tinsel, M. M. Strljic, S. Sint, M. Vierhauser, A. Wortmann, and M. Wimmer, "Digital twin platforms: Requirements, capabilities, and future prospects," *IEEE Softw.*, vol. 39, no. 2, pp. 53–61, Mar. 2022.

[34] Z. Lv, H. Lv, and M. Fridenfalk, "Digital twins in the marine industry," *Electronics*, vol. 12, no. 9, p. 2025, Apr. 2023.

[35] T. Lyu, U. Dwi Atmojo, and V. Vyatkin, "Towards cloud-based virtual commissioning of distributed automation applications with IEC 61499 and containerization technology," in *Proc. 47th Annu. Conf. IEEE Ind. Electron. Soc.*, Oct. 2021, pp. 1–7.

[36] E. Y. Nakagawa, P. O. Antonino, F. Schnicke, T. Kuhn, and P. Liggesmeyer, "Continuous systems and software engineering for Industry 4.0: A disruptive view," *Inf. Softw. Technol.*, vol. 135, Jul. 2021, Art. no. 106562.

[37] M. Grieves. *Digital Model, Digital Shadow, Digital Twin*. Accessed: May 26, 2024. [Online]. Available: https://www.researchgate.net/publication/369830792DigitalModelDigitalShadowDigitalTwin

[38] W. Hasselbring, "Formalization of federated schema architectural style variability," *J. Softw. Eng. Appl.*, vol. 8, no. 2, pp. 72–92, 2015.

[39] G. Smith, *The Object-Z Specification Language*, vol. 1. Luxembourg: Springer, 2012.

[40] J. M. Spivey and J. Abrial, *The Z Notation*, vol. 29. Upper Saddle River, NJ, USA: Prentice-Hall, 1992.

[41] Community Z Tools Project. *CZT: Community Z Tools*. Accessed: May 26, 2024. [Online]. Available: https://czt.sourceforge.net/parser/parser-oz/

[42] M. Sipser, *Introduction to the Theory of Computation*, 3rd ed. Belmont, CA, USA: Wadsworth, Jun. 2012.

[43] M. Grieves, *Digital Twins: Past, Present, and Future*. Basel, Switzerland: Springer, 2023.

[44] C. Leiva, "Demystifying the digital thread and digital twin concepts," *IndustryWeek*, Aug. 2016, vol. 1, pp. 1–3. Accessed: May 26, 2024. [Online]. Available: https://info.ibaset.com/hubfs/Demystifying_the_Digital_Thread_and_Digital_Twin.pdf

[45] M. Liebenberg and M. Jarke, *Information Systems Engineering With Digital Shadows: Concept and Case Studies: An Exploratory Paper*. Basel, Switzerland: Springer, 2020, pp. 70–84.

[46] P. Brauner, M. Dalibor, M. Jarke, I. Kunze, I. Koren, G. Lakemeyer, M. Liebenberg, J. Michael, J. Pennekamp, C. Quix, B. Rumpe, W. van der Aalst, K. Wehrle, A. Wortmann, and M. Ziefle, "A computer science perspective on digital transformation in production," *ACM Trans. Internet Things*, vol. 3, no. 2, pp. 1–32, Feb. 2022.

[47] P. Bibow, M. Dalibor, C. Hopmann, B. Mainz, B. Rumpe, D. Schmalzing, M. Schmitz, and A. Wortmann, *Model-Driven Development of a Digital Twin for Injection Molding*. Basel, Switzerland: Springer, 2020, pp. 85–100.

[48] M. Dalibor, N. Jansen, B. Rumpe, D. Schmalzing, L. Wachtmeister, M. Wimmer, and A. Wortmann, "A cross-domain systematic mapping study on software engineering for digital twins," *J. Syst. Softw.*, vol. 193, Nov. 2022, Art. no. 111361.

[49] J. Michael, J. Pfeiffer, B. Rumpe, and A. Wortmann, "Integration challenges for digital twin systems-of-systems," in *Proc. IEEE/ACM 10th Int. Workshop Softw. Eng. Systems-of-Systems Softw. Ecosystems (SESoS)*, May 2022, pp. 9–12.

[50] J. Michael and A. Wortmann, *Towards Development Platforms for Digital Twins: A Model-Driven Low-Code Approach*. Basel, Switzerland: Springer, 2021, pp. 333–341.

[51] G. Rieger. *Socat—Multipurpose Relay*. Accessed: May 26, 2024. [Online]. Available: http://www.dest-unreach.org/socat/

[52] M. Heithoff, A. Hellwig, J. Michael, and B. Rumpe, "Digital twins for sustainable software systems," in *Proc. IEEE/ACM 7th Int. Workshop Green Sustain. Softw. (GREENS)*, May 2023, pp. 19–23.

[53] J. I. Single, J. Schmidt, and J. Denecke, "State of research on the automation of HAZOP studies," *J. Loss Prevention Process Industries*, vol. 62, Nov. 2019, Art. no. 103952.

[54] P. Ralph, "ACM SIGSOFT empirical standards released," *ACM SIGSOFT Softw. Eng. Notes*, vol. 46, no. 1, p. 19, Jan. 2021.

[55] SunFounder. *PiCar-X—Smart Video Robot Car Kit for Raspberry Pi*. Accessed: May 26, 2024. [Online]. Available: https://www.sunfounder.com/products/picar-x

[56] A. Barbie and N. Pech. *ARCHES Digital Twin Framework*. Accessed: May 26, 2024. [Online]. Available: https://git.geomar.de/open-source/arches

[57] A. Barbie. *Digital Twin Prototypes*. Accessed: May 26, 2024. [Online]. Available: https://github.com/cau-se/DigitalTwinPrototypes

[58] A. Barbie, N. Pech, W. Hasselbring, S. Flögel, F. Wenzhöfer, M. Walter, E. Shchekinova, M. Busse, M. Türk, M. Hofbauer, and S. Sommer, "Developing an underwater network of ocean observation systems with digital twin prototypes—A field report from the Baltic Sea," *IEEE Internet Comput.*, vol. 26, no. 3, pp. 33–42, May 2022.

[59] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: An open-source robot operating system," in *Proc. ICRA Workshop Open Source Softw.*, Kobe, Japan, 2009, vol. 3, no. 3, pp. 1–5.

[60] A. Barbie, W. Hasselbring, N. Pech, S. Sommer, S. Flögel, and F. Wenzhöfer, "Prototyping autonomous robotic networks on different layers of RAMI 4.0 with digital twins," in *Proc. IEEE Int. Conf. Multisensor Fusion Integr. Intell. Syst. (MFI)*, Sep. 2020, pp. 1–6.

[61] Apache Foundation. (2009). *Apache Avro™*. [Online]. Available: https://avro.apache.org

[62] A. Barbie, W. Hasselbring, and M. Hansen, "Digital twin prototypes for supporting automated integration testing of smart farming applications," *Symmetry*, vol. 16, no. 2, p. 221, Feb. 2024.

[63] A. Nasirahmadi and O. Hensel, "Toward the next generation of digitalization in agriculture based on digital twin paradigm," *Sensors*, vol. 22, no. 2, p. 498, Jan. 2022.

[64] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Sep. 2004, pp. 2149–2154.

[65] S. Nativi, P. Mazzetti, and M. Craglia, "Digital ecosystems for developing digital twins of the Earth: The destination Earth case," *Remote Sens.*, vol. 13, no. 11, p. 2119, May 2021.

[66] W. Shengli, "Is human digital twin possible?" *Comput. Methods Programs Biomed. Update*, vol. 1, Jan. 2021, Art. no. 100014.

[67] D. Helbing and J. A. Sánchez-Vaquerizo, *Digital Twins: Potentials, Ethical Issues and Limitations*. U.K.: Edward Elgar Publishing, Jun. 2023, pp. 64–104.

[68] D. de Kerckhove, "The personal digital twin, ethical considerations," *Phil. Trans. Roy. Soc. A, Math., Phys. Eng. Sci.*, vol. 379, no. 2207, Aug. 2021, Art. no. 20200367.

ALEXANDER BARBIE was a Software Engineer with the GEOMAR Helmholtz Centre for Ocean Research Kiel, Germany, and the Alfred Wegener Institute Helmholtz Centre for Polar and Marine Research, Bremerhaven, Germany. He is currently a Software Architect in industry and a Ph.D. Researcher with the Software Engineering Group, Department of Computer Science, Kiel University, Germany.

WILHELM HASSELBRING is currently a Full Professor of software engineering with the Department of Computer Science, Kiel University, Germany. In the competence cluster Software Systems Engineering (KoSSE), he coordinates technology transfer projects with industry. In the excellence cluster Future Ocean, Helmholtz Research School Ocean System Science and Technology (HOSST), and Helmholtz School for Marine Data Science (MarDATA), he collaborates with the GEOMAR Helmholtz Centre for Ocean Research Kiel.