A Method for Model-Driven Engineering of Digital Twins in Manufacturing

Malte Heithoff*, Judith Michael *†, Bernhard Rumpe *, Jérôme Pfeiffer ‡, Andreas Wortmann ‡, Jingxi Zhang ‡
* *Software Engineering*, *RWTH Aachen University*, Aachen, Germany
Email: heithoff@se-rwth.de, michael@se-rwth.de, rumpe@se-rwth.de
† *University of Regensburg*, Regensburg, Germany, Email: judith.michael@ur.de
† *Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW)*,
University of Stuttgart, Stuttgart, Germany
Email: jerome.pfeiffer@isw.uni-stuttgart.de, wortmann@isw.uni-stuttgart.de, jingxi.zhang@isw.uni-stuttgart.de

***Abstract*—Several software architectures for digital twins have been proposed, specifying their structure and key components, such as the ISO 23247 standard for manufacturing. However, systematic methodologies for developing digital twins in the manufacturing domain remain lacking. Existing research does not adequately address methodological aspects, such as composing essential digital twin components, defining their interfaces, and systematically analyzing relevant data. This gap presents a challenge for the structured development of digital twins. In this work, we introduce a systematic approach to identify key requirements for manufacturing digital twins and propose a model-driven method for their creation. Our approach enables the generation of executable digital twins based on a formalized specification. We demonstrate its applicability through multiple industrial manufacturing demonstrators, highlighting its suitability in practice. The proposed method is adaptable regarding the purpose of the digital twins to be developed and supports a largely automated process.**

***Index Terms*—Engineering Method, Model-Driven Engineering, Digital Twin, Software Architecture, Manufacturing**

I. Introduction

While digital twins (DTs) are widely used in various application domains, *i.e.,* automotive [1], [2], [3], smart cities [4], energy systems [5], [6], or avionics [7], [8], they have become a vital pillar of digitalization in manufacturing [9]. Consequently, a plethora of publications on various kinds of DTs for manufacturing have been published in the last decade.

According to two very popular characterizations of DTs by Kritzinger [10] and Tao [11], a DT must be a complex software system that interacts with its (cyber-)physical counterpart for a variety of services (such as behavior optimization or predictive maintenance). And while many software architectures of such DTs have been proposed [9], systematic methods to engineer DTs according to these architectures are lacking. Most notably, the ISO 23247 standard [12] sketches the structure of a software architecture for DTs based on "functional entities" but does not disclose how they are meant to be composed, interface with another, or should be constructed. Furthermore, in practice, small and medium-sized enterprise (SMEs) oftentimes do not know how to begin with DTs or cannot afford the software engineering expertise necessary to implement DTs. We propose to close this gap by presenting a logical software architecture for DTs that complies with ISO 23247 and comes with a systematic, purpose-driven method to produce software artifacts realizing this architecture. To this end, we leverage integrated modeling languages for data structures, UI elements, and software components and combine two existing code generation toolchains to produce the different parts of the DTs. Hence, this paper contributes (1) a systematic method to identify the requirements for creating DTs according to our architecture, and (2) a pervasive model-driven method using model transformation and code generation to produce executable DTs based on these requirements.

In the remainder, Sec. II introduces our notion of DTs and the modeling infrastructures used for generating DT software components. Sec. III presents related work before Sec. IV introduces our base architecture for ISO 23247 compliant DTs. Sec. V outlines our method for identifying the requirements necessary to instantiate this architecture using the toolchain presented in Sec. VI. Sec. VII presents the method application in an industry, followed by a discussion of the approach's assumptions, strengths, and limitations. Sec. IX concludes.

II. Background

Digital Twins: Research and industry employ DTs (DTs) [10], [13] to better understand and use cyber-physical, biological, and social systems [9], [14]. The DTs promise to reduce development costs and time, improve operations, and deepen our understanding of the represented Actual Systems (ASs) [9]. Therefore, the DTs serve different purposes, such as analysis [15], control [16], or behavior prediction [17].

While research and practice have produced various reference models of DTs [18], there is still little consensus on what a DT actually is and what its implementation should comprise. Popular definitions either define DT based on the data flows between the DT and the represented AS [10], coarsely describe abstract modules that they may comprise [13], [12], or focus on software architectures of DTs for very specific DT applications. One thing that all DTs according to [10] have in common is that they obtain data from the AS, process this, and may use insights gained from this processing to manipulate that system [10]. However, there are many more expectations on what a DT should be able to do as outlined by the functional

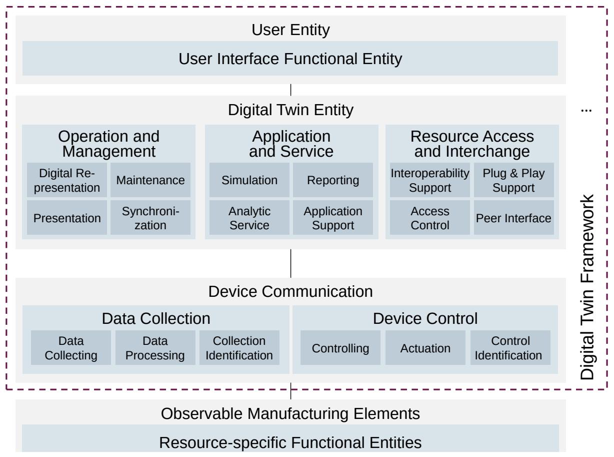


Fig. 1: Functions of DTs according to ISO 23247 [12].

entities[1] mentioned in ISO 23247 [12] (*cf.* Fig. 1): Here, DTs collect data from observable manufacturing elements, process the data and provide it to added-value functions, *e.g.,* for digital representation, analytics, reporting, or simulation. Their results are then sent back to control the AS.

Digital Shadows: Digital Shadows (DSs) are an important concept for data use and sharing [19]: They are *"a set of contextual data traces and their aggregation and abstraction collected for a specific purpose with respect to an original system"* [20] supporting the reduction to what is of importance for a particular data usage purpose, i.e., in DTs. The DSs are tailored to specific tasks and contexts by using data aggregation methods [21]. Becker et al. [20] and Michael et al. [22] describe relevant DSs concepts, including the purpose, data trace and data points, their metadata, the related system the data is coming from, and models providing essential contextualization to the data. One can consider different relationship variants between datatraces and models, dependent on the kinds of models [23], i.e., simulation, SysML, 3D models, the lifecycle phase [24], and the application domain. [22] provides additional model usages, e.g., data calculation models using data traces as input or creating them as output, or simulation models related to the data traces they are consuming and producing. In addition, one can use workflow models to describe how more complex processing steps consuming and producing data traces are chained [25]. Within [22], we also describe a method for creating DSs. This method is applied in [26] to create DSs at different levels of the automation pyramid for injection molding.

MontiArc: MontiArc [27] is an extensible component & connector architecture description language (ADL). It features core elements of component & connector ADLs, *i.e.,* hierarchically decomposable components with interfaces, types, connectors, and architectural configurations. MontiArc's semantics is based on the Focus theory of stream-processing functions [28]. Using the MontiCore [29] language workbench, the logical software architectures described with MontiArc are verified and translated into executable architecture implementations in Java, Mona, Python, and more [27]. MontiArc's components either comprise a topology of subcomponents, a single behavior model describing their input-output behavior, or neither. In the latter case, they expect a handcrafted behavior implementation to be provided following certain naming conventions, such that the generated component interface can link to it. Ultimately, each component model becomes a set of classes representing the component's interface, inputs and outputs, and its behavior implementation. We model the logical architecture of the DT with MontiArc and translate this into an executable Kotlin implementation using code generators.

MontiGem: MontiGem [30] is a generator for creating web-based applications. It builds on the functionality of MontiCore to take Class Diagrams (CDs) describing the data structure, Object Constraint Language (OCL) models to define data constraints, and Graphical User Interface (GUI) DSL models [31] to describe graphical user interfaces and are transformed into a client-server application. We refer to the DT user interfaces as DT cockpits [32]. The generated application can be extended with hand-written code. The resulting generated application consists of a database, a backend that interacts with the database, and a frontend that interacts with the backend. The frontend is an Angular web application that supports various customizable GUI widgets, such as adaptable tables and several chart components. MontiGem has been successfully applied in domains such as financial management [33], IoT systems [34], or DTs [32], [35].

III. Related Work

A. Systematic Engineering of Digital Twins

Various methods for the systematic engineering of DTs have been explored in the literature. The engineering of DTs can be guided by reference architectures and ontologies. One approach proposes an architecture for DT systems based on ISO 23247 [36]. An approach introducing a CI/CD pipeline for DTs in the cloud decouples DTs into micro-service. However, the steps required to instantiate these architectures for different use cases are not provided. Another approach proposes a service-oriented engineering workflow, guiding users through service selection and recommending relevant models derived from ontologies, followed by automated execution and deployment [37]. In contrast to our study, however, they do not define the dependencies between the different services and models. One study focuses on how the DT can be leveraged in different lifecycle phases of IoT [38]. However, it does not outline the engineering of the underlying DT, that enables the integration of IoT devices.

B. Requirements Engineering for Digital Twins

Recent work has begun to address requirements engineering for DTs, emphasising linking requirements to structured models and behaviors. Zhao et al. [39] proposes an approach for engineering IT/OT requirements within Asset Administration Shells to enable their traceability across development stages. However, a consideration for the derivation of DTs and a

[1]The standard does not require all of them to be present in at DT.

mapping of the requirements to software or modeling artifacts is missing. In the realm of human-centric requirements, studies focusing on non-functional requirements such as transparency, user competence, and safety [40], [41] are conducted. However, these works primarily frame these qualities as abstract system goals, without offering engineering guidelines, *i.e.,* without mapping requirements with available information of the physical system. Gar et al. [42] translates system requirements into behavior models, which are used to derive PLC control logic for virtual commissioning but lack the structural information or requirements for other services of the DT. Additional work addresses adaptability and lifecycle considerations. De Almeida et al. [43] focus on data integration as a foundational capability for DT requirements, while Kamburjan et al. [44] propose mechanisms for dynamic adaptation based on evolving requirements. Together, these studies emphasize the need for formalized models of both functional and non-functional requirements.

C. Model-Driven Development of Digital Twins

Some studies focus on engineering DTs based on data models. From these models, they can generate code or explore the modeling of sensors and data flows [45]. Another data-centric study leverages medical data to recommend treatments [46]. Parbat et al. [47] focus on formalizing the behavior of DTs with mathematical models while Nguyem et al. [48] focus on modeling data structures, integrating network protocols, and mapping Asset Administration Shell (AAS) submodels to physical and simulated assets. Another area of research applies AI and optimization techniques within model-based contexts [49]. A case study [50] applies DTs to agriculture, where model-based what-if analyses are used to simulate the effects of environmental and operational variables. Other works [51] integrate semantic models and a knowledge graph to represent the knowledge about a milling machine in the manufacturing domain. Despite defining model-driven methods for deriving DTs, the existing studies lack a systematic method of which models need to be created at which point in time, together with the model dependencies and the generation or transformation steps required to derive a functioning application-specific DT. A systematic mapping study [52] investigating MDE techniques in the context of DTs identifies data models as the most common models. These models are usually leveraged to generate data processing, data storage, and graphical user interfaces. This also aligns with our rather data-centric perspective on DTs, where we leverage data models to generate the different components of our DT architecture.

IV. Digital Twin Architecture

The base architecture of our DT is an extension of the architectures presented in [53], [20] that incorporates elements of the MontiGem [34] toolchain to include DT cockpit generation and omits a fixed MAPE-K loop [53] in favor of extensibility with services. As such, the core functionality of this architecture is to synchronize the information relevant to the DT purpose with the AS. Hence, the most vital components of our DTs are:

`Gateway`: Acts as the connection between the DT and its AS. As the ports of the `Gateway` describe the information requirements of the DT, the main duty of the `Gateway` is fulfilling these requirements by providing this information, either directly from the AS, by performing computations on data from the AS, or by taking into account other data sources in addition.

`Shadow Caster`: Takes input data from the `Gateway` and produces the digital shadows required by the services (incl. the cockpit). Hence, whenever data belonging to one or more digital shadows changes, the `Shadow Caster` takes care of constructing these shadows by fetching their other data as well and provides the shadows to the other components of the DT.

`Synchronizer`: Synchronizes the data between the DT and the AS. It receives every DS that is sent to the DT and receives DSs from other components to ensure that every DS is synchronized between `Gateway` and `Database`.

`Executer`: Acts as the inverse of the `Shadow Caster` by taking as input DSs meant to be send to the `Gateway`, *i.e.,* to change the behavior of the AS, and translates these to messages to the `Gateway`'s incoming ports.

`Controller`: Comprises a BPMN interpreter and models that describe the normal process of handling services and that listen to events [32], such as a new DS being available, that are relevant to other components. As of now, the BPMN models are handcrafted.

`Service Manager`: acts as the wrapper of the services of the DT and ensures that the concrete services are independent of the `Engine` and its interface. Hence, it forwards each incoming DS to each service and uses the `Service Merger` to collect the outputs of all services prior to forwarding them to the `Engine`. The interface of the `Service Merger` consequently is synthesized based on the services of the DT.

`Database Manager`: is responsible for abstracting SQL queries (depending on the used technology) to the database away and automatically disassembling DSs into their individual properties. Its interface is according to the *maximal data structure* (see Sec. VI).

`Database`: the *maximal data structure* specifies the database, which is capable of storing each defined property. Here, we decided on a relational SQL database, but it may be any other database technology if it seems suitable.

`DT Cockpit`: is the GUI provided by MontiGem and behaves like a service by requiring and providing DSs from/to the DT. User interactions with other services are communicated via DSs.

As the ports of `Gateway` and `Shadow Caster` depend on the data required by the DT's services, both components need to be provided specifically for a concrete DT. While the `Shadow Caster` is synthesized based on the required data, for the `Gateway`, only the interface can be synthesized, but its behavior must be implemented manually as of now. Likewise, the `Shadow Merger` depends on the inputs from the services

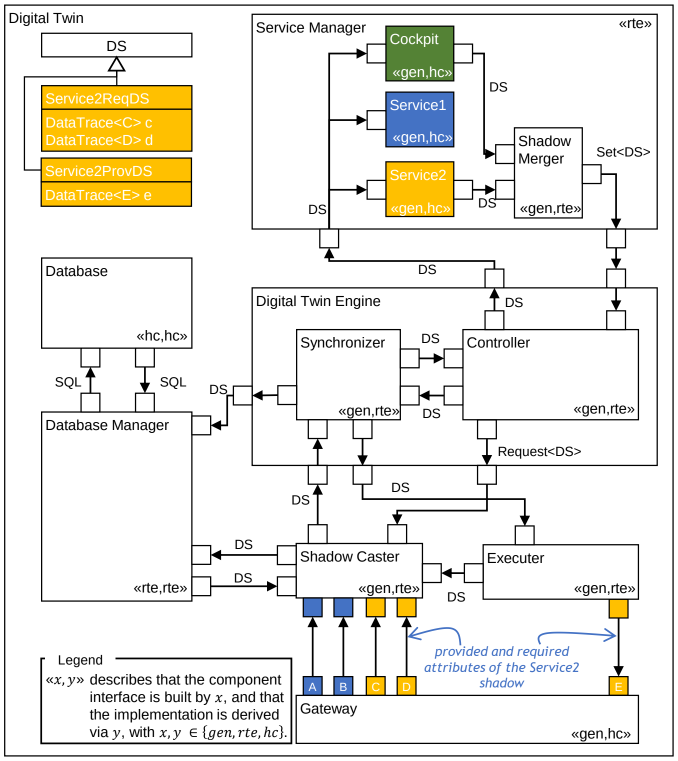


Fig. 2: Base architecture for our model-driven DTs

of the DT and is synthesized. The other components of the DT are generic.

Fig. 2 illustrates the base architecture's components and their relations but includes three services for better comprehensibility of how service interfaces govern ports of the gateway. Out of these services, only the DT cockpit is part of the base architecture. Also note that we consider the Java classes generated from the architecture and from the data types (*e.g.*, the digital shadow base type realizations) as part of the run-time environment (RTE) of the DT.

A. Implementation of ISO 23247

Our base architecture implements the essential functional entities of ISO 23247 with its core components as illustrated in Table I. The ISO standard does not prescribe which of the functional entities are mandatory. However, certain minimal functionalities follow from the definitions of DTs discussed above, while others (such as proper visualization) follow from pragmatic requirements. So, we opted to mandate the existence of components representing the functional entities for presentation, synchronization, digital representation, and device communication. The use of other functional entities depends on the purpose of the DT, and we decided these to be optionally realizable as services in our architecture.

B. Architecture Extension

Our base architecture requires certain extensions to become operational. To this end, it leverages the component-based approach on MontiArc, which supports the introduction of

TABLE I: Implementation of ISO 23247 (*cf.* Fig. 1) by our DT architecture (*cf.* Fig. 2).

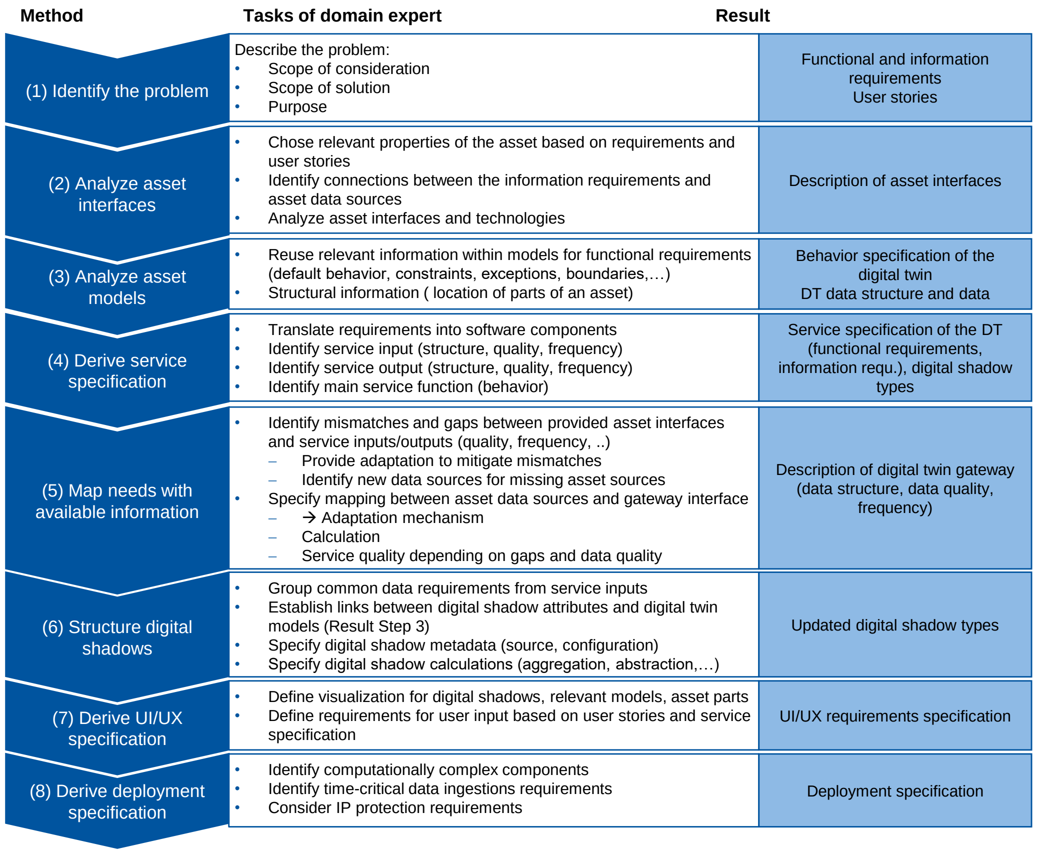
ISO 23247 Concept	**DT Component**
Device Communication	Gateway
Digital Representation	Database Manager
Synchronization	Synchronizer
Presentation	Cockpit
Maintenance, Simulation, Analytics, *etc.*	Services

hierarchically decomposed subcomponents without needing to change the interface of the containing component. Using this approach, new (potentially automatically synthesized) subcomponents are integrated into the DT architecture model prior to generating the component implementations with it. These new subcomponents may be hierarchically decomposed, feature behavior models, or behavior implementations themselves and, hence, might be of any complexity. We leverage this for integrating new services, provided as MontiArc components, into the architecture model and check their well-formedness, before producing the architecture's code. Of course, this method also can be applied to add other new components into the architecture directly (*e.g.*,, additional logging or reporting functionality) without relying on services.

V. Identifying Requirements for Digital Twins

Building DTs is a complex, time-intensive task, as the domain expert needs to deliver a plethora of information that is relevant, such that the software engineer can realize the DT in software artifacts. Based on our experience with building various DTs in practice (*cf.* Sec. VII), we have developed a method to enable domain experts to describe DTs from their perspective, yet independent from the actual implementation, to the software engineer. The following eight-step method can be applied by domain experts, e.g., product designers, factory planners, or production planners and controllers (*cf.* Fig. 3).

1) **Identify the problem.** In the first step, the problem at hand is identified. This problem is described using the scope of consideration, the possible solution scope, and the goal of achieving the solution to the problem, which is described by the purpose. The typical results of this step are functional requirements and information requirements, as well as user stories that describe the scope and purpose of the problem. These requirements serve as a basis for the identification of required data from the AS or from other systems (e.g., a manufacturing execution system or a product lifecycle management system) that the services of the DT will need as well as provide to the DT.
2) **Analyze asset interfaces.** For solving the problems from the previous phase, each solution requires one or more assets to deliver the necessary data. Therefore, the user stories and requirements from the previous step are used to identify relevant properties of the asset that can be used to directly provide or synthesize the required data. Furthermore, potential dependencies between the



Method	Tasks of domain expert	Result
(1) Identify the problem	Describe the problem: • Scope of consideration • Scope of solution • Purpose	Functional and information requirements User stories
(2) Analyze asset interfaces	• Chose relevant properties of the asset based on requirements and user stories • Identify connections between the information requirements and asset data sources • Analyze asset interfaces and technologies	Description of asset interfaces
(3) Analyze asset models	• Reuse relevant information within models for functional requirements (default behavior, constraints, exceptions, boundaries,…) • Structural information (location of parts of an asset)	Behavior specification of the digital twin DT data structure and data
(4) Derive service specification	• Translate requirements into software components • Identify service input (structure, quality, frequency) • Identify service output (structure, quality, frequency) • Identify main service function (behavior)	Service specification of the DT (functional requirements, information requ.), digital shadow types
(5) Map needs with available information	• Identify mismatches and gaps between provided asset interfaces and service inputs/outputs (quality, frequency, ..) – Provide adaptation to mitigate mismatches – Identify new data sources for missing asset sources • Specify mapping between asset data sources and gateway interface – → Adaptation mechanism – Calculation – Service quality depending on gaps and data quality	Description of digital twin gateway (data structure, data quality, frequency)
(6) Structure digital shadows	• Group common data requirements from service inputs • Establish links between digital shadow attributes and digital twin models (Result Step 3) • Specify digital shadow metadata (source, configuration) • Specify digital shadow calculations (aggregation, abstraction,…)	Updated digital shadow types
(7) Derive UI/UX specification	• Define visualization for digital shadows, relevant models, asset parts • Define requirements for user input based on user stories and service specification	UI/UX requirements specification
(8) Derive deployment specification	• Identify computationally complex components • Identify time-critical data ingestions requirements • Consider IP protection requirements	Deployment specification

Fig. 3: The method to design DTs as domain expert from initial problem identification until the deployment of the DT.

information requirements imposed by the DT and the data that can be provisioned by the asset data sources are identified. For instance, one information requirement might be fulfilled by multiple asset data sources. Besides, how the data sources can be utilized, *i.e.*,, the technology and description of the asset interfaces, are relevant for realizing the DT. Otherwise, the DT will be unable to connect to the asset to gather data. Ultimately, the result of this step is a description of the asset interface and how it can be used to provide the required data to the DT.

3) **Analyze asset models.** The services of the DT might require information about the AS its interface does not provide, but by models of the AS, such as SysML, Function Blocks, AutomationML, or CAD models. Hence, in this step, they are analyzed to identify the relevant information for the behavior and constraints of the DT. Furthermore, structural information is relevant for the DT, *e.g.,* to reason about which parts of the asset are impacted by predictive maintenance tasks, to give context to data in general, or for visualization. Together with the results of Step 2, the results are a behavior specification and a description of the data structures and data.
4) **Derive service specification.** To fulfill the functional requirements formulated in Step 1, these need to be translated and implemented into software components. In this step, we define such components based on their inputs, outputs, and functionality. For the inputs and outputs, we define the structure, quality, and frequency in which the data should be received or sent by the respective component. Results from this step are (1) the service specifications with their inputs, outputs, and behavior and (2) the DS [22] types that are defined based on the collected inputs and outputs of all service components.
5) **Map needs with available information.** After specifying the services, there may be potential mismatches between the information the services require (*cf.* Step 1) and the information the asset interface and its environment can provide (*cf.* Step 2). This may be, *e.g.,* due to different data types or a computation step that is required to combine multiple information provided by the asset. Mismatches can be resolved by adapting or converting the information, or by connecting another data source that provides the required data. In this step, the DT gateway is defined that is able to acquire the necessary data and provide it to the DT services in the right format, *i.e.,* conform to the data structure, data quality, and frequency requirements imposed by the service interface.
6) **Refine DSs.** We define the DS types[22] based on the service specification (*cf.* Step 4). Afterward, we map the asset interface information to the information required by the services; we might need to adapt these DS types based

on the mappings from Step 5. This includes DS metadata and calculations. Consequently, the result of this step is an updated DS type definition.

7) **Derive UI/UX specification.** We define a DT as a software system that can receive data from an asset, as well as actively interact with the asset, *i.e.,* control it. Consequently, the DT should present asset information to the domain expert and enable user input. Therefore, based on the user stories from Step 1, we define the visualization of the DSs and potentially relevant models. We also define the user input, *i.e.,* the interaction concept of the domain expert with the DT. The result of this step is a specification of the UI and UX requirements.
8) **Derive deployment specification.** This last step is concerned with the deployment of the DT. For this, it is relevant to know which components of the DT are computationally complex and need to be deployed, *e.g.,* to the cloud to scale computation resources dynamically, or that are time critical and need to be executed at a server at the shop floor, or on edge computing resources. Security may also impose requirements for the networking of the DT. All these requirements should be collected in a deployment specification.

VI. Generating Digital Twins

Our method to generate DTs relies on (1) the requirements and information gathered as outlined by the eight steps above, (2) encoding this information into models used as input for our DT code generation toolchain, and (3) combining the resulting artifacts such that the components of the DT are produced. In general, the generation consists of two steps: In the first step, all information specified by the domain expert is formalized into class diagrams, UI models, and MontiArc models. In the second step, these models are combined and enriched before they are translated into an executable DT architecture. Fig. 4 depicts the overview of all involved modeling languages, models, and software components of our method for generating DTs. Overall, the method consists of four sequences of main activities:

1) **Define data models and UI models** in the form of UML/P class diagrams and GuiDSL models that are based on service requirements (*cf.* Step 3-4), results in the DSs to be used in the DT (*cf.* Step 6), and visualization requirements (*cf.* Step 7). These data models are leveraged by the MontiGem toolchain to produce the database infrastructure to persist the data obtained from and about the DT as well as to produce the cockpit visualizing the DT and the data transport objects, *etc.* required for this. To this end, MontiGem leverages various model transformations and code generators [30], [34].
2) **Derive the components** of the DT that are specific to the services in the form of MontiArc component models and provide implementations for them. This includes components for services (*cf.* Step 4) as well as for the gateway (*cf.* Steps 2, Step 5) to the AS.
3) **Integrate the derived components** into the DT base architecture using the mechanisms of MontiArc [27] to create an application-specific variant of this architecture.
4) **Generate the architecture implementation** using the MontiArc code generation toolchain.

The remainder of this section elaborates on each of these activities in detail and links them to the requirements elicitation method presented above.

A. Data Modeling for Services, UI, and Asset Representation

In our purpose-driven approach to engineering DTs, the first artifacts to be produced are data models that capture the data structures (a) *required and provided by the services* that realize the functional requirements identified in Step 1, (b) the *static asset information* required about the asset and its parts, and (c) the *data structures required for visualization* of the services and of related DT data. Using the *Service shadow transformer*, the data models required and provided by the services are augmented to become the *required service shadows* and *provided service shadows*, respectively. The essential transformations conducted include that each data structure required or provided by a service becomes a subtype of the base DS data type provided by the DT run-time environment and that each attribute is augmented with metadata regarding the time of last change, owner, source, etc. as outlined in our conceptual model of DSs [20]. Using the *service shadows* computed this way, the data models for the UI and the static asset information, the *class diagram merger* produces a single class diagram holding the maximal data structure required to be captured by the DT to fulfill its purposes as described by the UI and services. With this in place, the *MontiGem generator* takes this maximal data structure and dedicated UI models (*cf.* [31]) as input from which it produces a dashboard, a database backend, and data transport objects for the communication between cockpit and backend for the DT [54], [30]. This yields a web-based UI able to represent all data required and provided by the services of the DT, the static asset information and UI specific elements.

B. Deriving Application-Specific Components

From the *Service shadows* derived above and component behavior implementations for the services, the *DT component model synthesizer* produces novel MontiArc components for each service. Additionally, it synthesizes the *gateway* component, such that it features one outgoing port for each attribute of a *required service shadow* and one incoming port for each attribute of a *provided service shadow*, *i.e.,* it decomposes the DSs into attributes for which the values can be provided individually by the implementation of the *gateway*. Based on the set of attributes derived from the DSs, the synthesizer then produces a novel, application-specific gateway component model, in which each attribute of a required shadow becomes an outgoing port and each attribute of a provided shadow becomes an incoming port. For this component model, an implementation that adapts between the incoming and outgoing ports of the gateway and the communication interface of the AS needs to be created manually according to MontiArc's

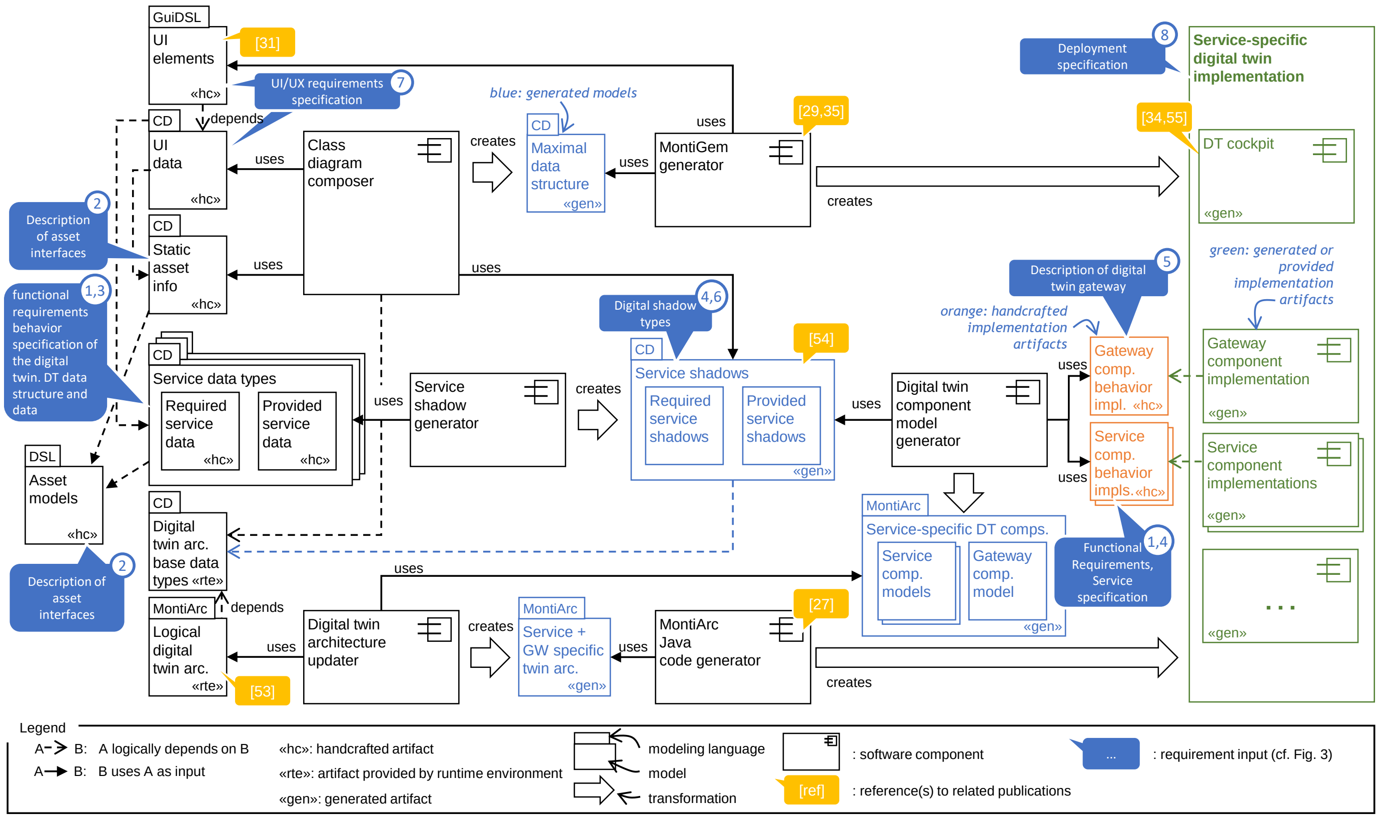

GuiDSL
UI elements «hc»
[31]
UI/UX requirements specification 7
blue: generated models
CD
UI data «hc»
depends
uses
Class diagram composer
creates
CD
Maximal data structure «gen»
uses
MontiGem generator
uses
[29,35]
Deployment specification 8
Service-specific digital twin implementation
[34,55]
creates
DT cockpit «gen»
Description of asset interfaces 2
CD
Static asset info «hc»
uses
uses
green: generated or provided implementation artifacts
functional requirements behavior specification of the digital twin. DT data structure and data 1,3
Digital shadow types 4,6
Description of digital twin gateway 5
orange: handcrafted implementation artifacts
CD
Service data types
Required service data «hc»
Provided service data «hc»
uses
Service shadow generator
creates
CD
Service shadows
Required service shadows
Provided service shadows
«gen»
[54]
uses
Digital twin component model generator
uses
Gateway comp. behavior impl. «hc»
Gateway component implementation «gen»
uses
Service comp. behavior impls. «hc»
Service component implementations «gen»
DSL
Asset models «hc»
Description of asset interfaces 2
CD
Digital twin arc. base data types «rte»
MontiArc
Service-specific DT comps.
Service comp. models
Gateway comp. model
«gen»
Functional Requirements, Service specification 1,4
...
«gen»
uses
MontiArc
Logical digital twin arc. «rte»
depends
uses
[53]
Digital twin architecture updater
creates
MontiArc
Service + GW specific twin arc. «gen»
uses
MontiArc Java code generator
[27]
creates
Legend
A ⇢ B: A logically depends on B
A → B: B uses A as input
«hc»: handcrafted artifact
«rte»: artifact provided by runtime environment
«gen»: generated artifact
modeling language
model
transformation
: software component
[ref] : reference(s) to related publications
... : requirement input (cf. Fig. 3)

Fig. 4: The models and generators involved in our model-driven engineering method of generating DT architectures.

patterns for integrating handcrafted code. As such, a gateway implementation must adapt between the ports of its interface and an arbitrary communication interface of the AS, fully generating the gateway is not feasible yet However, by restricting the gateway to communicate with a fixed kind of technology (such as OPC UA), this mapping could be partly automated.

C. Deriving the Application-Specific Digital Twin Architecture

Using as input the DT base architecture and the novel gateway component model, the *Digital twin architecture updater* replaces the base gateway with the specific gateway and adjusts the interface of the *Shadowcaster* component to receive all outgoing ports of the synthesized gateway. The *Shadowcaster* triggers the production of the corresponding DSs whenever an attribute being part of a shadow changes, *i.e.,* its implementation is generated accordingly based on the DSs as well. Due to the service shadows being subclasses of the DS base class, they can be communicated via the connectors typed with *DS* (*cf.* Fig. 2) and the other components of the architecture can process these accordingly. This, for instance, entails sending the produced DSs to the *Synchronizer* component to store them in the database as outlined above.

D. Generating the Digital Twin Implementation

Given the application-specific DT architecture, the code generator provided by MontiArc translates it and all of its subcomponents into Java implementations of the components as outlined in Sec. II. To this end, the generator takes into account the provided component implementations of the services and of the gateway. A resulting DT architecture for three services is presented in Sec. IV.

VII. Industrial Application

For the evaluation, we employed two advanced PhD students with mechatronics backgrounds from the ISW, who regularly conduct Industry 4.0 projects, and, hence, know the field partly. As such, they are as much domain experts as entry-level employees in manufacturing small and medium-sized enterprises (SMEs). We evaluated our method in an industrial case study for predictive maintenance on a FiveX drilling machine at the ISW[2]. The FiveX Milling Machine possesses five degrees of freedom and a tool changer with multiple milling tools for different materials. The machine's design comprises two physical systems that interact with each other, enabling flexible adaptation to diverse requirements. The DT of the FiveX monitors the tool exchange based on a wear index. For evaluation, we used the physical FiveX together with co-simulation. The tool exchange itself was simulated only and not performed on the physical machine. In the following we describe the steps of our method for model-driven engineering of DTs in manufacturing (see Fig. 3) instantiated in our FiveX case study.

[2]https://www.isw.uni-stuttgart.de/forschung/projekte/Stuttgarter-Maschinenfabrik/, Fig. 2

Problem Identification: In this case study, we identify three problems, denoted as physical machine requirements. The first is the ability to control multiple axes individually through a dedicated interface (see Fig. 5). For this, the FiveX machine provides machine interfaces, position, and rotation to an OPC UA [55] server [51]. The OPC UA server also stores information such as the name, description, manufacturer, together with the current machine status, containing the current vibration, spinning speed, and the applied current of the equipped tool. The tool wear is a value to be calculated by the DT, as the FiveX machine does not have predictive functionality itself. The second problem is the command execution from the OPC UA server. The OPC UA server directly calls the methods provided by the FiveX machine interface to change the tool, to power the equipped tool, or to move an axis by a certain distance. However, there are various methods as well as technical differences in the interfaces of each tool. This is also the case for the axes, as they are from different manufacturers, but also as two axes describe a rotation and three axes describe a translation. The third problem is self-adaptive tool change, where the machine is provided with spare tools when a tool is nearing the end of its life. The tool change must be triggered by OPC UA commands sent from a prediction service provided by the DT. Each tool has a unique name and is stored in the tool changer (see Fig. 5), which provides these tools concerning their rotation. The tool changer also contains a list of the available tools. Since the tool changer is an independent system, it has a unique name for its addressing. It is possible to equip tools for different purposes. In this case study, we show milling tools, but functionality such as 3D printing or pick-and-place operations are also possible. The tool changer contains a rotary axis for tool selection and a mechanism for retrieving and providing tools. For the execution of process models, a series of milling operations on wood are conducted and recorded by the sensors. Consequently, this results in a time series of position, spindle speed, current, and vibrations. The material properties of wood, including its density, contribute to the estimation of the tool's health status.

Asset models: In this step, we look at available asset models that are created during requirement analysis. One asset model is a class diagram of the FiveX milling machine. It describes that this machine (see Fig. 8 in a simulation) contains interfaces for connecting industrial computers, an OPC UA server, and for integration into a production line scenario. Besides, the machine comprises two simulation models deployed on a simulation computer (see the left half of Fig. 8): one for the five-axis machine and one for the machine tool changer. Each of these models comprises a 3D representation of the machine along with a state chart. These models show the possible states the machine can reach. For the sake of simplicity, we focus on the following states: The initial state, which signifies that the machine is prepared for operation. The second state is "error," which occurs when the tool is unknown, the movement of an axis collides with other parts of the machine, or the current on the inputs contains an illegal

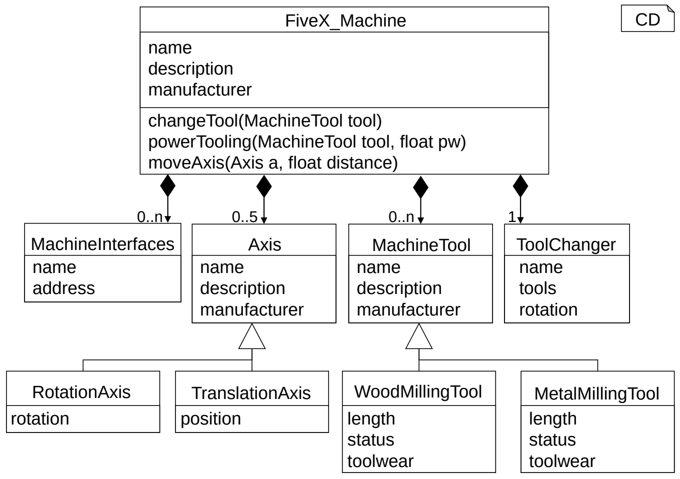


Fig. 5: Asset model of simulated FiveX milling machine.

state. These states can be further delineated as follows: Errors can be generated by signals on the inputs that result in an illegal state, such as a command to move both forward and backward at the same time. Finally, the operation state is where the machine is moving or milling without any problems. The machine has a nameplate containing a serial number, its location, a unique identifier, and a bill of materials, which includes detailed information on each machine part of each manufacturer. Additionally, the sensors utilized by the machine are documented. These sensors are capable of acquiring data regarding the machine's current status, such as its rotational speed and the pressure to which the spindle is subjected during operation. Additionally, sensors are employed to ensure the precise positioning of each axis. Lastly, we employ an NC model, which prescribes the movements and spindle speed during a milling process operation.

Asset interfaces: In this step we analyze the interfaces of the machine to identify the data sources and technologies. The asset provides its tool status and axis positions and commands via OPC UA. As we are using a simulation at the hardware level, the requirements and asset models for the simulation and the physical machine are the same. The interfaces need to be addressed with an OPC UA client implementation (see Fig. 6). This serves as the DT gateway. As the OPC UA server only displays the most recent values, the asset must be connected to a database. In our case, we use a time series database (see Fig. 6) to store historical data. For the details of the connection to the OPC UA server, the values are stored in a tree representation that contains objects with variable nodes in which the concrete value is stored. The object node also contains a method node to manipulate the position, rotation, or current of a tool or axis.

Additional Services: The additional analysis and planning services identified during the derivation of service specifications fulfill the third requirement that is a self-adaptive tool wear prediction service. The service should analyze the state of the input from the motor (see Fig. 6). This analysis receives the DS and analyzes the tool vibration, the applied current, and

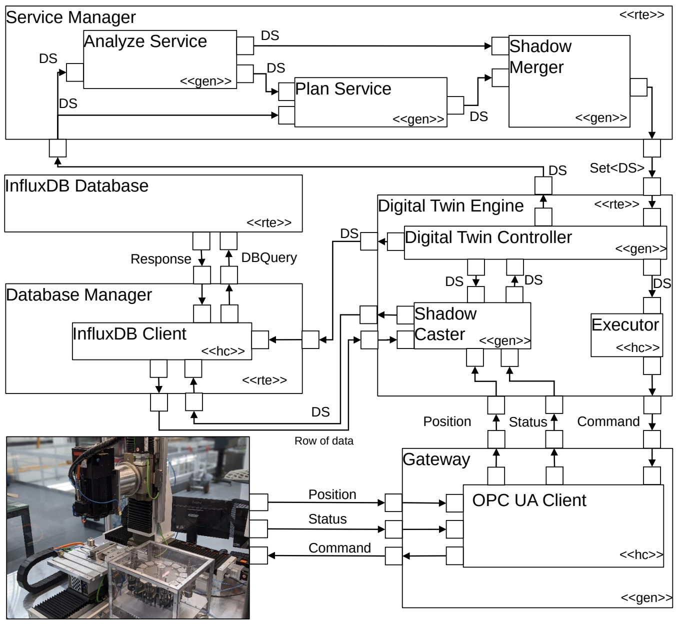


Fig. 6: Result of application of methodology on FiveX milling machine. A service providing further functionality is the self-adapted tool change consisting of Analyze and Plan Services.

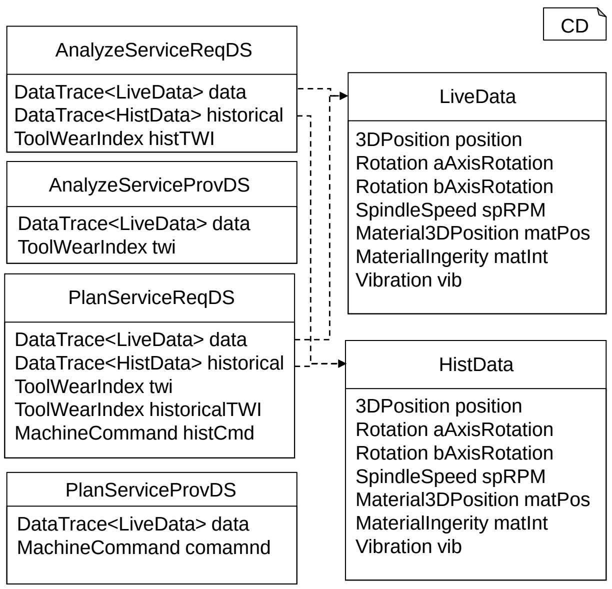


Fig. 7: FiveX DS after consideration of the required DS of the services. This also includes the outcoming DS of the services.

the tool position as well as the integrity of the material to infer the tool wear. The knowledge is retrieved from the database and transformed into a DS. The DS then is sent to the shadow merge and the planning service. The planning service uses the historical knowledge, which is a set of data points from the DS of the machine and the analyzed current state in this case, to send a plan to the shadow merger. The command is forwarded to the executor and the OPC UA client for a tool change, while the plan and the analyzed data are stored as a new data point in the database.

Mapping of information: With the identified DS types, the next step is the mapping of needs with the available information. The position, tool vibration, current, and spindle speed can be obtained from the values monitored by the sensors. However, it should be noted that the tool wear value is not subject to monitoring. This value is calculated with regard to spindle speed, vibration, and current. For the sake of simplicity, we refer to the tool's health as new, worn, or broken. In the OPC UA client implementation, a command is translated in a way that triggers a subsequent step in the milling process or a tool change, which comprises multiple steps. These steps include a movement to the tool changer, a rotation to the desired tool or position, an unloading of an equipped tool, and a loading of a new one, culminating in a final movement to an initial position, ready for the continuation of the milling process.

Digital Shadows: As preparation for the derivation of the final steps, we first update our digital shadow types. The initial DS from the shadow caster is derived from the interface of the asset in Fig. 7. This DS encompasses the position, the tool vibration, the spinning speed, and the current applied to the machine tool. Subsequently, the analyze service incorporates a tool wear index into the DS, with this index being derived from the aforementioned values. The plan service further takes a data trace of historical data with the tool wear and action taken to the already existing DS. This process culminates in the generation of a DS with an action as the output. The action is categorized as either process continuation, process stoppage, or tool change command.

Derivation of UI/UX: The UI is derived from the structured DS from the previous step. In Fig. 8 we depict the resulting UI that is generated with MontiGem. The class diagrams in Fig. 7 of the data are now used to derive the GUI models. With these, the generation of the cockpit is performed. In addition, we can customize the generated UI by adding new custom GUI models or altering the derived GUI models.

Derivation of Deployment: Finally, the deployment specification is taken from the requirements. In this case study, the computationally complex task is the simulation of the physical device. The optimization of the tool change comprises reduced complexity by using case-based reasoning on historical data and decisions. This setup allows the DT to be deployed on the simulation device for time-critical decision making. Alternatively, for non-time critical applications, such as monitoring, the DT may be deployed in a cloud environment.

Lessons Learned: We have shown a systematic approach for the development of the DT, from the initial requirements to the final product. During this process, several refinement steps were implemented, including information mapping and concept structuring. These refinement steps ensure the suitability of the concepts for their intended purpose and for aligning them with the established requirements. The deployment of the system is not elaborated in detail upon in this scope. The incorporation of IP protection, virtualization, and computation time requirements has the potential to refine the deployment of both

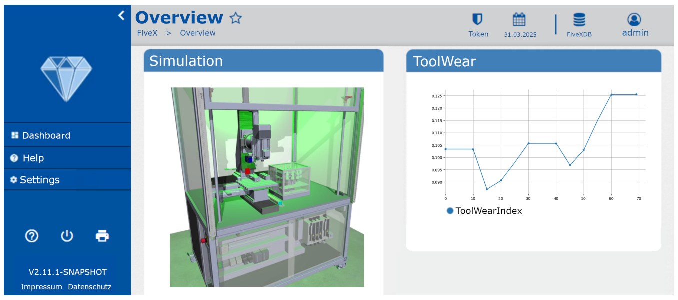


Fig. 8: Derived UI from the combined DS. Here, the tool wear derived from the analyze service is plotted.

the system and its associated services and methods. According to the evaluating PhD students, the software performed as expected. However, it's neither optimized for manual extension (unless outlined in Fig. 4), nor for real-time.

VIII. Discussion

The method presented above relies on a specific understanding of the term "digital twin" that follows from the definitions of Kritzinger [10] and Tao [13] and is in line with the ISO standard on DTs for manufacturing. Consequently, the resulting architecture focuses on representing the AS at its runtime and, thus, assumes that (a) the AS exists already and that the DT can connect to it digitally and that (b) certain components for such an architecture are necessary to manage the DT elements proposed by Tao [13] (*e.g.,* models, data, and services). Moreover, we do not assume that the DT has any essential core functionality aside from synchronizing the required data (and possibly models) between the AS and the DT. We also did not consider aspects that might be relevant to DT development, including service-level testing (especially the need for simulated physical twins to realize this), fault simulation, handling non-functional requirements, and integration with legacy systems. Every other often-cited functionality [9] of DTs, such as predictive maintenance, what-if analysis, or behavior optimization are specific to a concrete DT and may not be required by others. Hence, we assume that these functionalities are made available to the DT via dedicated services.

Based on these assumptions, we have conceived a method for the systematic model-driven engineering of ISO 232471 [12] compliant DTs that can support many industrial use cases, ranging from the computation of KPIs, to predictive maintenance to simulation-based AS behavior validation at runtime using corresponding services. This method is not only flexible regarding the purpose of the DTs to be developed, but also largely automated regarding the implementation of the architecture as only the data models, implementation of services, and of the gateway must be provided. Yet, using our base architecture as a blueprint and MontiArc's extension mechanisms, tailoring this architecture to specific use cases is straightforward as well.

Our method heavily relies on the integration of MontiGem and MontiArc, both of which can be used very flexibly on their own already. This enables supporting many extensions to the method outlined above but is far from a low-code solution for engineering DTs. By making additional restrictions, *e.g.,* to a specific gateway technology or a fixed library of services to be reused, our method can become applicable more easily. However, we opted for the flexibility, as we do not expect individual DTs to be changed by non-experts very often.

Deploying our method to the creation of DTs for manufacturing has also led to some insights about the challenges that we take as lessons learned for improving the systematic engineering of DTs.

Insight 1. Digital shadows require powerful data modeling techniques.

Digital shadows are meant to carry various metadata about their values, such as desired frequency, quality, or source of the data. As adding this information to class diagrams directly is not supported without stressing stereotypes too much, currently, this is encoded by translating the class diagram attributes defining the values into more complex data types that carry the data values but also the metadata. This is cumbersome, and we are investigating whether SysML v2 block definition diagrams are better suited for this.

Insight 2. Asset models become obsolescent over time.

Cyber-physical production systems can change over time due to wear and tear, cyber-physical augmentations (e.g., as a sensor being replaced by a newer model), changes to the software, or other environmental influences. This entails that the models and static asset information the DT and its services might rely upon for decision-making (e.g., to determine whether maintenance should take place soon) is out of sync with reality, and decisions made based on that information might be wrong. Hence, DTs need means to automatically calibrate and update their models of the AS automatically.

Insight 3. Services need to be aware of the real-time requirements of the AS.

In production, most systems rely on some form of hard real-time computation and/or communication. Whenever DT services need to manipulate the AS, this means that the services need to be aware of the real-time requirements of the AS. Otherwise, reactions by the DT's services might be too late. This entails that services contributing real-time relevant functionalities must be constructed and deployed such that the requirements of the AS can be fulfilled.

IX. Conclusion

Our comprehensive methodology has illustrated how to tame the complexity of creating DTs for manufacturing by drilling it down to dedicated steps from understanding the purpose of why one wants to create a DT, to the required interfaces, specifying models, data, services and visualizations, further down to creating an executable DT in a model-driven approach. Our industrial case studies show that applying the method is suitable in practice. With our methododolgy, we want to enable SMEs to produce a DT for a machine in a day using a novel combination of model-driven methods. In future work we will integrate popular technologies of Industry 4.0, such as the Asset Administration Shell [56], with our DTs.

Acknowledgments

Partly funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2023 Internet of Production - 390621612. Website: https://www.iop.rwth-aachen.de.
Partly funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Model-Based DevOps – 505496753. Website: https://mbdo.github.io.
Partly funded by the Bundesministerium für Wirtschaft und Klimaschutz (BMWK) – Factory-X – 13MX001F. Website https://factory-x.org.
Partly funded by the Innovation Campus Future Mobility - Website: https://www.icm-bw.de/en/.

References

[1] K. Zhang, J. Cao, and Y. Zhang, "Adaptive digital twin and multiagent deep reinforcement learning for vehicular edge computing and networks," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 2, pp. 1405–1413, 2021.

[2] D. Lehner, J. Pfeiffer, S. Klikovits, A. Wortmann, and M. Wimmer, "A method for template-based architecture modeling and its application to digital twins," *J. Object Technol.*, vol. 23, no. 3, pp. 1–14, 2024.

[3] J. Pfeiffer, D. Fuchß, T. Kühn, R. Liebhart, D. Neumann, C. Neimöck, C. Seiler, A. Koziolek, and A. Wortmann, "Modeling languages for automotive digital twins: A survey among the german automotive industry," in *ACM/IEEE 27th Int. Conf. on Model Driven Engineering Languages and Systems, MODELS'24*, pp. 92–103, ACM, 2024.

[4] A. Francisco, N. Mohammadi, and J. E. Taylor, "Smart city digital twin–enabled energy management: Toward real-time urban building energy benchmarking," *Journal of Management in Engineering*, vol. 36, no. 2, p. 04019045, 2020.

[5] P. Jain, J. Poon, J. P. Singh, C. Spanos, S. R. Sanders, and S. K. Panda, "A digital twin approach for fault diagnosis in distributed photovoltaic systems," *IEEE Transactions on Power Electronics*, vol. 35, no. 1, pp. 940–956, 2019.

[6] T. Facchinetti and T. Routhu, "Modular digital twin for air handling units," in *29th Int. Conf. on Emerging Techn. and Factory Aut. (ETFA'24)*, IEEE, 2024.

[7] E. M. Kraft, "The air force digital thread/digital twin-life cycle integration and use of computational and experimental knowledge," in *54th AIAA aerospace sciences meeting*, p. 0897, 2016.

[8] C. Mandolla, A. M. Petruzzelli, G. Percoco, and A. Urbinati, "Building a digital twin for additive manufacturing through the exploitation of blockchain: A case analysis of the aircraft industry," *Comput. Ind.*, vol. 109, pp. 134–152, 2019.

[9] 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," *Journal of Systems and Software*, p. 111361, 2022.

[10] 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.

[11] F. Tao, M. Zhang, and A. Nee, "Five-dimension digital twin modeling and its key technologies," in *Digital Twin Driven Smart Manufacturing*, pp. 63–81, Academic Press, 2019.

[12] "ISO 23247-1:2021. Automation systems and integration. Digital twin framework for manufacturing. Part 1: Overview and general principles," 2021.

[13] F. Tao, H. Zhang, A. Liu, and A. Y. Nee, "Digital twin in industry: State-of-the-art," *IEEE Transactions on industrial informatics*, vol. 15, no. 4, pp. 2405–2415, 2018.

[14] J. Michael, L. Cleophas, S. Zschaler, T. Clark, B. Combemale, T. Godfrey, D. Khelladi, V. Kulkarni, D. Lehner, B. Rumpe, M. Wimmer, A. Wortmann, S. Ali, B. Barn, I. Barosan, N. Bencomo, F. Bordeleau, G. Grossmann, G. Karsai, O. Kopp, B. Mitschang, P. Muñoz Ariza, A. Pierantonio, F. Polack, M. Riebisch, H. Schlingloff, M. Stumptner, A. Vallecillo, M. van den Brand, and H. Vangheluwe, "Model-Driven Engineering for Digital Twins: Opportunities and Challenges," *INCOSE Systems Engineering*, 2025.

[15] W. Yang, W. Xiang, Y. Yang, and P. Cheng, "Optimizing federated learning with deep reinforcement learning for digital twin empowered industrial iot," *IEEE Transactions on Industrial Informatics*, vol. 19, no. 2, pp. 1884–1893, 2022.

[16] I. Verner, D. Cuperman, A. Fang, M. Reitman, T. Romm, and G. Balikin, "Robot Online Learning Through Digital Twin Experiments: A Weightlifting Project," in *Online Engineering & Internet of Things: 14th Int. Confe. on Remote Engineering and Virtual Instrumentation (REV 2017)*, pp. 307–314, Springer, 2018.

[17] G. Knapp, T. Mukherjee, J. Zuback, H. Wei, T. Palmer, A. De, and T. DebRoy, "Building blocks for a digital twin of additive manufacturing," *Acta Materialia*, vol. 135, pp. 390–399, 2017.

[18] A. De Benedictis, F. Flammini, N. Mazzocca, A. Somma, and F. Vitale, "Digital twins for anomaly detection in the industrial internet of things: Conceptual architecture and proof-of-concept," *IEEE Transactions on Industrial Informatics*, 2023.

[19] P. Brauner, M. Dalibor, M. Jarke, I. Kunze, I. Koren, G. Lakemeyer, M. Liebenberg, J. Michael, J. Pennekamp, C. Quix, *et al.*, "A computer science perspective on digital transformation in production," *ACM Transactions on Internet of Things*, vol. 3, no. 2, pp. 1–32, 2022.

[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 (ER'21)*, pp. 271–281, Springer, 2021.

[21] M. Liebenberg and M. Jarke, "Information systems engineering with digital shadows: Concept and case studies," in *Advanced Information Systems Engineering*, (Cham), pp. 70–84, Springer, 2020.

[22] J. Michael, I. Koren, I. Dimitriadis, J. Fulterer, A. Gannouni, M. Heithoff, A. Hermann, K. Hornberg, M. Kröger, P. Sapel, N. Schäfer, J. Theissen-Lipp, S. Decker, C. Hopmann, M. Jarke, B. Rumpe, R. H. Schmitt, and G. Schuh, "A Digital Shadow Reference Model for Worldwide Production Labs," in *Internet of Production: Fundamentals, Applications and Proceedings*, pp. 1–28, Springer, June 2023.

[23] J. Michael, J. Blankenbach, J. Derksen, B. Finklenburg, R. Fuentes, T. Gries, S. Hendiani, S. Herlé, S. Hesseler, M. Kimm, J. C. Kirchhof, B. Rumpe, H. Schüttrumpf, and G. Walther, "Integrating Models of Civil Structures in Digital Twins: State-of-the-Art and Challenges," *Journal of Infrastructure Intelligence and Resilience*, vol. 3, September 2024.

[24] B. Combemale, N. Jansen, J.-M. Jézéquel, J. Michael, Q. Perez, F. Rademacher, B. Rumpe, D. Vojtisek, A. Wortmann, and J. Zhang, "Model-Based DevOps: Foundations and Challenges," in *MoDDiT Workshop, 2023 ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems Companion*, pp. 429–433, IEEE, 2023.

[25] M. Heithoff, J. Michael, and B. Rumpe, "Enhancing Digital Shadows with Workflows," in *Modellierung 2022 Satellite Events*, pp. 142–146, Gesellschaft für Informatik e.V., June 2022.

[26] M. Heithoff, C. Hopmann, T. Köbel, J. Michael, B. Rumpe, and P. Sapel, "Application of Digital Shadows on Different Levels in the Automation Pyramid," *Data and Knowledge Engineering (DKE)*, 2025.

[27] A. Butting, A. Haber, L. Hermerschmidt, O. Kautz, B. Rumpe, and A. Wortmann, "Systematic language extension mechanisms for the montiarc architecture description language," in *Modelling Foundations and Applications: 13th European Conference, ECMFA 2017, Held as Part of STAF 2017, Marburg, Germany, July 19-20, 2017, Proceedings 13*, pp. 53–70, Springer, 2017.

[28] B. Rumpe and A. Wortmann, "Abstraction and refinement in hierarchically decomposable and underspecified cps-architectures," in *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, pp. 383–406, Springer, 2018.

[29] K. Hölldobler, J. Michael, J. O. Ringert, B. Rumpe, and A. Wortmann, "Innovations in Model-based Software And Systems Engineering," *The Journal of Object Technology*, vol. 18, pp. 1–60, July 2019.

[30] K. Adam, L. Netz, S. Varga, J. Michael, B. Rumpe, P. Heuser, and P. Letmathe, "Model-Based Generation of Enterprise Information Sys-

tems," in *Enterprise Modeling and Information Systems Architectures (EMISA'18)* (M. Fellmann and K. Sandkuhl, eds.), vol. 2097 of *CEUR Workshop Proceedings*, pp. 75–79, CEUR-WS.org, May 2018.

[31] A. Gerasimov, J. Michael, L. Netz, and B. Rumpe, "Agile Generator-Based GUI Modeling for Information Systems," in *Modelling to Program (M2P)* (A. Dahanayake, O. Pastor, and B. Thalheim, eds.), pp. 113–126, Springer, March 2021.

[32] D. Bano, J. Michael, B. Rumpe, S. Varga, and M. Weske, "Process-Aware Digital Twin Cockpit Synthesis from Event Logs," *Journal of Computer Languages (COLA)*, vol. 70, June 2022.

[33] A. Gerasimov, P. Letmathe, J. Michael, L. Netz, and B. Rumpe, "Modeling Financial, Project and Staff Management: A Case Report from the MaCoCo Project," *Enterprise Modelling and Information Systems Architectures - International Journal of Conceptual Modeling*, vol. 19, February 2024.

[34] C. Buschhaus, A. Gerasimov, J. C. Kirchhof, J. Michael, L. Netz, B. Rumpe, and S. Stüber, "Lessons Learned from Applying Model-Driven Engineering in 5 Domains: The Success Story of the MontiGem Generator Framework," *Science of Computer Programming*, vol. 232, p. 103033, 2024.

[35] M. Dalibor, M. Heithoff, J. Michael, L. Netz, J. Pfeiffer, B. Rumpe, S. Varga, and A. Wortmann, "Generating customized low-code development platforms for digital twins," *Journal of Computer Languages*, vol. 70, p. 101117, 2022.

[36] S. Vale, S. Reddy, S. Subramanian, S. R. Chaudhuri, S. H. Nistala, A. Deodhar, and V. Runkana, "A model-driven approach for knowledge-based engineering of industrial digital twins," in *26th ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems (MODELS'23)*, pp. 13–23, IEEE, 2023.

[37] B. J. Oakes, C. Gomes, E. Kamburjan, G. Abbiati, E. E. Bas, and S. Engelsgaard, "Towards ontological service-driven engineering of digital twins," in *ACM/IEEE 27th Int. Conf. on Model Driven Engineering Languages and Systems, MODELS Companion 2024*, pp. 464–469, ACM, 2024.

[38] J. Maree, K. Kruger, and A. Basson, "Opportunities for digital twins for the provisioning, management and monitoring of heterogeneous iot devices," in *ACM/IEEE 27th Int. Conf. on Model Driven Engineering Languages and Systems, MODELS Companion 2024*, pp. 378–389, ACM, 2024.

[39] Y. Zhao, A. Eve, T. Miny, and T. Kleinert, "Bridging the gap: A python-to-structured text compiler with IEC 61131-3 compliance," in *29th IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA'24)*, pp. 1–7, IEEE, 2024.

[40] L. Pietrantoni, M. S. Román-Niaves, and M. de Angelis, "Human trust and digital twins in a human factors and ergonomic framework," in *29th IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA'24)*, pp. 1–9, IEEE, 2024.

[41] A. D. Rocha, M. Arvana, N. Freitas, R. M. Dinis, T. Gouveia, D. Machado, and J. Barata, "Human-centric digital twin-driven approach for plug-and- produce in modular cyber-physical production systems," in *29th IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA'24)*, pp. 1–7, IEEE, 2024.

[42] I. Garg, D. Hoffmann, A. Lüder, and M. Rudolph, "Integrating domain expertise into dynamic digital models: A methodology for behaviour-driven production modelling," in *29th IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA'24)*, pp. 1–8, IEEE, 2024.

[43] V. K. de Almeida, D. E. M. de Oliveira, C. D. T. de Barros, G. dos Santos Scatena, A. N. Q. Filho, F. L. Siqueira, A. H. R. Costa, E. S. Gomi, L. A. F. Mendoza, E. C. S. Batista, C. E. Muñoz, I. G. Siqueira, R. A. Barreira, I. H. F. dos Santos, C. Cardoso, E. S. Ogasawara, and F. Porto, "A digital twin system for oil and gas industry: A use case on mooring lines integrity monitoring," in *ACM/IEEE 27th Int. Conf. on Model Driven Engineering Languages and Systems, MODELS Companion 2024*, pp. 322–331, ACM, 2024.

[44] E. Kamburjan, N. Bencomo, S. L. T. Tarifa, and E. B. Johnsen, "Declarative lifecycle management in digital twins," in *ACM/IEEE 27th Int. Conf. on Model Driven Engineering Languages and Systems, MODELS Companion 2024*, pp. 353–363, ACM, 2024.

[45] G. D. S. Sidibé and S. Dhouib, "An approach for sovereign data exchange of AAS digital twins through the international data space network," in *29th IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA'24)*, pp. 1–4, IEEE, 2024.

[46] G. d'Aloisio, A. D. Matteo, A. Cipriani, D. Lozzi, E. Mattei, G. Zanfardino, A. D. Marco, and G. Placidi, "Engineering a digital twin for diagnosis and treatment of multiple sclerosis," in *ACM/IEEE 27th Int. Conf. on Model Driven Engineering Languages and Systems, MODELS Companion 2024*, pp. 364–369, ACM, 2024.

[47] S. Parbat, I. Iqbal, I. Viedt, and L. Urbas, "Architecture of digital twin for automating waste and recycling material sorting process," in *29th IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA'24)*, pp. 1–4, IEEE, 2024.

[48] Q. Nguyen, Y. Huang, G. D. S. Sidibé, and S. Dhouib, "From multiple digital twins to a multi-faceted digital twin: Towards an aas-based approach," in *29th IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA'24)*, pp. 1–4, IEEE, 2024.

[49] S. Barat, V. Kulkarni, T. Clark, and B. Barn, "Digital twin as risk-free experimentation aid for techno-socio-economic systems," in *25th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS'22)*, pp. 66–75, ACM, 2022.

[50] I. David, P. Archambault, Q. Wolak, C. V. Vu, T. Lalonde, K. Riaz, E. Syriani, and H. A. Sahraoui, "Digital twins for cyber-biophysical systems: Challenges and lessons learned," in *26th ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems (MODELS'23)*, pp. 1–12, IEEE, 2023.

[51] P. Spaney, S. Becker, R. Ströbel, J. Fleischer, S. Zenhari, H.-C. Möhring, A.-K. Splettstößer, and A. Wortmann, "A model-driven digital twin for manufacturing process adaptation," in *2023 ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pp. 465–469, IEEE, 2023.

[52] D. Lehner, J. Zhang, J. Pfeiffer, S. Sint, A.-K. Splettstößer, M. Wimmer, and A. Wortmann, "Model-driven engineering for digital twins: a systematic mapping study," *Software and Systems Modeling*, pp. 1–39, 2025.

[53] 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," in *Int. Conf. on Advanced Information Systems Engineering (CAiSE'20)*, vol. 12127 of *LNCS*, pp. 85–100, Springer, June 2020.

[54] J. Michael, I. Nachmann, L. Netz, B. Rumpe, and S. Stüber, "Generating Digital Twin Cockpits for Parameter Management in the Engineering of Wind Turbines," in *Modellierung 2022*, pp. 33–48, Gesellschaft für Informatik, June 2022.

[55] W. Mahnke, S.-H. Leitner, and M. Damm, *OPC unified architecture*. Springer Science & Business Media, 2009.

[56] Industrial Digital Twin Association, "Specification of the Asset Administration Shell. Part 1: Metamodel," 2023.