An Architecture for Coupled Digital Twins with Semantic Lifting

Santiago Gil[1†], Eduard Kamburjan[2†], Prasad Talasila[1], Peter Gorm Larsen[1]

[1]Department of Electrical and Computer Engineering, Aarhus University, Aarhus, Denmark.
[2]Department of Informatics, University of Oslo, Oslo, Norway.

Contributing authors: sgil@ece.au.dk; eduard@ifi.uio.no; prasad.talasila@ece.au.dk; pgl@ece.au.dk;
[†]These authors contributed equally to this work.

Abstract

To enable the reuse of Digital Twins, in the form of simulation units or other forms of behavioral models, of single physical components, one must be able to connect and couple them. Current platform and architectures consider mostly monolithic digital twins and offer little support for coupling and checking the consistency of the coupling. The coupling must be internally consistent — satisfy constraints related to their co-simulation — and externally consistent — mirror the structure of the composed physical system. In this paper, we propose an extension to a behavior-extended Digital Twin architecture for individual Digital Twins to include co-simulation scenarios for coupled systems lifted from configuration files, which can be implemented along with a Digital-Twin-as-a-Service platform to make assets reusable in time. To monitor and query these connections, we introduce a semantic lifting service, which interprets the coupled Digital Twins as Knowledge Graphs and enables the use of queries to express internal and external consistency constraints. Two representative case studies for systems with coupled behavior are used for the demonstration of this approach and show that it indeed enables reusability of components and services between different Digital Twins.

Keywords: Digital Twin, Knowledge graph, Behavioral model, Co-simulation

1 Introduction

Digital Twins (DTs) are a paradigm that connects physical systems with their digital model in a closed loop to enable monitoring and other analyses throughout the complete life cycle of a system. To accommodate the need for their simple and efficient integration into workflows, DT platforms provide the foundation to easily connect the Physical Twin (PT) with its DT, which often contains a simulation aspect [1].

Existing DT platforms are designed to reduce some of the implementation effort related to connectivity and deployment [2], but these focus primarily on the structural data representation and not much on the behavioral aspects of the DTs [3], i.e., coupled behavior on existing DT platforms cannot be covered either. Extending from DT platforms, there are also Digital-Twins-as-a-Service (DTaaS) platforms [4, 5].

So far, such DTaaS platforms focus on single systems, providing means to connect simulators to live data streams and actuators. However, systems are interconnected with each other and are changing during their lifetime, which requires (a) connecting their DTs, and (b) managing these connections. Consider a system with several distillation columns, connected in a row. As their behavior is interconnected, their simulations must be as well. These connections must be correct in two ways: First, they must have the right simulation structure with respect to connected simulators. Second, they must have the right semantic structure and configuration with respect to domain and PT.

Without a compositional approach, the whole system must be twinned by a *single* simulator. The inner structure of both the simulator and system is lost, which makes the reuse harder (in the case of the simulators) and loss of information about internal structure (in the case of the system). This internal structure is however important to analyze single components in isolation. Nevertheless, the fine composition of behavioral models is a complex task because even if the components and models are logically and correctly composed, some of the dynamic and programmatic aspects are difficult to cover in a functional way [6, 7].

We, thus, face two challenges: (1) How to extend the implementation for composed DTs with behavioral models when managing coupled behaviors and (2) how to give actionable semantics to such structures and their configuration.

We pose the following research questions to investigate, and subsequently, assess our contribution.

RQ 1 How can a software architecture enable the functional implementation of composed DTs with coupled behavior while enhancing reusability?
RQ 2 How can the structure of a composed DTs in a software architecture be queried and reasoned over at runtime?
RQ 3 How to further boost the reusability of the architectural approach in more heterogeneous software environments?

In this paper, we propose an architecture for a co-simulation-enabled behavior-extended DT platform that overcomes these challenges and enables *hierarchical* and *coupled* DTs. To do so, we take the architecture proposed in [8] to integrate behavioral models as an extension to existing DT platforms as the starting point, and then, propose additional classes and interfaces to cover *DT Systems*, which are an abstract composition of connected individual DTs. Additionally, to manage the increased complexity of the structure there is a *semantic lifting [9] service*, that maps the structure of the platform, including the managed twins, into a Knowledge Graph (KG), and enables other services to use this KG to configure the structure, detect errors in connections, and load and store configurations.

We evaluate the approach on a multi-case study setting with two case studies: (1) a system of three connected water tanks for illustration, and (2) a manufacturing cell where two industrial robotic arms cooperate for assembly processes.

Structure

The remainder of this paper is structured as follows: Section 2 presents the background and research context of this study and the case studies used in this work. Section 3 elaborates on the proposal of this work. Section 4 illustrates the instantiation of the proposed architecture on the case studies. The approach is then evaluated in Section 5. Section 6 discusses the main outcomes and limitations of the study. Finally, Section 7 provides the concluding remarks and directions for future work.

2 Background

2.1 State-of-the-Art

2.1.1 Digital Twin

DTs were proposed as a virtual representation of a physical asset, usually called the PT, with enabled bi-directional communication [10]. DTs are also implementations of a Cyber-Physical Systems (CPSs), where there is an integration of cyber components (i.e., the DTs themselves) and physical components (i.e., the PTs) [11]. There are several definitions for DTs in the literature [12]. The classification proposed by Kritzinger et al. [10] distinguishes a DT, which enables bi-directional communication with its PT, in comparison to its downgraded versions, the Digital Shadow (DS) and the Digital Model (DM); the former enables uni-directional communication from PT to DT but not vice versa and the latter has no communication capabilities at all. However, this classification of DTs in comparison to DSs and DMs is vague in relation to minimal services and models a DT should contain since it does not elaborate on a DT in terms of its business goals or logic. Regarding this aspect, DTs may require different types of models, including *data models* (i.e., geometry, structure, information, etc.) and

behavioral models (i.e., system dynamics, physics, state-machines, etc.) [13, 14] to achieve different services that provide a benefit [15], including optimization, self-reconfiguration, and what-if simulation [16, 17]. Further elaborating on this, behavioral models are especially relevant when the DTs are intended to include simulations and experimental versions of the PTs [18], and so these can be used for virtual commissioning too [19].

Basic implementations of DTs that satisfy Kritzinger's definition of a DT include the representation of digital assets, which map properties of physical assets, such as attributes and operations, in a hierarchical style. Such representations can be achieved by using, for example, Asset Administration Shell (AAS), which recently turned into the IEC 63278-1 Standard [20]. These representations can commonly be used by existing DT platforms, such as Azure Digital Twins, Eclipse Ditto, Eclipse BaSyx, and AWS IoT Greengrass [2]. However, simulations and semantic analysis are not necessarily included in such existing platforms and model-based representations, which may limit the usage of the DT services or increase the difficulty of implementation when simulations or semantic analysis are required. Other simulation-driven frameworks, such as the INTO-CPS Co-simulation framework [21], can be used for implementing DTs while complementing the simulation requirements [22]; on the downside, co-simulation frameworks do not follow model-based mechanisms, which limits the rapid initialization and versatile implementation of DTs that can easily be integrated with other external systems. As an attempt to overcome the current challenges, our previous work [8] proposed an architectural framework to bridge the gap between existing DT platforms and integrated simulation for individual decoupled DTs by defining endpoints connected to physical assets or simulation engines that can be initialized from configuration files and twin schemas, including AAS representations, following model-based engineering practices.

DTs can also be engineered and operated by reusing existing software components and models. Reusing DT components and models may avoid the new development of a DT for its corresponding PT from scratch [16, 23]. Other implementations for DT deployment have used a model-based design approach to increase the reusability of DT components from a high-level perspective [24]. The reusable components can, for instance, be hosted in a DTaaS platform [4], where along with other tools and services, DTs have a different lifecycle and can be used for different purposes with less design and implementation effort. Our previous work on a DTaaS platform [5] is a complementary approach across different design patterns for developing DTs. The DTaaS platform provides an infrastructure to host and maintain DTs and their assets with a particular asset configuration definition. Within the platform, it is possible to convey existing DT platforms and multiple engines to run simulations of the DTs, and integrate them with third-party services. The DTs in such a DTaaS platform are defined by (i) *DT assets*, namely, *Data*, *Models*, *Functions*, and *Tools*; (ii) the *DT configuration(s)*, including the reference to services, external integrations, endpoints, and so on;

and (iii) six lifecycle phases, *namely*, *create*, *execute*, *save*, *analyze*, *evolve*, and *terminate*, which determine the lifecycle of DTs.

2.1.2 Composition of Digital Twins

Composition enables the representation of entities as hierarchies of meaningful components, which provides a richer context and constraints to the composed system [25]. Composition is highly relevant in software engineering, where its has been effective for system implementation and reusability [26]. Due to the cyber-physical nature of DTs, composition is also relevant to enable the effective reuse of DT components [16].

The composition of DTs is another way of reusing the components of available smaller DTs into larger DTs that have different scopes [16]. The composition DTs can be used to represent complex systems by splitting them into sub-components, which are usually split by spatial or contextual reference [27]. The concept of composition of DTs has been presented in several studies [16, 27–29], and in other formats as composition by microservice architectures [30], *aggregation* of DTs [31], *systems-of-systems* of DTs [6], and hierarchical DTs [32].

Nevertheless, composing DTs poses a challenge when the systems are 1) heterogeneous, 2) highly coupled, and 3) highly dynamic [6, 7]. Even though some previous studies have proposed the composition of DTs through relationships like *isComposedOf* [33, 34], they do not cover the composition for dynamic or highly coupled systems with programmatic features.

2.1.3 Co-simulation

Simulation has been used since the 20th century to analyze *what-if* questions about existing or conceptual systems, with the ability to run cost-effective experiments in a risk-free scenario [35, 36].

A newer approach for simulation, and a key enabling technology for implementing DTs, is co-simulation [37]. The standard we use for co-simulation allows the global simulation of a coupled and complex system through the composition of heterogeneous simulators, that is, co-simulation upon the Functional Mock-up Interface (FMI) standard[38]. Therefore, we use Functional Mock-up Units (FMUs) for model exchange and co-simulation. Co-simulation is particularly useful when the DTs are to include their behavioral models and these are coupled to other components. In addition to this, co-simulation addresses the challenge for interoperability of models and simulation environments in the realization of DTs [6].

Co-simulation is also related to the concept of hardware-in-the-loop simulation [37, 39], where it is possible to integrate simulators with hardware components. This feature for co-simulation, though, requires to integrate hybrid timing mechanisms to combine discrete events and continuous dynamics [40].

There are available tools for implementing co-simulation, such as the *integrated tool chain for Cyber-Physical Systems*, the INTO-CPS application, which

was introduced by Larsen et al. [21] as a tool suite for model-based design and implementation of co-simulation experiments. The INTO-CPS application uses Maestro as the co-simulation engine, introduced by Thule et al. [41], which orchestrates the co-simulation setups of heterogeneous simulators that compose cyber-physical systems.

Some studies that use co-simulation for DTs include Havard et al. [42], who proposed an architecture for co-simulation and communication between DT and virtual reality software, with a case study of a robotic arm; and Fitzgerald et al. [43], who implemented multi-modeling and co-simulation for the design of cyber-physical systems and their further integration with DTs capable of performing online and offline decision-making, with a case study of a line-following robot. The latter study uses the INTO-CPS application as the co-simulation orchestration environment.

2.1.4 Semantics and Digital Twins

Semantic lifting [9] has been introduced for general programs, including geological simulators [44], and applied by Kamburjan and Johnsen [45] to lift programs containing co-simulation units, which is in effect lifting arbitrary co-simulation master algorithms. This was then subsequently used to self-adapt to structural defects in the twinning relation in DTs [46]. However, due to the generality of lifting the whole program, the lifted state of a platform is not in terms of platform concepts, but in terms of the implementation language. Thus, an additional *modeling* step is required to analyze those systems in terms of an architecture.

The semantic interpretation of a software enables several operations, the most important being that we can define and check *consistency* of the DT structure in a uniform way, based on a formalization of consistency on a semantic level of a KG. This has proven useful for programs [45, 46] that monolithically model and control a PT. Consistency can be either internal, i.e., expressing that the connections between DTs possess some property, e.g., no loops, or external. External consistency expresses that the structure coupling the simulation units is consistent with the structure of the simulated PTs and adheres to domain constraints, e.g., that a water tank cannot be directly connected to an oil tank. This way, we can address *behavioral aspects* of consistency. Both domain constraints and physical connections can be expressed in ontologies and KGs, which can be connected to the lifted structure.

During the execution of a semantically lifted program, its states are lifted to different KGs: not only do concrete properties (such as data values stored in variables) change, but so does the structure of the state itself. Whenever an object is created or references are change, the lifted state drastically changes. In the architectural setting of this work, the structure is less dynamic, as all components are known *a priori*. However, their properties still change, and the concrete identity or number of components may change as well.

Paredis and Vangheluwe [47] aim to coordinate multiple DTs for the same physical system, by providing a virtual KG that describes their purposes and description. However, their approach is not concerned with coupling behavior or lifting computational structures.

2.1.5 Cognitive Digital Twins

Cognitive DTs (CDTs) put forward the idea to use an ontology to enable enhanced decision-making based by integrating data using a top-level ontology and using it to provide a uniform and semantic view on the system. This is very much in line with our proposal, as CDTs aim to use semantics, among others, for “ [...] identifying the dynamics of virtual model evolution, promoting the understanding of interrelationships between virtual model [...]” [48–50]. The proposed original framework [48] however is too abstract to guide the design of concrete architectures. While it does consider the interrelations between simulation models, it does not provide any support for their coupling. Recently, Li et al. [51] propose the integrated view to configure co-simulation scenarios. They do not use this to detect domain-specific or structural errors but semantically interpret the co-simulation scenario to express knowledge about co-simulation itself.

Other definitions of CDTs are more concrete in what functionalities the use of KGs enable, but focus on different aspects: Abburu et al. [52] consider CDTs as system that use KGs to adapt to unforseen situations, while Ali et al. [53] use KGs to establish a network to enable reasoning capabilities. For further, less related, definitions we refer to Zheng et al. [54].

2.1.6 Research Gap

Although the architectural approach in [8] intends to overcome the shortcomings regarding the integration of behavioral models (simulation) in existing DT platforms, the heterogeneous composition DTs with behavioral models still poses a major challenge in the DT field [6, 7]. More precisely, such an architectural approach is limited to function with individual DTs, i.e., DTs cannot have internal coupling with other fellow DTs, constraining the approach to realize hierarchical DTs. Although hierarchical composition of DTs has been touched in several papers [6, 7, 16, 27–29, 31, 32], these mostly cover the conceptual/modeling aspect of composition, which still limits its implementation. Moreover, although the DTaaS platform seems suitable to address this challenge since it is a platform to host multiple DTs, it behaves as a high-level hub-like platform to orchestrate tools, models, data, and configurations to the end of creating DTs rather than a specialized solution to realize DT systems with coupled behavior itself.

KGs and ontologies have been proposed to enhance DTs, an idea sometimes referred to as CDTs, but are rarely concerned with analyzing coupled behavior, and are often too abstract to guide architecture design. Approaches based on some form of semantic lifting have been demonstrated to have potential for this task, but are so far limited to analyze the structure in terms of the *programming*

language, which is disconnected from the software *architecture*. Furthermore, current approaches to semantic lifting based on the programming language face scalability issues, as by default the whole software state is part of the KG [44].

Hence, in this paper, we take inspiration of the conceptual modeling approach for DTs with composition enabled proposed in [7] to propose an extension to the architectural approach in [8] for supporting the realization of hierarchical DTs, i.e., DT systems with coupled behavior. This approach also takes inspiration of the structure given by the ontological model in [7], to provide an improvement to the ontology that enables the attachment of DT services based on semantic reasoning and querying. Additionally, for the sake of conveying the engineering effort in the DT engineering process, the DTaaS platform proposed by Talasila et al. [5] is also considered to showcase how the architectural approach can be integrated into such an environment to create DT applications with enhanced reusability. In terms of semantic lifting, we extend it in this paper to the generation of a KG of a DT *architecture*, not just a single program. This gives the generated KG the right abstraction level to reason about coupled behavior of components and reduce its size.

2.2 Introduction to Case Studies

For the evaluation of this approach, we use case study research. More precisely, we use a multi-case study research setting [55] aiming at finding the theoretical generalization of the methods by architectural similarity [56].

The two case studies, coming from two different domains, provide a sufficient base to theoretically validate the generalizability of the method proposed here. The case studies are introduced by complexity. The first case study is a simulation-only system and represented by a single DT class. The second case study counts with PT and simulation, and is represented by four DT classes.

In the following, we describe two DT systems as our case studies.

2.2.1 Three-Tank System

This simple case study allows us to represent a system that is composed of three individual components that are highly coupled because and output of the first tank (o_1) is connected to the input of the second tank (i_2), and the output of the second tank (o_2) is connected to the input of the third tank (i_3), as shown in Figure 1. In a static composition of the Three-Tank system, the virtual representation of the system can be trivial; however, the dynamic composition of the behaviors of the individual tanks complicates the virtual representation of the actual coupled system.

The scenario is mainly for demonstration purposes and, thus, it only counts with the simulation part. It, however, still allows us to collect computation metrics that do not require actually physical system, and investigate new features enabled by the architecture.

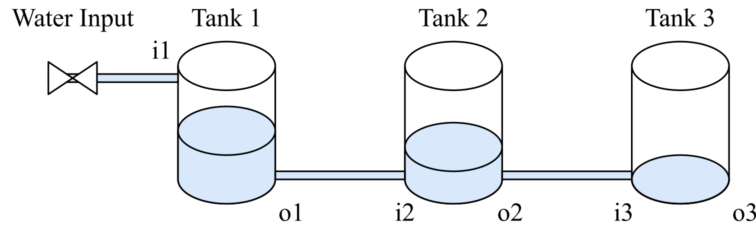


Fig. 1: Graphical representation of the Three-Tank System mock-up.

2.2.2 A Coupled DT of a Manufacturing Cell: The Flex-cell

We use the *flex-cell* case study [57], which is a manufacturing cell composed of two industrial robotic arms, namely, the Kuka lbr iiwa 7 and the Universal Robots UR5e, to evaluate the approach in terms of coupled behavior due to synchronized motions. Figure 2 shows the physical setup of the Flex-cell case study. The case study is representative of a composed, dynamic, and complex system where both robotic arms are intended to cooperate, yet current DTs struggle to handle their behavioral coupling [7].

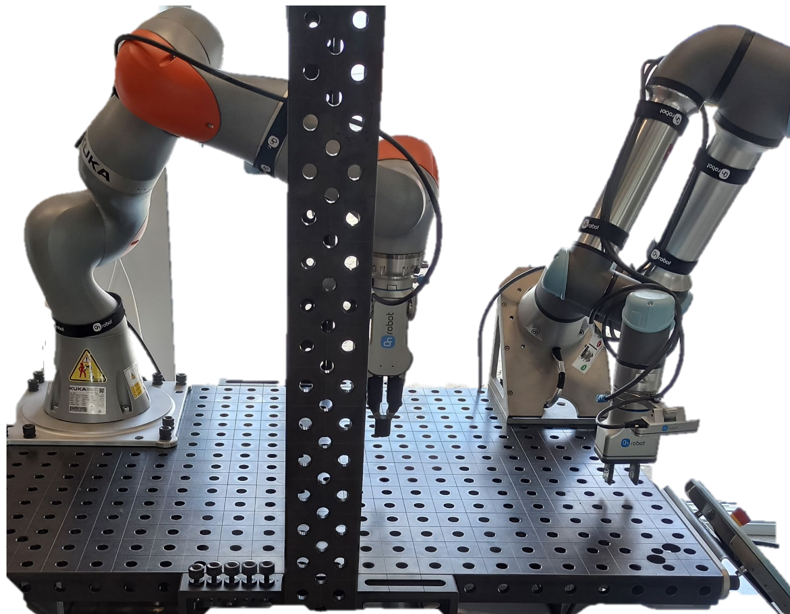


Fig. 2: Flex-cell case study: A manufacturing cell composed of the robotic arms Kuka lbr iiwa 7 (left) and UR5e (right) and grippers OnRobot RG6 (left) and 2FG7 (right).

The robotic arms can be modeled in several ways, including robot kinematics and dynamics techniques, which are important when simulating the robots [58]. The robot dynamics can be particularly difficult to model and compute due to the robotic arms being 7-axis and 6-axis robots respectively. There are some tools, such as the AURT Toolbox [59], which can be used to provide the dynamic models of the robots based on data recorded.

Our scope is limited to the positioning of the robots with respect to the flex-cell plate space; therefore, the kinematic models are sufficient to satisfy the requirements in this case. We make use of the Robotics Toolbox for Python [60] to include the forward and inverse kinematics, and trajectory generation. The models are complemented with specific transformations for the Flex-cell plate space, i.e., there is a discrete cartesian space mapping the 16×24 holes separated by 5mm each to the (X, Y) axis. The Z axis is also discretized using a separation of 5mm for the positioning. The models for the robots are then stored in FMUs using the UniFMU tool [61], enabling the interfacing of the models with the FMI interface.

3 Materials and Methods

In this section, we are facing two challenges: (1) How to implement DTs with coupled behavior, such that these connections are easy to configure and access and (2) how to use semantic structures over these connections to utilize uniform monitoring of structure. Hence, we introduce the solution in two steps. First, taking the *DT Manager* architecture proposed in [8], hereafter called *Twin Manager*, we extend it to structure behaviorally coupled DTs in Section 3.1. Second, we describe how the structure of the architecture can be lifted to a KG, and describe the usage of reusable services for monitoring in Section 3.2.

Beyond analysis and monitoring, the very same semantic representation can also be used for loading and storing configurations. This creates a uniform representation for the configuration and monitoring of coupled and behaviorally interconnected DTs.

Terminology

Digital Twin, Digital Shadow, and Digital Model. Taken from the categorization of Kritzinger et al. [10]. A Digital Twin is a digital representation of a real-world entity with automatic bi-directional data flow. A Digital Shadow only has data flow from from its real-world entity to the model. A Digital Model has no automatic data flow capabilities.

Physical Twin. A physical object that is being featured by a Digital Twin.

Twin (Abstraction). An abstraction used along with the Twin Manager to represent the state and interface of either a Digital Twin or a Physical Twin.

Twin System. An abstraction for systems that aggregate twin composites.

Physical Twin System. An abstraction that represents a composition of Physical Twins.

Digital Twin System. An abstraction that represents a composition of Digital Twins.
Digital Twin-enabled System. A delimitation of the joint system where Physical Twins and Digital Twins are engineered (*twined*) together [62].
Digital Twin Platform. Refers to the software operating environment where the abstractions for Digital Twin-enabled Systems are twined and synchronized.
Digital Twin Service. Refers to the perceived benefits, usages, or use cases [15, 17] a Digital Twin-enabled System provides.
Endpoint. Refers to the software interfaces between the twin abstractions and their corresponding data sources.
Defect Query. Refers to a query that encodes consistency conditions on a system. It returns a set of *violations* of these conditions. The violations are referred to as *defects*.

3.1 Architecture for coupled DT systems

3.1.1 Concepts and Foundations

The architecture for coupled DT systems is conceptualized from the Twin Manager approach proposed in [8]. The Twin Manager provides an interface to easily read from and write to PTs and simulation by providing the abstraction *Twin* and the interface *Endpoint*. The Twin Manager administrates indistinctly PTs and DTs as *Twins*, providing the same methods to access either the physical or simulation endpoints through specializations of the interface *Endpoint*, as shown in Figure 3.

The Twin Manager architecture is defined by three main layers, namely, the *DT Platform Layer*, where the *Twins* are created, managed, executed, and stored; the *Endpoint Layer* where the connections to PTs and simulations are carried out; and the *DT Service Layer*, where the case-independent DT services are defined to be executed on demand. Ideally, any DT service could be attached to the Twin Manager to provide a particular benefit for the DT-enabled system in effect. The instantiation of the architecture for a particular case study requires *Twin schema files*, which define the data model of the twin classes, and *twin configuration files*, which contain the endpoint information for each twin. These files facilitate the quick setup of the applications using the models and structured data provided.

The key components of the architecture are 1) the `TwinManager` class, which is the unique interface for the user and services to access the twins and their attributes and operations, and 2) the `Endpoint` interface, which is an interface that is used for multiple specializations for different data sources, such as for example, the FMI interface.

However, as discussed in Section 2.1.6, this architecture only supports individual simulations, and thus, it does not have the capabilities to deploy hierarchical DTs that have coupled behavior.

We want to incorporate the capability to realize twin systems that are composed of multiple twin composites, i.e., hierarchical twins, as shown in Figure 4.

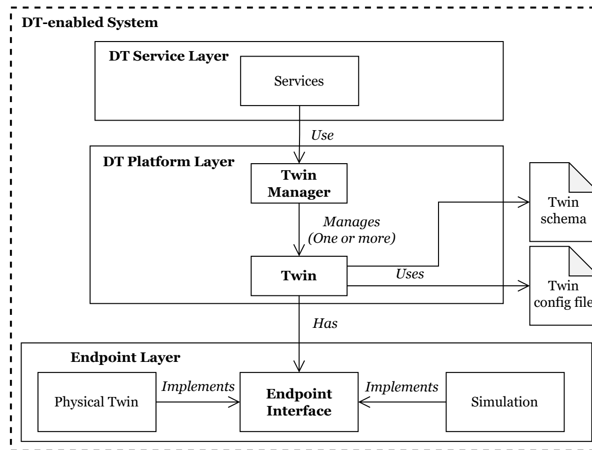


Fig. 3: Architectural abstraction for realization of twins in [8].

Here, the PT system is a composition of smaller PTs which have some kind of internal coupling. The simulation needs to represent the PT composition by smaller simulation composites and replicate the same coupling existing in the PT system. However, homogeneity in models and simulators is not guaranteed, which means, these may be heterogeneous and not necessarily interoperable, which is a current challenge in composition of DTs [6].

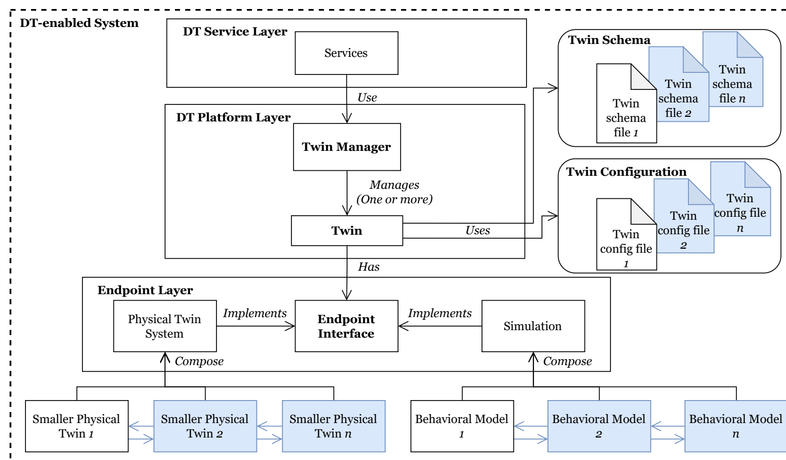


Fig. 4: Architectural abstraction for realization of (hierarchical) twin systems. The blocks in blue refer to additions due to having several twin composites and the arrows in blue refer to the internal coupling among them.

3.1.2 Proposed Extensions

We stick to the same layers and components of the base architecture for this extension, i.e., we keep the same architectural structure and patterns, where the TwinManager and Twin classes follow a Façade pattern and the Endpoint interface follows a Factory pattern [63]. We then propose new concepts to cover DT-enabled systems with coupled behavior. First, a new class, the `TwinSystem` class, which represents the composition of DTs and provides the interfaces to administrate the attributes and operations of the internal twins and the composed system. Since we keep the model-based approach, the `TwinSystem` is proposed along with a new configuration file, the *Twin System Configuration file*, which defines how are inputs and outputs bound in the twin system. This class, behaving similar to the `Twin` class, also follows a Façade pattern targeting the twin system endpoint with the methods `get/setAttributeValue` and `executeOperation` at the system level.

The `TwinManager` class is also extended with the new interfaces to manage the `TwinSystem` class, i.e., a new creation method `createTwinSystem` and the interfaces for the `get/setAttributeValue` and `executeOperation` at the system level. These methods are named `get/setSystemAttributeValue` and `executeOperationOnSystem` unlike it is done for individual twins.

Moreover, since there is no specific *endpoint* for interfacing twin systems, we propose a new abstraction to do so. Such an abstraction consists of splitting the base *Endpoint* interface into two children interfaces, the *IndividualEndpoint* and the *AggregateEndpoint* interfaces. The former works similarly to how the *Endpoint* interface was defined initially in [8], and therefore, connects individual endpoints, referring to individual decoupled twins. The latter focuses on enabling the interface to systems of systems, i.e., twin systems that are composed of multiple twin composites. The *AggregateEndpoint* interface enables specializations to access a PT system and its components from a hierarchical view and a DT system as a co-simulation setting with a particular co-simulation engine. Further elaborating on this interface, a co-simulation setting can be attached to the Twin Manager, representing the behavioral components of the DT system. Although the focus of this co-simulation endpoint is on representing the coupling in the virtual side, i.e., on the DT side, it can also be used to represent hybrid cyber-physical PT systems in hardware-in-the-loop co-simulation settings. However, this feature is out of the scope of this work, and thus, it is uniquely used to represent the DT system. The PT system, on the other hand, does not yet count with a particular specialization of the *AggregateEndpoint* interface since the coupled behavior is assumed to be present intrinsically in the system.

The specialization for the DT system part takes inspiration from the FMI standard as the mechanism to integrate heterogeneous behavioral models in a coupled co-simulation setting [37]. We choose the second version of the *Maestro* co-simulation engine [41], part of the INTO-CPS co-simulation framework [21],

as the open-source co-simulation interface to create the co-simulation-enabled endpoint specialization, named `MaestroEndpoint`.

The `MaestroEndpoint` specialization requires a file defining the simulation conditions and parameters and the connections between the different FMUs in the system. This file is represented by the *Twin System Configuration file*. This file can be set up on JSON, which is then interpreted by the Maestro engine as a *mabl* specification.

The `MaestroEndpoint`, initialized with a twin system configuration file, uses Maestro as a slave, enabling the execution of a co-simulation for a certain amount of time or stepped co-simulations (via *doStep* calls), which returns the simulation values for the multiple components in the system. This endpoint specialization, along with the Twin Manager, provides a bridge between the output files of the co-simulation and the programmatic features of the system, making the data accessible through the Twin Manager's getter and setter methods. Additionally, it is also possible to input values to and read from the co-simulation using the RabbitMQ FMU (RMQ FMU) [64]. The RMQ FMU provides a channel where a controller can send commands, and thus, these are forwarded to both the PT and DT simultaneously. Similarly, the co-simulation outputs are logged to the RMQ FMU channel, and thus, the Twin Manager can read the events and synchronize the DT with the PT.

Figure 5 illustrates the prototypical implementation for the architectural extension for twin systems with coupled behavior using a class diagram.

3.1.3 Configurations

In order to set up the DT-enabled system, three types of files are needed as follows: the *Twin Schema files*, the *Twin Configuration files*, and the *Twin System Configuration files*.

First, the *Twin Schema files* contain the data models of the twin in effect in the DT-enabled system. One schema is required per asset class. This schema file can be provided, for instance, using AAS representations. For example, in the Three-Tank System case study, only one schema is required since the three objects belong to the same class and there are no significant differences between the objects. On the other hand, in the Flex-cell case study, two schemas are required since the two robotic arms present belong to different classes.

Second, the *Twin Configuration files* contain the endpoint-specific information the twins need to get access to their corresponding endpoints, such as for example, IP addresses, topics, path to source files, credentials, etc. One configuration file is required per twin in the DT-enabled system.

Third and last, the *Twin System Configuration file* provides the structure of the internal coupling of the twin system so Maestro can interpret it. One Twin System Configuration file is required per twin system featuring a DT system in the DT-enabled system. This file is not needed for twin systems featuring PT

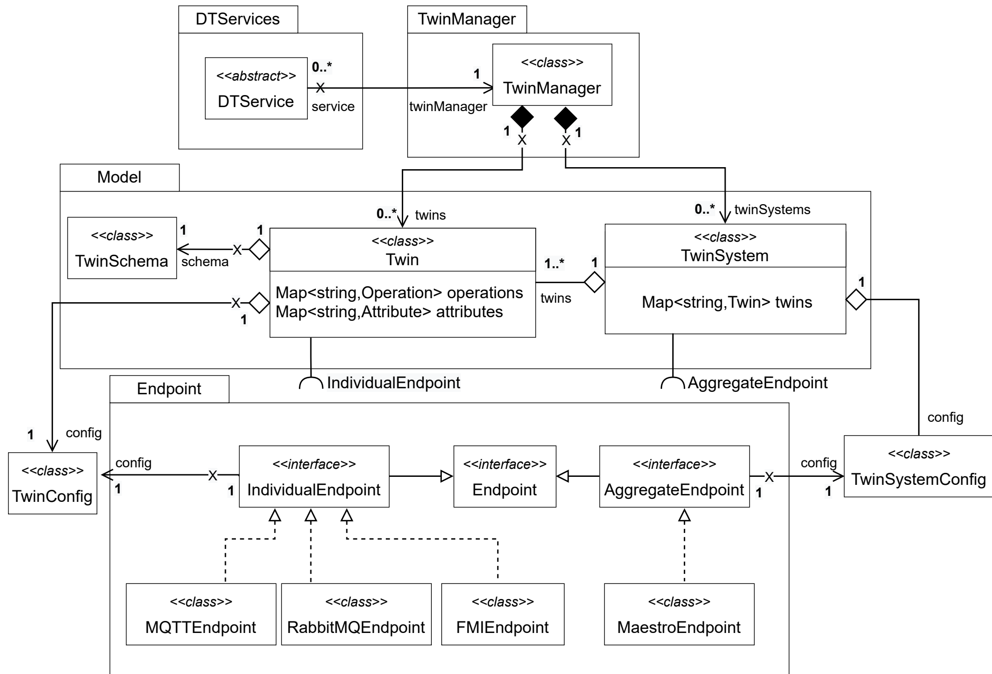


Fig. 5: UML class diagram for the architecture for Twin systems with coupled behavior.

systems. This file includes the fields for *FMUs*, *connections*, *parameters*, *logVariables*, *algorithm*, and *endtime*. The *algorithm* and *endtime* fields are specific to Maestro and refer to the step size and the step type. The *FMUs* field is where the FMU models of the smaller simulation composites that compose the larger simulation are defined. This field can also be obtained from the *Twin Configuration file* when the individual DTs use a `FMIEndpoint` specialization. The *parameters* field is used to initialize the values of the simulation. And the *logVariables* field is used to define which variables are to be logged in the output file. Finally, we propose two additional fields that are independent of the co-simulation arguments, which are the *aliases* and *rabbitmq* fields. The *aliases* field is used to map the name of variables in the co-simulation structure to the one used by the Twin Manager. The *rabbitmq* field is used to initialize the connectivity with RabbitMQ when RMQ FMU is used for the co-simulation setting.

3.2 Reusable DT services for semantic lifting

Semantic lifting is the process of generating a KG from a piece of software using some predetermined mapping from the software state to the KG. Semantic reflection is the consequent use of this lifted state from the very same software. The

generated KG is expressed in terms of the software state it originates from – in the case of a lifted program state, this would be variables, memory locations, etc.

For the system presented here, semantic lifting is the generation of a KG in terms of the architecture. In contrast to language-based lifting, the mapping is determined by the *architecture*, not the programming language. Nonetheless, it is predetermined and must not be designed anew for a new application.

The service is decoupled from the running system and has its own lifecycle. The lifting service can be reconfigured to adopt to new requirements, such as new defect queries. A reconfiguration of the DTs themselves does not require adaptation of the lifting service. Indeed, the structure of couplings itself, can be expressed using configuration files for the DTs and the DT systems, the latter representing extended co-simulation scenarios [37], which enables the configuration of the connections without using the KG initially.

3.2.1 Semantic Lifting of Architectures

Given an architecture, we need to define the mapping from a state of a piece of software implementing this architecture to a KG. We do so in two steps. First, we fix the vocabulary for the *software* (and basic axioms) using the OWL ontology language. Second, we define the mapping from a *state* into graph data using this vocabulary.

An architecture for our purpose is a set of (a) classes with attributes, that may refer to other classes, and (b) connections between those classes. Those connections have some cardinality and can have different kinds, e.g., composed-of relations, or be explicitly labeled with names. The software implementing the architecture, in turn, has further classes and connections (realized by class fields) between them[1].

There are two ways to define an ontology for such systems. The first is using an automatic, general scheme from diagrams in the architecture description language to ontologies. The second is a tailor-made ontology with a specific purpose in mind. In this work, we describe a tailor-made ontology: No general, syntactic transformations between UML and OWL (or other pairs of architecture description and knowledge representation languages) that covers all features are available [65]. Thus, we employ a manual transformation.

We describe the ontology and the lifting of a state concretely further below when describing the lifting service. Instead, we turn our attention to the notion of *consistency*, and introduce some notation precisely describe it.

We denote an architecture with $\mathcal{A}$, and any software application implementing it with $\mathcal{S}_\mathcal{A}$. A software application can be executed from some initial state $\sigma_0^{\mathcal{S}_\mathcal{A}}$, and we denote the sequence of the states during execution with $\rho^{\mathcal{S}_\mathcal{A}} = \langle \sigma_0^{\mathcal{S}_\mathcal{A}}, \sigma_1^{\mathcal{S}_\mathcal{A}} \dots \sigma_i^{\mathcal{S}_\mathcal{A}}, \dots \rangle$. We denote the set of indices $\{0, \dots, i, \dots\}$ in ρ with I_ρ. A KG is a pair of vertices and labeled edges $\mathcal{K} = (V_\mathcal{K}, E_\mathcal{E})$. A lifting for an architecture

[1] We restrict ourselves to object-oriented programming here

$\mathcal{A}$ is a map μ from software application state (of a software $\mathcal{S}_\mathcal{A}$ implementing $\mathcal{A}$) to a KG.

Consistency can be understood in general terms, e.g., that the generated KG is logically consistent w.r.t. its ontology. Indeed, this property is required for a semantic lifting map. More specific is the consistency w.r.t. additional constraints, such as queries that express consistency conditions: such a *defect query* returns violations of what is considered consistency in a given system [46]. The advantage is a more fine-grained control[2] over the notion of consistency, and that every detected defect is retrieved with information where in the system it occurs.

Let $n \in \mathbb{N}$. An n-ary defect query q over a KG $\mathcal{K}$ returns an answer set $[\![q, \mathcal{K}]\!] \subseteq V_\mathcal{K}^n$, where each answer contains n nodes that together indicate a defect. We say that a software $\mathcal{S}_\mathcal{A}$ is strongly consistent w.r.t. a set of defect queries Q, if for each run $\rho^{\mathcal{S}_\mathcal{A}} = \langle \sigma_0^{\mathcal{S}_\mathcal{A}}, \dots \sigma_i^{\mathcal{S}_\mathcal{A}}, \dots \rangle$, each lifted state returns an empty set for all defect queries.

$$\forall i \in I_\rho.\ \forall q \in Q.\ [\![q, \mu(\sigma_j^{\mathcal{S}_\mathcal{A}})]\!] = \emptyset$$

Strong consistency, just as its verification counterpart, strong invariants, is too restrictive. During adaptation, the system may have intermediate steps where it inhibits a defect – for example, when fixing a defect requires multiple steps. For this reason, we use weak consistency, where we only require defect-freedom at certain places. We say that a software $\mathcal{S}_\mathcal{A}$ is weakly consistent w.r.t. a set of defect queries Q, and a function W from index sets to a subset of the input set, if for each run $\rho^{\mathcal{S}_\mathcal{A}} = \langle \sigma_0^{\mathcal{S}_\mathcal{A}}, \dots \sigma_i^{\mathcal{S}_\mathcal{A}}, \dots \rangle$, each lifted state returns an empty set for all defect queries on $W(I_\rho)$.

$$\forall i \in W(I_\rho).\ \forall q \in Q.\ [\![q, \mu(\sigma_j^{\mathcal{S}_\mathcal{A}})]\!] = \emptyset$$

In the following we present the services needed to realize dynamic checking of weak consistency and generating reports using reflection, i.e., the services operating on $\mu(\sigma)$ are running in the software that is currently in state σ.

3.2.2 Services Supporting Semantic Reflection

The platform layer is supported by a service layer, composed of microservices that offer reusable services on the Twin Manager. The services enable the reuse of their functionalities in further contexts. To monitor the consistency of the platform, we use this layer and provide four services, where under consistency we understand the absence of *defects* on twins for DTs. A defect is defined as the result of a *defect query* and a defect query is a query over the platform layer that must return an empty set – it expresses the presence of inconsistencies. To enable this, we provide four different reusable services.

[2]One can also use additional axioms to express such constraints. Queries have the additional advantage of not necessarily requiring expensive logical reasoning.

Lifting Service. The lifting service lifts the platform layer, as well as the endpoints, to a KG. This KG has a fixed ontology, but the service can also be used to combine it with external KGs.
Query Service. The query service offers query access to the lifted state, to retrieve all answers to a SPARQL graph query or an OWL membership query.
Monitor Service. The monitoring service uses the query services to monitor internal and external consistency using defect queries.
Defect Analysis Service. While the monitoring service uses rather lightweight queries for monitoring and only reports that an inconsistency has been detected, the analysis service uses this to retrieve further information about the exact nature of the defect.

In terms of the above formalization, the lifting service computes $\mu(\sigma)$, the query service computes $[\![q, \mu(\sigma)]\!]$ for a given query q, the monitor service manages the set of defect queries Q and computes $\{[\![q, \mu(\sigma)]\!] \mid q \in Q\}$, and the defect analysis service then builds on that set and asks more details queries to act and report on the defects.

Lifting Service

The semantic lifting service maps the endpoints and the DTs in the DT platform layer, into a KG. It is not necessary to lift the static parts and the used configuration files directly, as the aim of the lifting service is to provide a means to reflect on the instantiated system.

The ontology used in the KG formalizes the structures needed for reflection. Figure 6 illustrates it in Manchester syntax [66], while Figure 7 shows a visual overview. The upper part of Figure 6 shows the concepts of the ontology and their subclass relation. A `DTObject` is either a `SimulationComponent`, i.e., an FMU, or a `ContainerComponent`, used to structure composed DTs by introducing hierarchy and encapsulation. In terms of the class diagram in Figure 5, a `SimulationComponent` is twin using an FMIEndpoint and a `ContainerComponent` is a twin using a MaestroEndpoint, which represents the hierarchy of composed twins. Furthermore, we have `Ports`, which can be either `InPorts` for inputs, or `OutPorts` for outputs and `Connections` between ports. The classes `AliasedPort` and `AliasingPort` are defined via the `aliasOf` property below. They are modeled as disjoint, expressing that one cannot alias a port that is already an alias. Aliases are needed because variable names amd instance names may differ between simulation models, the PT system interface, and communication channels.

The relations are defined in the lower part of Figure 6. To model that a port is an alias of another, the `aliasOf` property is used. Note that contrary to the presentation in the configuration file, the relation is defined over ports, not strings, i.e., to generate the lifting, all connections are configured and the content of the String configuring them is resolved. This property is irreflexive – a port cannot be an alias for itself. Connections are related to their ports via the `connectFrom` and `connectTo` properties, while the hierarchy induced by `ContainerComponents` is

```
Class: DTObject  DisjointUnionOf: ContainerComponent, SimulationComponent
Class: Port      DisjointUnionOf: InPort, OutPort
Class: Connection
Class: ContainerComponent   SubClassOf: DTObject
Class: SimulationComponent  SubClassOf: DTObject
Class: InPort               SubClassOf: Port
Class: OutPort              SubClassOf: Port
Class: AliasedPort EquivalentTo: inverse (aliasOf) some owl:Thing
    DisjointWith: AliasingPort
Class: AliasingPort EquivalentTo: aliasOf some owl:Thing
    DisjointWith: AliasedPort
```

```
ObjectProperty: aliasOf
    Characteristics: Irreflexive
    Domain: Port
    Range: Port

ObjectProperty: connectFrom
    Characteristics: Functional
    Domain: Connection
    Range: OutPort

ObjectProperty: connectTo
    Characteristics: Functional
    Domain: Connection
    Range: InPort

ObjectProperty: contains
    Domain: ContainerComponent
    Range: DTObject
```

```
ObjectProperty: hasPort
    Characteristics: InverseFunctional
    Domain: DTObject
    Range: Port

DataProperty: hasDescriptor
    Domain: DTObject
    Range: xsd:string

DataProperty: hasFile
    Domain: SimulationComponent
    Range: xsd:string

DataProperty: hasName
    Domain: DTObject or Port
    Range: xsd:string
```

Fig. 6: Textual representation of the ontology used for semantic lifting of the platform layer.

expressed using `contains`. Any `DTObject` can have ports related to the `hasPort` property. The data properties related components to their name, description or the name of the file they were loaded from.

For the lifting, the service iterates over all DTs, lifts their endpoints as `SimulationComponents` and the DTs themselves as `ContainerComponents`. Their connections are modeled using `Connections` (note that a connection can be between two `ContainerComponent` as well), and the remaining properties follow directly from the structure. The lifting service offers the following operations:

- `getModel` returns a KG for the current state.
- `getModelCombined` takes a KG and returns the lifted state combined with the additional KG.

The latter is necessary to query the relation of the lifted state to external knowledge, e.g., an asset model describing the PT structure.

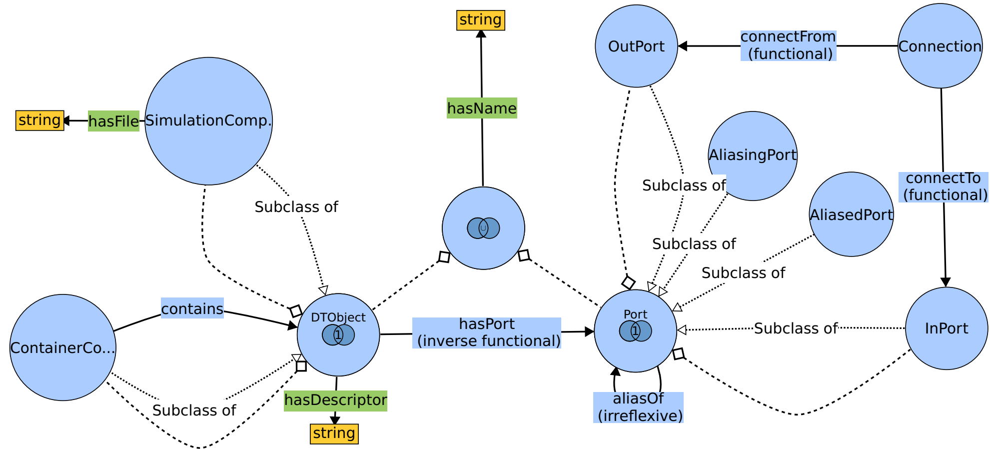


Fig. 7: Visual representation of the ontology used for semantic lifting of the platform layer, created with WebVOWL [67].

Query Service

The *Query Service* takes as input a query, executes it on the lifted state and returns the results. We assume that the service supports at least SPARQL queries, but it can also provide access through other query languages. For example, it may take reasoning tasks, such as an OWL axiom, and returns whether it can be derived from the lifted state, or answer DL membership queries, i.e., return all members described by a DL concept. A simple reasoning task would be the consistency of the lifted state – if a port is an alias of another alias, then the KG is inconsistent because of the axiom that `AliasingPort` and `AliasedPort` are disjoint.

Membership and SPARQL queries retrieve individuals within the platform and can be used to extract specific structures. For example, the SPARQL query in Listing 1 returns all FMUs whose value is aliased to a port named `outFlow` by the containing DT. This removes the need to navigate the internal structure manually and can be used by external users without knowledge about the exact model description of the FMU and without manually traversing the platform's structure. The example is using the Three-Tank system example introduced in Section 2.2.1.

```
SELECT ?fmu {
    ?dtc a ContainerComponent;  hasPort ?po1.
    ?fmu a SimulationComponent; hasPort ?po2.
    ?po1 aliasOf ?po2;          hasName "outFlow".
}
```

Listing 1: Query for FMUs whose port is aliased to "outFlow".

Building on the query service, we build the tools that we need to ensure consistency using reusable components. It offers a single operation: `query` takes a query/reasoning task and returns a result set.

Monitor Service

The *Monitor Service* is used to detect defects in the lifted state. It is configured using a set of SPARQL queries, where each such query corresponds to a defect and consistency is defined as the absence of any (detected) defect. Internal defects are detected by queries related to the lifted state alone.

For example, the SPARQL query in Listing 2 returns all input ports of FMUs that are not connected to any port, and have the name "tank4Outflow". A non-empty result set may indicate that the connections are not set up correctly.

```
SELECT ?fmu {
    ?port a InPort; a AliasingPort; hasName "tank4Outflow".
    ?fmu hasPort ?port.
    FILTER NOT EXISTS { _ connectTo ?port }
}
```

Listing 2: Query for monitoring disconnected ports of the FMUs.

External defects are detected by queries related to the lifted state in combination with external KGs. For example, an external KG can be the asset information model that describes the actual structure of the PT Another examples would be general knowledge, in form of an ontology, about the relations between DT components. For example, the SPARQL query in Listing 3 returns all tanks (modeled as `Tank`) whose simulator (modeled as `hasSimulator`) is not instantiated. This may indicate that the DT is incomplete [46].

```
SELECT ?tank {
    ?tank a Tank; hasSimulator ?path.
    FILTER NOT EXISTS { _ hasFile ?path. }
}
```

Listing 3: Query for defect detection.

The monitor service depends on the query service and offers the following operations:

- `setDefectQueries` takes a set of SPARQL and reasoning queries and is used to configure the service used to detect defects,
- `getViolations` returns the results of all queries configured before, together with the id of the query (per result), and
- `getViolationsRegular` automatically returns the defects every n time units.

The monitoring service, while used for consistency monitoring in our setting, is reusable for any kind of runtime monitoring based on regular queries.

Defect queries can be generic for all instantiations of the architecture, and can consequently be reused for all of them. We denote such defect queries as *generic*. Consider the generic query **Q0a** in Fig. 8. It detects the defect that a `DTObject` has an `InPort` that is aliased by an `OutPort` or vice versa. Query **Q0b** detects if two objects have the same name and query **Q0c** ensures that there are no cycles in the `contains` relation, i.e., a `ContainerComponent` does not contain itself.

```
SELECT ?obj ?port ?alias { #Q0a
    ?obj hasPort ?port. ?port a Port.
    ?alias a Port; aliasOf* ?port.
    {?port a InputPort. ?alias a OutputPort} UNION
    {?port a OutputPort. ?alias a InputPort}
}
```

```
SELECT ?obj1 ?obj2 ?name { #Q0b
    ?obj1 a DTObject; hasDescriptor ?name.
    ?obj2 a DTObject; hasDescriptor ?name.
}
```

```
SELECT ?obj  { ?obj contains* ?obj. } #Q0c
```

Fig. 8: Generic query **Q0a** for alias defect detection, generic query **Q0b** for unique descriptors and generic query **Q0c** for cycle-free containers.

Defect Analysis Service

The *Analysis Service* is tasked with a more complex task, building on the results of the monitoring service: Given a detected defect, it executes a query to retrieve enough information about the defect for a detailed report about the circumstances, and possible information needed to repair the defect.

For each SPARQL or OWL concept membership query, the service offers an operation that takes the result of the query (i.e., a detected defect) and executes a SPARQL query to retrieve further information. In particular, it will execute a more complex, and possibly more time-consuming query. For example, consider the monitoring query from Listing 3. To retrieve the exact information about the missing tank, one can ask for the next tank further down in the pipeline, as shown in Listing 4. Note that the connection, modeled with `connectedOutFlow`, may involve logical reasoning.

```
SELECT ?tank ?tank2 {
    ?tank a Tank; hasSimulator ?path.
    ?tank connectedOutFlow ?tank2.
    ?tank2 a Tank; hasSimulator ?path2.
    FILTER NOT EXISTS { _ hasFile ?path. }
}
```

Listing 4: Query for pipeline defect detection.

The relation between a monitor query and the analysis query is encapsulated in a `DefectHandler`, which contains both queries. The defect analysis service depends on the monitor and the query services and offers the following operations:

- `addDefectHandler` takes a `DefectHandler` as input and adds it to the internal list of handlers;
- `registerDefectHandlers` configures the used monitoring service to use the monitor queries of the defect handlers of this instance;

- `getReports` gets all consistency violations from the monitor services, executes the corresponding defect handler analysis query, and returns the set of all results of the analysis queries; and
- `getReportsRegular` returns the reports every n time units.

We stress that these services are part of the *platform*, not the *DT* itself. Consequently, their configuration with external KGs and defect queries follows a different life cycle: The services are developed, maintained, and deployed independently of the system and can be reused for different instances, while their configuration can react to both design requirements of the twin, as well as to more short-term effects, such as additional requirements added while the system is running.

4 Results

4.1 Instantiation of the Architecture

Following the multi-case study approach of this study, we first showcase the use of the architecture for the two previously introduced case studies in Section 2.2 to demonstrate its theoretical generalizability.

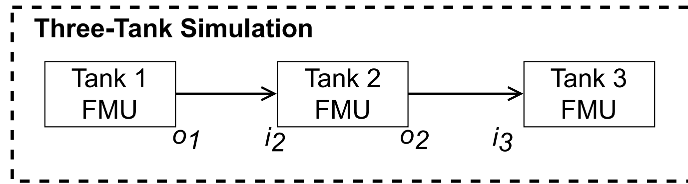
Three-Tank System

The instantiation of the Three-Tank System is illustrated in Figure 9. The simulation block in Figure 9a, named *Three-Tank Simulation*, represents the coupled behavior of a generic Three-Tank PT system with a cascade coupled behavior. This instantiation, which is represented by a UML object diagram (Figure 9b), intends to ground the abstraction for the Three-Tank System co-simulation (Figure 9a). Notice that this example does not count with a PT in the instantiation of the architecture due to its simulation-only feature. Therefore, the Three-Tank System example behaves as a DM (or more precisely, as a pre-DT [19]).

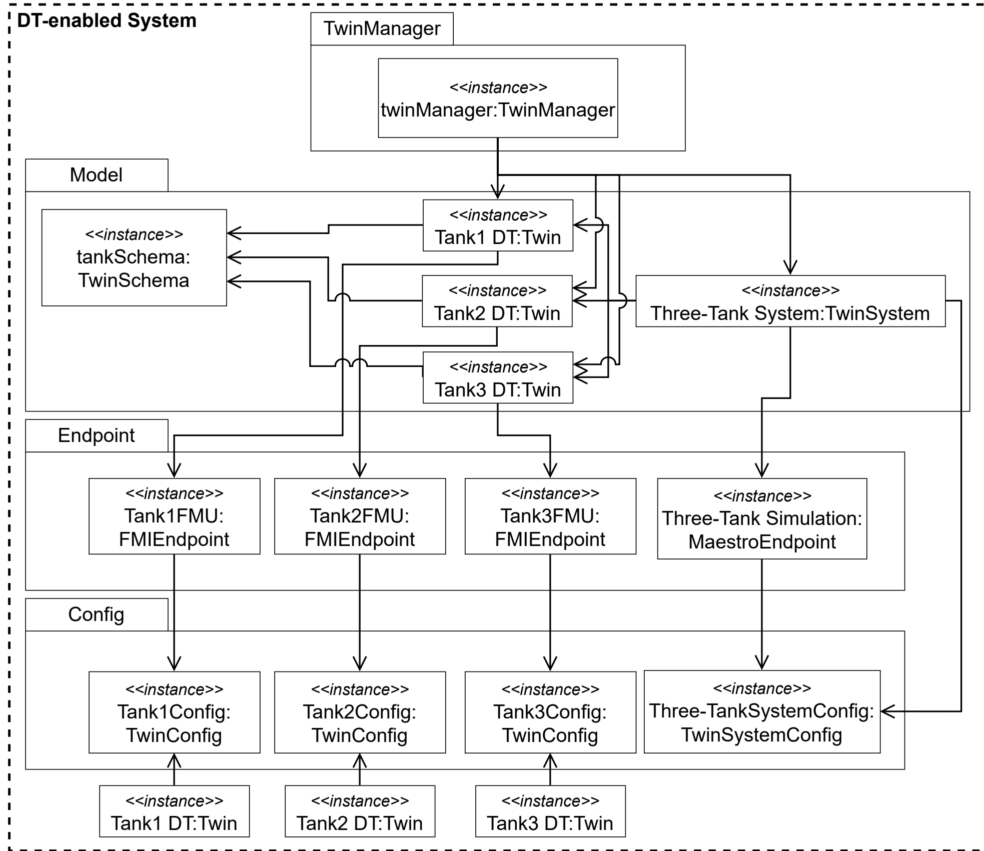
We start by defining the three individual tanks as objects of the `Twin` class. The individual twins are initialized from the same *Twin Schema file* since they all have the same properties and are connected via `FMIEndpoint` endpoints, i.e., each tank can be managed independently using its FMI endpoint.

Subsequently, we proceed with the extension to support the coupled simulation of the DT system. We instantiate an object of the `TwinSystem` class, with the name *"Three-Tank System"*. The three twins featuring the independent tanks are aggregated to the DT system and the system is bound to a `MaestroEndpoint` endpoint.

The `MaestroEndpoint` endpoint requires the *Twin System Configuration file* for its initialization; then, we provide such a file, which contains the definition of the internal FMUs, connections, and algorithm settings. The most relevant field is the *connections* field, which represents the coupling of the system. The connections for this particular DT system are described in Listing 5. This configuration



(a) Co-simulation block diagram.



(b) UML object diagram.

Fig. 9: Diagrams to realize the DT-enabled system for the Three-Tank System case study. The direction of arrows in (b) indicate the use of the pointed instance by the pointing instance.

means that the output of *tank1* goes to the input of *tank2*, and the output of *tank2* goes to the input of *tank3*. Maestro interprets these connections to perform the co-simulation, obtaining the simulation results of the coupled system.

```
1 "connections": {
```

```
2     "{tank}.tank1.outPort": ["{tank}.tank2.inPort"],
3     "{tank}.tank2.outPort": ["{tank}.tank3.inPort"]
4 },
```

Listing 5: *Connections* field for the three-tank DT system configuration with Maestro.

Flex-cell

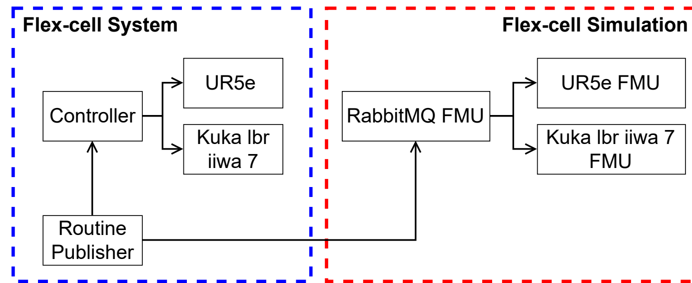
The instantiation of the Flex-cell is illustrated in Figure 10. The simulation block in Figure 10a, named *Flex-cell Simulation*, represents the hierarchical composition of the simulation composites with coupled behavior by synchronization since both robotic arms are intended to work cooperatively and simultaneously. This instantiation, which is represented by a UML object diagram (Figure 10b), intends to ground the abstractions of the Flex-cell physical setup and co-simulation (Figure 10a). Notice that the controllers in Figure 10a, i.e., the physical controller and RMQ FMU, are considered part of the DT-enabled system, but they are not represented as twins.

We start by defining the robots as objects of the `Twin` class. Since we consider the PTs in this case, there are four objects of the Twin class in total, two for the simulated robots and two for the real robots. Additionally, as the two robotic arms are different, we use two different *Twin Schema files*, one representing the UR5e robot and the other representing the Kuka lbr iiwa 7 robot. The individual twins referring to the simulation are initialized using `FMIEndpoint` endpoints and the individual twins referring to the PT are initialized using `MQTTEndpoint` endpoints.

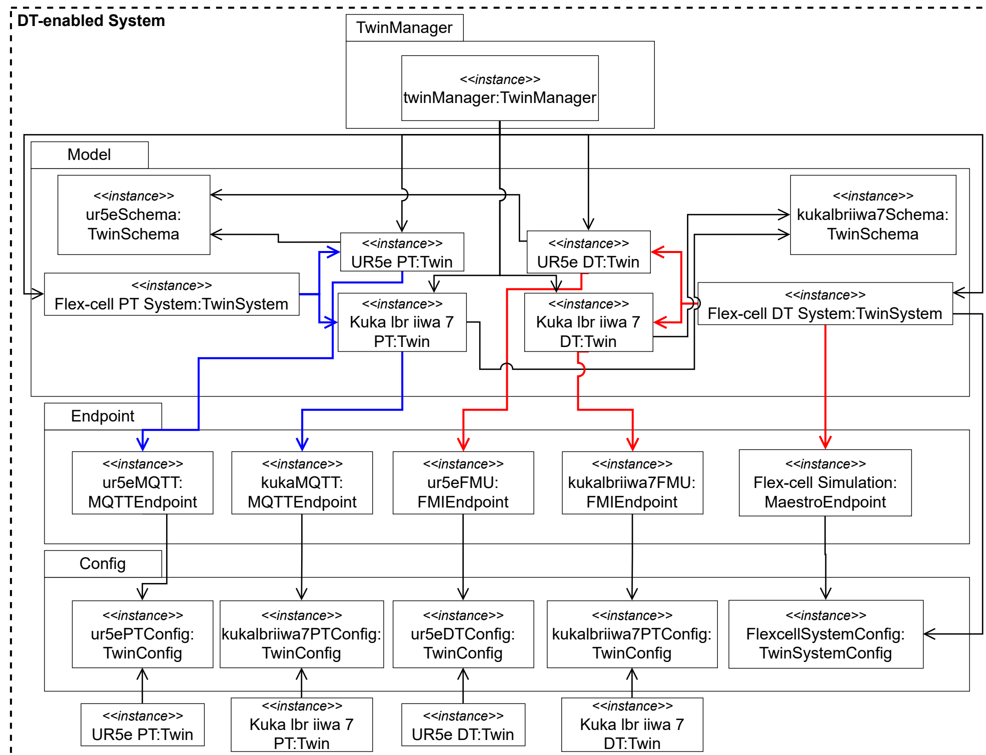
Subsequently, we proceed with instantiating two objects of the `TwinSystem` class, one for the Flex-cell PT system and one for the Flex-cell DT system. The Flex-cell PT system with name *"Flexcell PT System"* aggregates the two individual twins featuring the real robots. Similarly, the Flex-cell DT system with name *"Flexcell DT System"* aggregates the two individual twins featuring the simulated robots. In this case, we bind the twin system featuring the DT part to a `MaestroEndpoint` endpoint. The twin system featuring the PT part simply adds access to the individual twins from a hierarchical view since the coupled behavior is already existing in the PT intrinsically.

The `MaestroEndpoint` endpoint requires the *Twin System Configuration file* for its initialization; then, we provide such a file, which contains the definition of the internal fields. The *connections* field in this case represents the coupling of the system, as described in Listing 6. This configuration means that the output of *RMQ FMU* for the *discretized target positions* and *motion time* goes simultaneously to the inputs of *UR5e FMU* and *Kuka lbr iiwa 7 FMU*, thus, it replicates the physical setup in the simulation part. This way, Maestro can execute co-simulation experiments that represent the cooperative execution of both robots.

Since the co-simulation uses RMQ FMU, Maestro runs an asynchronous co-simulation based on the inputs from RMQ FMU, which need to be passed at specific time steps. These inputs are inputted at the same time the physical



(a) Co-simulation block diagram (red, right) mapping the physical setup (blue, left).



(b) UML object diagram.

Fig. 10: Diagrams to realize the DT-enabled system for the Flex-cell case study. The arrows in blue in (b) indicate the PT-related relations. The arrows in red in (b) indicate DT- and co-simulation-related relations. The direction of arrows in (b) indicate the use of the pointed instance by the pointing instance.

controller sends the commands to the real robots (illustrated by the *Routine*

Publisher block in Figure 10a), enabling the synchronization, and therefore, comparison between DT and PT systems. As a plus, the co-simulation can also be executed stand-alone, i.e., decoupled from the execution of the real process, so the results of the co-simulation can be used for virtual commissioning. With this feature, this example behaves as a DM (or pre-DT) since the PT is decoupled from the execution.

```
1 "connections": {
2     "{rabbitmq}.rabbitmq.target_X_ur5e": ["{ur5e}.ur5e.target_X"],
3     "{rabbitmq}.rabbitmq.target_Y_ur5e": ["{ur5e}.ur5e.target_Y"],
4     "{rabbitmq}.rabbitmq.target_Z_ur5e": ["{ur5e}.ur5e.target_Z"],
5     "{rabbitmq}.rabbitmq.motion_time_ur5e": ["{ur5e}.ur5e.motion_time"],
6     "{rabbitmq}.rabbitmq.target_X_kuka": ["{kuka}.kuka.target_X"],
7     "{rabbitmq}.rabbitmq.target_Y_kuka": ["{kuka}.kuka.target_Y"],
8     "{rabbitmq}.rabbitmq.target_Z_kuka": ["{kuka}.kuka.target_Z"],
9     "{rabbitmq}.rabbitmq.motion_time_kuka": ["{kuka}.kuka.motion_time"]
10 },
```

Listing 6: *Connections* field for the Flex-cell DT System configuration with Maestro and RMQ FMU.

Along with the instantiation of the architecture for the Flex-cell case study, the experiment executes a cooperative motion in the real robots (PT) and co-simulation (DT) simultaneously to showcase the applicability of the architecture for systems with coupled behavior due to synchronization.

The experiment consists of moving the robots simultaneously to two different discrete position on the Flex-cell working space, namely, the positions (1) $(X, Y, Z) = (6, 16, 1)$ and $(X, Y, Z) = (8, 11, 2)$, and (2) $(X, Y, Z) = (5, 22, 0)$ and $(X, Y, Z) = (7, 14, 1)$, for the UR5e and the Kuka lbr iiwa 7 respectively. The second command is sent 5.0 seconds after the first command is sent.

Figure 11 shows the joint positions over time for the two cooperative execution in the real robots (below) and simulated robots (above), which is the result of the experiment. It is possible to see that all real robots and simulated robots behave accordingly as expected. Notice that there are time offsets for the real robots compared to the simulated ones due to motion speed and time delays when running the commands. Additionally, the co-simulation can introduce additional delays due to (i) using discrete time steps instead of continuous time (especially evident for larger step sizes) and (ii) processing of messages coming from RMQ FMU.

4.2 Mapping to the DTaaS Platform Implementation

One of the goals of this architecture is to be sufficiently generalizable to provide and use reusable components that are shared in a more heterogeneous software environment that further boosts the reusability to realize DTs with less implementation effort. To do so, we propose the integration of this approach with the DTaaS platform proposed by Talasila et al. [5], which is designed to reuse DT services and components, as well as tools and infrastructure services, to facilitate the realization of DTs and the execution of multiple DT instances for different

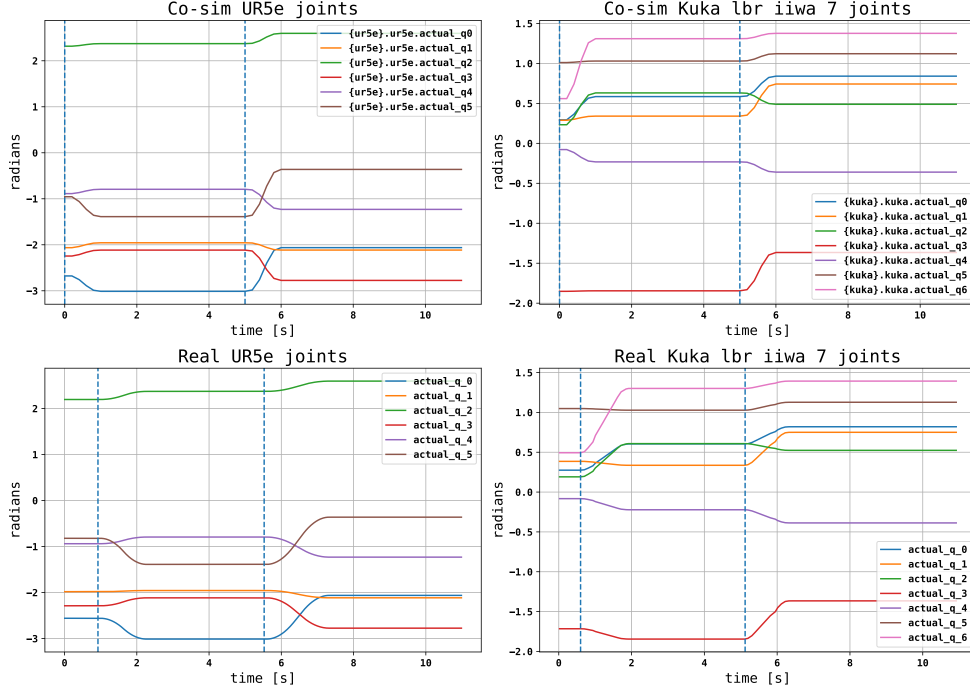


Fig. 11: Plot of the joint positions of the robotic arms running the commands synchronously with the co-simulation setting. The vertical dashed lines indicate when the commands are sent (precise for the co-simulation; approximate for the real robots).

applications. Therefore, we map the components of the architectural approach to the definitions of the DTaaS platform (see Section 2.1.1), i.e., to *Models*, *Tools*, *Data*, and *Functions*, and *DT configurations*.

In relation to our proposed architecture, the DTaaS platform handles the Twin Manager as a *Tool*, the FMUs and twin schemas as *Models*, and the twin configurations and Twin System Configuration configurations as *DT configurations*. Similarly, Maestro and RMQ FMU, which are used by the Twin Manager for the co-simulation settings, are also available by default in the DTaaS platform, the former as a *Tool* and the latter as a *Model*. Therefore, it is feasible to map the implementation of the architecture into the DTaaS platform implementation to further boost the reuse of DT services and components.

Three-Tank System

We map the components defined in Figure 9 to assets of the DTaaS platform. First, regarding the models, the DTaaS platform stores FMUs as *Models*; these FMUs are treated as binary files which are used by co-simulation toolchains. The twin

schema file in AAS is stored as *Model* too. The tank configurations and three-tank system configuration are stored as *DT configuration files*. The Twin Manager is packaged as one Java jar file and handled as a *Tool*. For the execution of the example, a script handled on top of the DTaaS brings together the Twin Manager jar package along with the DT-enabled system, the tank FMUs, twin schema and twin configuration files to run the three-tank application. The complete case study is then executed by the DTaaS platform to produce the results of the (pre-DT) DM, i.e., the co-simulation results, since no PT is used. The example of the DT for the Three-Tank System on the DTaaS platform is publicly available on GitHub[3] .

Flex-cell

We proceed similarly for the Flex-cell case study. We map the components defined in Figure 10 to assets of the DTaaS platform. First, regarding the models, the DTaaS platform stores the UR5e FMU and Kuka lbr iiwa 7 FMU as *Models*; these FMUs are treated as binary files which are used by co-simulation toolchains. The twin configurations for the real and simulated robots and Flex-cell DT system configuration are stored as *DT configuration files*. In addition, both robots have their twin schema files as AAS representations as *Models*.

The Twin Manager jar package, which was used in the DT application of the Three-Tank System, as well as Maestro and RMQ FMU (these two provided by default by the DTaaS platform), have been reused for this case study, thus demonstrating the benefits of this approach in terms of reusability when developing DTs. The DTaaS helps setting up the example and its dependencies and an execution script brings together the components and dependencies, including the twin and twin system configuration files, twin schemas, FMUs, channel credentials, and co-simulation setting, to run the DT-enabled system. The complete case study is then executed by the DTaaS platform to produce the results of the DT-enabled system as (i) a proper DT when the PT is bound to the platform during execution or (ii) as a (pre-DT) DM when the PT is decoupled from the platform to run simulation and *what-if* experiments. The example of the DT for the Flex-cell on the DTaaS platform is publicly available on GitHub[4].

4.3 Applying Semantic Lifting Services

As part of our evaluation experiment we use the defect analysis service to monitor and report while both case studies are executed. In either case, we set up the system such that an error is injected after some time and evaluate the reusability of implementing the detection of this error. The services are fully reusable as they are not concerned with any specifics of the instantiation of the architecture.

Instead, the configuration happens in the defect queries. There are two sets of defect queries: the set of generic defect queries, and the set of application-specific defect queries. In the implementation, a defect SPARQL query and a

[3] https://github.com/INTO-CPS-Association/DTaaS-examples/tree/main/digital_twins/three-tank
[4] https://github.com/INTO-CPS-Association/DTaaS-examples/tree/main/digital_twins/flex-cell

function that generates a report specific to a found defect are handled as a unit (i.e., the services work not on sets of SPARQL queries, but on sets of pairs of a SPARQL query and a function.)

We run two different queries for each of the two systems. They do not only reflect on the internal structure, as the simple queries in Section 3, but also use the asset model that describes the physical system to ensure further consistency properties. The asset model in our case also specifies the requirements on single ports of simulators in the DTs, thus realizing a form of runtime monitoring. The queries used for report generation are given in Figure 12.

- Query **Q1a** returns all DTs `?x` loading the linear model of a tank that are twinning some tank `?asset` that flows into another tank `?next`. It then removes all the DTs, for which `?next` is correctly connected. Thus, the result of the query is all those DTs that are not correctly coupled since all correctly coupled connections are removed.
- Query **Q2a** returns all DTs `?x` of tanks and the current value `?out` of their outflow, which do not satisfy the requirement for their PT. The requirement expresses that the value must be over a certain threshold `?lim`. The value, and the requirement itself are part of the asset model and connected to twinned asset `?asset` with the `domain:specifiedBy` property.

Query **Q1b** for the Flex-cell is similar to **Q1a** and checks that the ports of the RabbitMQ on one side, and the UR5 and Kuka lbr iiwa 7 simulators are connected correctly. There is no explicit asset model, this property is internal to the system. Query **Q2b** is analogous to **Q2a** but targets the coordinates of the `X` axis in the Flex-cell. For the sake of brevity, they are only given in the auxiliary online material. A generic query, used in both case studies, is given in Figure 8.

The following is the output of a report for two defects found by **Q2a**. The string `outPortLimit` is the internal name of the query. Each report is, in this case, a single line generated by the parameterized string “`simulator $?x has value $?out >= $?lim !`”.

```
Report on "outPortLimit". The following simulators exhibit defects:
-----------------------------------------------------------------
    simulator {tank}.tank3 has value 11 >= 10!
    simulator {tank}.tank1 has value 12 >= 10!
```

5 Evaluation

The approach we have described so far enables the realization of hierarchical DTs, and the value that can be added to these through semantic services. In this section, we are further assessing the reusability of the approach using the multi-case study setting used throughout this work.

```
1  SELECT ?x  {?x a domain:SimulationComponent;  #Q1a
2                domain:hasFile "DTProject/fmus/Linear.fmu";
3                domain:hasName ?id.
4             ?asset domain:twinnedWithName ?id;
5                    domain:flowsInto ?next.
6             ?next domain:twinnedWithName ?idNext].
7             FILTER NOT EXISTS {
8                 ?y a domain:SimulationComponent; domain:hasName ?id.
9                 ?cont a domain:ContainerComponent;
10                      domain:contains ?x;
11                      domain:contains ?y;
12                      domain:hasConnection [
13                            domain:connectFrom [domain:aliasOf [domain:hasName "outPort"]];
14                            domain:connectTo [domain:aliasOf [domain:hasName "inPort"]]].
15                    }
16 }
```

```
1  SELECT ?x ?out ?lim {?x a domain:SimulationComponent;  #Q2a
2                         domain:hasFile "DTProject/fmus/Linear.fmu";
3                         domain:hasPort ?p;
4                         domain:hasName ?id.
5                      ?asset domain:twinnedWithName ?id;
6                             domain:specifiedBy [domain:minValue ?lim].
7                      ?p a domain:OutPort;
8                         domain:hasName "outPort";
9                         domain:hasValue ?out.
10                     FILTER ( ?out >= ?lim )
11 }
```

Fig. 12: Queries **Q1a** (above) and **Q2a** (below) for the three-tank system of **Ex2**.

5.1 Metrics and Settings

In alignment with **RQ1** and the IEEE 1517 Standard for systematic reuse [68], some metrics to evaluate the reusability of the proposed architecture are collected. Although some approaches, such as Ghasemi et al. [69] and Lia and Colella [70] have proposed strategies to measure reusability using DTs, there are no standard metrics to collect such data. The former measures the reusability using the number of methods and the latter measures the principles for findability, accessibility, interoperability, and reusability using dictionary-like data, such as JSON, XML, and RDF files. However, these methods do not fit this approach's scope to measure reusability.

Therefore, we use the multi-case study approach with the two case studies considered in this work to measure the reusability of the Twin Manager components for the two case studies. Additionally, we also use the measurement for reusability of software components proposed by Washizaki et al. [71] to measure the reusability of the Twin Manager approach as a software component since it is set up based on black-box components and a unique Façade interface to users and services. For collecting these metrics, we assume that the configuration files, twin schemas, and black-box models already exist for both case studies, and these only need to be attached to the instantiation of the architecture. Therefore, the evaluation considers the components as black boxes that can be reused,

replaced, added, or deleted. Such black boxes are based on the instantiation of the architectures in Figure 9 and Figure 10.

Metrics concerning the use and reuse of the architecture and its components

To evaluate the architecture and the defect queries separatly, each of the following metrics is used once for the architecture with and once without defect queries.

Metric 1. Ratio of components that can be reused *as-is* with light parameterization for the instantiation of second case study based on the instantiation of the first one.
Metric 2. Ratio of components to be replaced for the instantiation of the second case study based on the instantiation of the first one.
Metric 3. Ratio of components to be added for the instantiation of second case study based on the instantiation of the first one.
Metric 4. Ratio of components to be deleted for the instantiation of second case study based on the instantiation of the first one.
Metric 5. Value of the Component Overall Reusability (COR) proposed by Washizaki et al. [71] for all components .

5.2 Evaluation Results

For collecting Metrics 1 to 4, we use the instantiations of the architecture for the two case studies. Defect queries are listed separately, but are considered as configuration files.

The instantiation of the Three-Tank System in the architecture (see Figure 9) is achieved by using a total of 17 (22 with queries) components, as follows:

Architecture-specific components (9): Twin Manager, Tank 1 DT, Tank 2 DT, Tank 3 DT, Three-Tank System, FMIEndpoint Tank 1, FMIEndpoint Tank 2, FMIEndpoint Tank 3, and MaestroEndpoint.
Models (4): Tank schema, FMU Tank 1, FMU Tank 2, and FMU Tank 3.
Configuration files (4): Tank 1 Config, Tank 2 Config, Tank 3 Config, and Three-Tank System Config.
Defect Queries (5): **Q0a**,**Q0b**,**Q0c**,**Q1a**,**Q2a**

Similarly, the instantiation of the Flex-cell in the architecture (see Figure 10) is achieved by using a total of 22 (27 with queries) components, as follows:

Architecture-specific components (12): Twin Manager, UR5e DT, Kuka lbr iiwa 7 DT, UR5e PT, Kuka lbr iiwa 7 PT, Flex-cell DT System, Flex-cell PT System, FMIEndpoint UR5e, FMIEndpoint Kuka lbr iiwa 7, MQTTEndpoint UR5e, MQTTEndpoint Kuka lbr iiwa 7, and MaestroEndpoint.
Models (5): UR5e schema, Kuka lbr iiwa 7 schema, UR5e FMU, Kuka lbr iiwa 7 FMU, and RabbitMQ FMU.

Configuration files (5): UR5e PT Config, UR5e DT Config, Kuka lbr iiwa 7 PT Config, Kuka lbr iiwa 7 DT Config, and Flex-cell System Config.
Defect Queries (5): **Q0a**,**Q0b**,**Q0c**,**Q1b**,**Q2b**

In order to instantiate the Flex-cell based on the instantiation of the Three-Tank System, the models need to be replaced (since we assume models are already provided), the architecture-specific components can be either reused with light parameterization or added/deleted. Finally, the configuration files for the new setup also need to be provided.

More precisely, the components to be changed are as follows: Three FMUs need to be replaced, one twin schema needs to be replaced and one twin schema needs to be added. Two twin configuration files featuring FMIEndpoint interfaces need to be replaced and one need to be deleted; two twin configuration files featuring MQTTEndpoint interfaces need to be added; and one twin system configuration featuring the co-simulation (DT part) needs to be replaced and one featuring the PT system needs to be added. In terms of the architecture-specific components, the instances for the Twin Manager, three twins, one twin system, two FMIEndpoints, and one MaestroEndpoint can be reused with light parameterization, i.e., by initializing them to the second case study with the corresponding configuration files and models; one additional twin, one twin system, and two MQTTEndpoints need to be added; and one FMIEndpoint needs to be deleted.

This yields that to instantiate the Flex-cell case study based on the Three-Tank System instantiation, seven components need to be replaced ($replacements = 7$), seven components can be reused ($reuses = 7$), eight components need to be added ($additions = 8$), and two components need to be deleted ($deletions = 2$). To calculate the ratio, we use the formula $ratio_{metric} = \frac{metric}{total_c}$, where $total_c$ stands for the total number of components to be modified (including the deletions), i.e., 24.

We also consider the queries as components (more precisely, as configuration files, which configure the defect monitoring) to analyze their contributions to the ratios; of these, two are replaced and three are reused. These metrics are all given in in Table 1, with and without the defect queries.

The instantiation of the Flex-cell based on the Three-Tank System is drastic since it changes from one case study to another one completely different. It demonstrates reuse in an extreme case and illustrates the capabilities of the Twin Manager to generalize to multiple systems. A less drastic, yet hypothetical, reconfiguration scenario is the replacement of a robotic arm of the Flex-cell, e.g., to use two UR5e. In that case, the modifications to set up the Flex-cell with two UR5e include: the replacement of the FMU for the second UR5e instead of the one for the Kuka LBR iiwa 7 and the configuration file featuring the existing instance of the FMIEndpoint previously used by the Kuka LBR iiwa FMU; the replacement of the configuration file featuring the existing instance of the MQTTEndpoint previously used by the real Kuka LBR iiwa 7; the twin schema for the second UR5e is reused from the first UR5e; and the MaestroEndpoint is reused by replacing

the twin system configuration file of the existing co-simulation with the updated setting including the two UR5e FMUs.

Regarding Metric 5, we follow the method proposed by Washizaki et al. [71], where we compute the sub-metrics for the component's observability (RCO), customizability (RCC), and self-completeness of return's value (SCCr) and self-completeness of parameter (SCCp) to come up with the COR. For this computation, we have four settings, namely, (1) the business logic inside the Twin Schema file is considered, and the defect queries are not, (2) the business logic inside the Twin Schema file is not considered, and the defect queries are not, (3) like (1), but with the defect queries, and (4) like (2), but with the defect queries. For setting (1) and (3), we make an assumption the Twin Schema is provided with six attributes (observations) and three operations (business methods), replicating the UR5e's useful properties under the scope of this work, which is part of the Flexcell case study (see Section 2.2.2). The observations refer to the joint positions for joints zero to five and the business methods refer to the executable commands `movej`, `movel`, and `movep`. For the properties, we use the number of readable and writable properties, and number of business methods without return value and without parameters in the `TwinManager` class, which is the unique interface to the DT-enabled system, in relation to the actual existing properties and business methods in the component. For settings (3) and (4) we make the assumption that the two specific queries are provided as writable attributes, and the three generic ones as business methods (without return value), that are used by the lifting services internally.

For readable and writable properties of the Façade component, we consider that the Twin Manager's getter and setter methods are the interfaces to access the internal properties, namely, *getAttributeValue*, *getSystemAttributeValue*, *setAttributeValue*, and *setSystemAttributeValue*. Similarly, for the business methods of the Façade component, we consider that the Twin Manager's methods to execute operations are the interfaces to access the internal business methods, namely, *executeOperation* and *executeOperationOnSystem*. Internally speaking, we consider nine relevant system-wise properties in the component (*name, schema, configuration, endpoint, list of attributes, list of operations, twins, and twin systems*). Similarly, we consider four relevant system-wise internal business methods without return value (*executeOperation, simulate, doStep, and resetSimulation*) and two without parameters (*simulate and doStep*).

These data yield to the metrics in Table 1, with and without the twin schema and the defect queries.

We discuss the results in the next section.

Table 1: Summary of reusability metrics. ✓denotes that this part is considered, $\mathcal{X}$ denotes that it is not. The upper table gives metrics 1 – 4, the lower metric 5.

Queries	$ratio_{reuses}$	$ratio_{replacements}$	$ratio_{additions}$	$ratio_{deletions}$
$\mathcal{X}$	0.292	0.292	0.333	0.083
✓	0.344	0.310	0.275	0.069

Schema	Queries	RCO	RCC	SCCr	SCCp	COR
✓	$\mathcal{X}$	0.533	0.533	0.741	0.286	0.189
$\mathcal{X}$	$\mathcal{X}$	0.22	0.22	0.5	0.5	−0.120
✓	✓	0.533	0.66	0.8	0.2	0.317
$\mathcal{X}$	✓	0.22	0.44	0.71	0.29	0.136

6 Discussion

6.1 Regarding the Evaluation Results

Further analyzing the evaluation of this approach, we identify some of the outcomes highly relevant for the realization of DT systems with coupled behavior with additional semantic services.

The metrics collected here based on the multi-case study approach indicate the potential for reuse of the architecture in two very different case study settings of a similar scale. Additionally, the computation for the metrics proposed by Washizaki et al. [71] indicate that, the Twin Manager is reusable when either semantic queries or twin schemas and their business logic behind are considered since $COR > 0$, but it is not sufficient reusable if neither twin schema nor semantic queries are considered. Additionally, in any setting, according to Washizaki et al. [71], the Twin Manager is highly customizable in terms of business methods, which limits the reusability of the approach to some extent.

Speaking of the advantages, if we assume the models, queries , and configuration files are already existing and can be straightforwardly replaced, the ratios for replacements and reuses indicate the saving in engineering effort, while the additions and deletions indicate the extra effort required. Specifically for this evaluation based on the multi-case study approach, 29.2% (34.4% if considering queries) of the components can be reused and 29.2% (31.0% if considering queries) can be replaced. This means, that at least, there is a saving in engineering effort of 58.4% (65.4% if considering queries) where at least half of it is due to the enhanced reusability of the approach, which responds to the need for reusability in **RQ1**. This reusability can be further enhanced when the approach is incorporated into the DTaaS platform, which responds to **RQ3**, where infrastructure services, such as the RabbitMQ and MQTT brokers, tools, such as, Maestro and the Twin Manager jar, and models, such as RMQ FMU, can be reused to realize DTs. On the other hand, there is at least a 33.3% (27.5% if considering queries) of components that require engineering effort. This number, in addition,

increases for systems of larger scales and the incorporation of more elaborated business logic. Therefore, there is still potential to further reduce the engineering effort, especially for the realization of large DT-enabled systems with multiple twins and twin systems.

Regarding the reusability of the semantic lifting services we point out that all four are reusable between the case studies and are directly reusable for further applications following the same architecture. The generic defect queries are also reusable, while the non-generic defect queries must be configured anew. This means that only 1 out of 6 components (4 services, 2 sets of defect queries) must be replaced, and none must be added or removed, which gives us a high reusability for the semantic lifting: Only 16.6% of components require engineering work. This shows that a positive answer of **RQ2** does not necessarily compromise the reusability aspects of **RQ1** and **RQ3**.

6.2 Capabilities

This work provides an improvement to our previous work [8] that adds the capabilities and interfaces to (i) realize composed DT systems with coupled behavior, (ii) run co-simulation settings, (iii) be used in a DTaaS platform, and (iv) provide case-independent DT services using semantic reasoning and querying. The integration of co-simulation with DT platforms also bridges two categories of frameworks to realize DTs, namely, IoT-based frameworks and co-simulation-based frameworks [22]. On the downside, this extension adds overhead for the computation of the co-simulation settings, which introduces delays that can affect the real-time response capabilities of DTs.

With the multi-case study approach used in this work, we validate the capability of this architectural approach to address the challenge regarding composition of DTs with coupled behavior, which was present in our previous work that approaches composition of DTs [7] and a common challenge in relation to composition of heterogeneous DTs [6]. This capability is the result of integrating DTs with co-simulation, which has been proved to be useful to approach the interoperability of heterogeneous hierarchies of simulators [72]. Such capability has been validated to work with coupled behavior in cascade (Three-Tank System) and synchronization (Flex-cell) coupling. Other types of coupling are yet to be researched.

The architectural approach enables a one-level composition of *twins* into *twin systems*, which provides a functional implementation of composition for DTs with coupled behavior, which responds to the qualitative aspects of **RQ1**. Twin systems, however, cannot be composed into other twin systems, and therefore, the approach is limited to a one-level composition hierarchy. Thus, the ultimate twin system to be instantiated with the architecture is to be the last in the model hierarchy (e.g., following the modeling approach in [7]). This limitation does not necessarily affect the consistency of the internal coupling if the relationships of

the composed twin are translated into hierarchical connections of the simulators in the Twin System Configuration file, which would represent the multi-level composition in the background.

Answering **RQ2**, we have shown that semantic lifting enables to check additional consistency constraints during execution. Using the generic lifting map for our architecture, it can be used to check (a) internal consistency, i.e., constraints within the components of DT platform, such as correct coupling between the robotic arms, (b) external consistency, i.e., constraints between the DT structure and external knowledge, such as the structure of the PT as described by an asset information model, and (c) generic consistency, such as the existence of at least one DT model.

The application of defect queries has been proven useful for structural self-adaptation to detect defects in programs [46] and DTs [73]. Here, we have generated the lifting map from the software architecture by manually designing an ontology for it. Alternatively, one can consider a general scheme that generates a mapping from a software architecture, based on the used architecture description language (ADL). As discussed, there is no direct mapping from UML architecture diagram to ontologies that cover all features, but alternative ADLs or a restricted subset are possible ways to give a generic mapping. In this case, one of the four semantic services would be not directly reusable, but would be automatically generated.

6.3 Limitations of This Study

In the following, we discuss the identified limitations that are threats to validity and the mitigation strategies to address them.

External validity

Simulation-only case study The Three-Tank System case study used in this work is a simulation-only system which helps with the illustration of the coupled behavior involved in such a system. However, this system is purely conceptual and does not include a proper PT, and therefore, no additional physical considerations or constraints are involved in this case study, which can limit its generalizability to more realistic settings. *Mitigation strategy:* A mitigation strategy to avoid this limitation is to use the architecture in a different case study that includes a proper PT, which also requires to have more accurate definitions for the models, configurations, and constraints required in the real setup.
One-level composition The architectural approach only supports a one-level composition, where objects of the *Twin* class can be aggregated to objects of the *Twin System* class. Objects of the *Twin System* class cannot yet be composed into a higher hierarchical level. *Migitation strategy:* A workaround for this limitation is to only implement the composed twin at the highest hierarchy level (in the conceptual model, i.e., using the modeling approach proposed in [7]) as the object of the *Twin System* class, and represent the internal hierarchical relationships as

part of the *connections* field in the *Twin System Configuration* file. This ensures the coupling and the hierarchy are consistent throughtout and implementable with the architecture.

Scalability of the semantic lifting services Our ontology is built specifically for the architecture and does neither refer to top-level-ontologies, nor to a full industrial ontology (e.g., the Industrial Data Ontology [74] or the Industrial Ontologies Foundry Core Ontology [75]). This can limit the applicability where the semantic lifting is also concerned with constraints stemming from industrial standards or data integration. Semantic technologies are computationally costly and we did not perform a detailed performance evaluation on defect queries. In our experiments, the monitoring overhead was negligible. Therefore, the case studies do not demonstrate the applicability on real-time changes of large industrial assets. We assume, however, that structural changes detected by defect queries must not be monitored in real time due to their rather rare occurrence.

Internal validity

Synchronization with the RMQ FMU The bridge between real world and co-simulation with RMQ FMU still requires improvement regarding the synchronization of input/output messages in the co-simulation and the time delays that introduced. This is still a current challenge in the architectural approach to be solved and requires deeper research in how to effectively integrate hybrid co-simulation within the Twin Manager, which is a challenging feature of co-simulation [40]. *Mitigation strategy:* A workaround for this limitation is to externally administrate the messaging between the Twin Manager and RMQ FMU (as shown in Figure 10 with the *Routine Publisher* block).

Grippers The grippers attached to the robotic arms, which are controlled independently from the robotic arms, have not been considered as explicit assets of the Flex-cell in this study, though they have modeled as components of the Flex-cell in our previous study [7]. This is because of the lack of behavioral models for the grippers, which limits their integration in the system with coupled behavior. Therefore, they have not been included in the co-simulation setting nor in the representation of the twins in the DT-enabled system. *Mitigation strategy:* The mitigation strategy for this limitation is to include the behavioral models of the grippers as FMUs. These FMUs should then be integrated via FMIEndpoints, and subsequently, aggregated into the Flex-cell DT System and interfaced via a MaestroEndpoint. This integration would require an additional change of the connections in the Twin System Configuration file of the twin system.

Measuring the engineering effort As part of the evaluation, we collect some metrics related to reusability of the components in the architectural approach. However, we do not make emphasis on the associated engineering efforts required to create the models for the case study, define the configuration settings, or set up the application with the Twin Manager. *Mitigation strategy:* A mitigation strategy for this limitation is to further measure in detail the engineering effort

required to carry out the tasks involved in the set up of a new case study with the Twin Manager approach.

7 Concluding Remarks

This paper presents an extension to a DT architecture for systems of DTs with coupled behavior via co-simulation with additional semantic services. Two representative case studies of highly coupled systems are used for the evaluation of the approach. The findings show improvements regarding the capabilities to realize DTs with coupled behavior and the combination with other tools, such as the DTaaS platform, and an increased reusability of components that can be used for different DT case studies, which leads to reducing the implementation effort of the DT engineering.

Concerning the semantic reflection, we point out that this is the first application of this approach in DTs that reflects the target structure in a generic fashion, yet abstracts from implementation detail – the lifting of the DT is not the lifting of the implementation. This opens new avenues for software engineering approaches targeting cognitive and other semantically enhanced twins.

The main contributions are related to (1) the use of extended co-simulation scenarios for hierarchical and coupled DTs, managed from a DT architecture that can be integrated into a DTaaS platform, and (2) the use of semantic lifting for microservice platforms to store, load, monitor, and check connections using semantic technologies.

Future Work

This work lays the foundation to integrate coupled DTs and semantic lifting into a DTaaS platform. We conjecture that reusing the optimizations of the DTaaS platform, as well as suitable virtualization [76] of the lifting state, will improve the performance of coupled DT simulations and increase the number of available case-independent services for deployed DTs. Furthermore, improving the synchronization and the asynchronous functionalities of the architecture and its integration with co-simulation engines is a promising research direction.

The authors also plan to assess the approach in a real industrial setting at one of the partner companies to further assert the validity for generalization and collect data regarding reusability and implementation effort. These data can then be used to more objectively evaluate the benefits of the approach using a third case study deployed in an industrial setting with feedback provided by practitioners, which would address the limitation *Measuring the engineering effort* provided in Section 6.3.

Declarations

Funding. This work has been partially funded by the Ringkøbing-Skjern Municipality, Denmark, under the Framework Collaboration Agreement for Aarhus

University Digital Transformation Lab-Skjern, the CP-SENS project supported by the Danish Innovation Foundation, the Poul Due Jensen Foundation, the Research Council of Norway through PeTWIN (Grant 294600) and SIRIUS (Grant 237898), as well as the EU project SM4RTENANCE (Grant 101123423).

Competing Interests. The authors declare that there are no competing interests.

Code Availability. The prototypical implementation and source code are publicly available on GitHub[5].

References

[1] Wagg, D., Worden, K., Barthorpe, R. & Gardner, P. Digital twins: state-of-the-art and future directions for modeling and simulation in engineering dynamics applications. *ASCE-ASME Journal of Risk and Uncertainty in Engineering Systems, Part B: Mechanical Engineering* **6**, 030901 (2020).

[2] Lehner, D. *et al.* Digital Twin Platforms: Requirements, Capabilities, and Future Prospects. *IEEE Software* **39**, 53–61 (2022).

[3] Pfeiffer, J., Lehner, D., Wortmann, A. & Wimmer, M. Modeling Capabilities of Digital Twin Platforms - Old Wine in New Bottles? *Journal of Object Technology* **21** (2022).

[4] Aheleroff, S., Xu, X., Zhong, R. Y. & Lu, Y. Digital Twin as a Service (DTaaS) in Industry 4.0: An Architecture Reference Model. *Advanced Engineering Informatics* **47**, 101225 (2021). URL https://www.sciencedirect.com/science/article/pii/S1474034620301944.

[5] Talasila, P. *et al.* *Digital Twin as a Service (DTaaS): A Platform for Digital Twin Developers and Users*, SWC 2023 (IEEE, Portsmouth, UK, 2023).

[6] Michael, J., Pfeiffer, J., Rumpe, B. & Wortmann, A. *Integration Challenges for Digital Twin Systems-of-Systems*, SESoS, 9–12 (IEEE/ACM, 2022).

[7] Gil, S., Mikkelsen, P. H., Tola, D., Schou, C. & Larsen, P. G. *A Modeling Approach for Composed Digital Twins in Cooperative Systems*, 1–8 (IEEE, 2023).

[8] Lehner, D., Gil, S., Mikkelsen, P. H., Larsen, P. G. & Wimmer, M. *An architectural extension for digital twin platforms to leverage behavioral models*, 1–8 (2023).

[9] Kamburjan, E., Klungre, V. N., Schlatte, R., Johnsen, E. B. & Giese, M. *Programming and debugging with semantically lifted states*, Vol. 12731 of *Lecture Notes in Computer Science*, 126–142 (Springer, 2021).

[10] Kritzinger, W., Karner, M., Traar, G., Henjes, J. & Sihn, W. *Digital Twin in manufacturing: A categorical literature review and classification*, Vol. 51 of *IFAC*, 1016–1022 (Elsevier, 2018).

[11] Lee, E. A. *Cyber physical systems: Design challenges*, 363–369 (2008).

[12] VanDerHorn, E. & Mahadevan, S. Digital Twin: Generalization, characterization and implementation. *Decision Support Systems* **145**, 113524 (2021). URL https://doi.org/10.1016/j.dss.2021.113524.

[13] Tao, F., Xiao, B., Qi, Q., Cheng, J. & Ji, P. Digital twin modeling. *Journal of Manufacturing Systems* **64**, 372–389 (2022).

[5] https://github.com/Edkamb/ConfLiftingaaS

[14] Zambrano, V. *et al.* Industrial digitalization in the industry 4.0 era: Classification, reuse and authoring of digital models on digital twin platforms. *Array* **14**, 100176 (2022).

[15] Jones, D., Snider, C., Nassehi, A., Yon, J. & Hicks, B. Characterising the Digital Twin: A systematic literature review. *CIRP Journal of Manufacturing Science and Technology* **29**, 36–52 (2020). URL https://doi.org/10.1016/j.cirpj.2020.02.002.

[16] Dalibor, M. *et al.* A Cross-Domain Systematic Mapping Study on Software Engineering for Digital Twins. *Journal of Systems and Software* **193**, 111361 (2022).

[17] Oakes, B. *et al.* *Improving Digital Twin Experience Reports*, 179–190 (SCITEPRESS - Science and Technology Publications, Online, 2021).

[18] Schluse, M., Priggemeyer, M., Atorf, L. & Rossmann, J. Experimentable Digital Twins-Streamlining Simulation-Based Systems Engineering for Industry 4.0. *IEEE Transactions on Industrial Informatics* **14**, 1722–1731 (2018).

[19] Barbieri, G. *et al.* A virtual commissioning based methodology to integrate digital twins into manufacturing systems. *Production Engineering* **15**, 397–412 (2021). URL https://doi.org/10.1007/s11740-021-01037-3.

[20] IEC. *Asset Administration Shell for industrial applications - Part 1: Asset Administration Shell structure* IEC 63278-1:2023 edn (International Electrotechnical Commission, Geneva, Switzerland, 2023). URL https://webstore.iec.ch/publication/65628.

[21] Larsen, P. G. *et al.* *Integrated tool chain for model-based design of Cyber-Physical Systems: The INTO-CPS project*, 2016 2nd International Workshop on Modelling, Analysis, and Control of Complex CPS, CPS Data 2016 (2016).

[22] Gil, S., Mikkelsen, P. H., Gomes, C. & Larsen, P. G. Survey on open-source digital twin frameworks–A case study approach. *Software: Practice and Experience* **54**, 929–960 (2024).

[23] Zambrano, V. *et al.* Industrial digitalization in the industry 4.0 era: Classification, reuse and authoring of digital models on digital twin platforms. *Array* 100176 (2022). URL https://www.sciencedirect.com/science/article/pii/S2590005622000352.

[24] Talasila, P. *et al.* *Comparison Between the HUBCAP and DIGITBrain Platforms for Model-Based Design and Evaluation of Digital Twins*, Vol. 13230 LNCS, 238–244 (2022).

[25] Geman, S., Potter, D. F. & Chi, Z. Composition systems. *Quarterly of Applied Mathematics* **60**, 707–736 (2002).

[26] Keller, R. K. & Schauer, R. *Design components: Towards software composition at the design level*, 302–311 (1998).

[27] Jia, W., Wang, W. & Zhang, Z. From simple digital twin to complex digital twin Part I: A novel modeling method for multi-scale and multi-scenario digital twin. *Advanced Engineering Informatics* **53**, 101706 (2022).

[28] Gao, Y., Lv, H., Hou, Y., Liu, J. & Xu, W. *Real-time modeling and simulation method of digital twin production line*, ITAIC, 1639–1642 (IEEE, 2019).

[29] Andryushkevich, S. K., Kovalyov, S. P. & Nefedov, E. *Composition and application of power system digital twins based on ontological modeling*, INDIN, 1536–1542 (IEEE, 2019).

[30] Preuveneers, D., Joosen, W. & Ilie-Zudor, E. *Robust Digital Twin Compositions for Industry 4.0 Smart Manufacturing Systems*, International Enterprise Distributed Object Computing Workshop, 69–78 (IEEE, 2018).

[31] Human, C., Basson, A. H. & Kruger, K. A design framework for a system of digital twins and services. *Computers in Industry* **144**, 103796 (2023).
[32] Schroeder, G. N. *et al.* *A Methodology for Digital Twin Modeling and Deployment for Industry 4.0*, Vol. 109, 556–567 (2021).
[33] Dinar, M. & Rosen, D. W. A design for additive manufacturing ontology. *Journal of Computing and Information Science in Engineering* **17**, 1–9 (2017).
[34] Ayinla, K., Vakaj, E., Cheung, F. & Tawil, A. R. H. *A semantic offsite construction digital Twin-Offsite Manufacturing Production Workflow (OPW) ontology*, Vol. 2887, 1–14 (2021).
[35] Maria, A. *Introduction to modeling and simulation*, 7–13 (1997).
[36] Banks, J. *Handbook of Simulation* (Wiley, 1998).
[37] Gomes, C., Thule, C., Broman, D., Larsen, P. G. & Vangheluwe, H. Co-simulation: A survey. *ACM Computing Surveys* **51** (2018).
[38] Sadjina, S. *et al.* Distributed co-simulation of maritime systems and operations. *Journal of Offshore Mechanics and Arctic Engineering* **141** (2019).
[39] Bullock, D., Johnson, B., Wells, R. B., Kyte, M. & Li, Z. Hardware-in-the-loop simulation. *Transportation Research Part C: Emerging Technologies* **12**, 73–89 (2004).
[40] Cremona, F. *et al.* Hybrid co-simulation: it's about time. *Software and Systems Modeling* **18**, 1655–1679 (2019).
[41] Thule, C., Lausdahl, K., Gomes, C., Meisl, G. & Larsen, P. G. Maestro: The INTO-CPS co-simulation framework. *Simulation Modelling Practice and Theory* **92**, 45–61 (2019).
[42] Havard, V., Jeanne, B., Lacomblez, M. & Baudry, D. Digital twin and virtual reality: a co-simulation environment for design and assessment of industrial workstations. *Production and Manufacturing Research* **7**, 472–489 (2019). URL https://doi.org/10.1080/21693277.2019.1660283.
[43] Fitzgerald, J., Larsen, P. G. & Pierce, K. *Multi-modelling and Co-simulation in the Engineering of Cyber-Physical Systems: Towards the Digital Twin*, Vol. 11865 LNCS of *Lecture Notes in Computer Science*, 40–55 (Springer, 2019). URL http://dx.doi.org/10.1007/978-3-030-30985-5{_}4.
[44] Qu, Y., Kamburjan, E., Torabi, A. & Giese, M. Semantically triggered qualitative simulation of a geological process. *Applied Computing and Geosciences* **21**, 100152 (2024).
[45] Kamburjan, E. & Johnsen, E. B. *Knowledge structures over simulation units*, ANNSIM, 78–89 (IEEE, 2022).
[46] Kamburjan, E. *et al.* *Digital twin reconfiguration using asset models*, Vol. 13704 of *Lecture Notes in Computer Science*, 71–88 (Springer, 2022).
[47] Paredis, R. & Vangheluwe, H. *Towards a digital Z framework based on a family of architectures and a virtual knowledge graph*, 491–496 (ACM, 2022).
[48] Lu, J., Zheng, X., Gharaei, A., Kalaboukas, K. & Kiritsis, D. Cognitive twins for supporting decision-makings of internet of things systems. *CoRR* **abs/1912.08547** (2019).
[49] Li, H., Wang, G., Lu, J. & Kiritsis, D. Cognitive twin construction for system of systems operation based on semantic integration and high-level architecture. *Integr. Comput. Aided Eng.* **29**, 277–295 (2022).

[50] Rozanec, J. M. *et al.* Actionable cognitive twins for decision making in manufacturing. *Int. J. Prod. Res.* **60**, 452–478 (2022).

[51] Li, Y. *et al.* Co-simulation of complex engineered systems enabled by a cognitive twin architecture. *International Journal of Production Research* **60**, 7588–7609 (2022).

[52] Abburu, S. *et al. COGNITWIN - hybrid and cognitive digital twins for the process industry*, 1–8 (IEEE, 2020).

[53] Ali, M. I., Patel, P., Breslin, J. G., Harik, R. F. & Sheth, A. P. Cognitive digital twins for smart manufacturing. *IEEE Intell. Syst.* **36**, 96–100 (2021).

[54] Zheng, X., Lu, J. & Kiritsis, D. The emergence of cognitive digital twin: vision, challenges and opportunities. *Int. J. Prod. Res.* **60**, 7610–7632 (2022).

[55] Tsang, E. W. Generalizing from research findings: The merits of case studies. *International Journal of Management Reviews* **16**, 369–383 (2014).

[56] Wieringa, R. & Daneva, M. Six strategies for generalizing software engineering theories. *Science of Computer Programming* **101**, 136–152 (2015). URL http://dx.doi.org/10.1016/j.scico.2014.11.013.

[57] Gil, S., Oakes, B. J., Gomes, C., Frasheri, M. & Larsen, P. G. Toward a systematic reporting framework for digital twins: a cooperative robotics case study. *SIMULATION* 1–27 (2024). URL https://doi.org/10.1177/00375497241261406.

[58] Azizkhani, M., Godage, I. S. & Chen, Y. Dynamic Control of Soft Robotic Arm: A Simulation Study. *IEEE Robotics and Automation Letters* **7**, 3584–3591 (2022).

[59] Madsen, E., Tola, D., Hansen, C., Gomes, C. & Larsen, P. G. *AURT: A Tool for Dynamics Calibration of Robot Manipulators**, 190–195 (IEEE, 2022).

[60] Corke, P. & Haviland, J. *Not your grandmother's toolbox–the robotics toolbox reinvented for python*, 11357–11363 (IEEE, 2021).

[61] Legaard, C. M., Tola, D., Schranz, T., Macedo, H. D. & Larsen, P. G. *A universal mechanism for implementing functional mock-up units*, SIMULTECH 2021, to appear (Virtual Event, 2021).

[62] Fitzgerald, J., Gomes, C. & Larsen, P. G. (eds) *The Engineering of Digital Twins* (Springer, 2024).

[63] Gamma, E., Johnson, R., Helm, R., Johnson, R. E. & Vlissides, J. *Design patterns: elements of reusable object-oriented software* (Addison-Wesley, 1995).

[64] Frasheri, M., Ejersbo, H., Thule, C. & Esterle, L. Macedo, H. D., Thule, C. & Pierce, K. (eds) *Rmqfmu: Bridging the real world with co-simulation for practitioners.* (eds Macedo, H. D., Thule, C. & Pierce, K.) *Proceedings of the 19th International Overture Workshop* (Overture, 2021).

[65] Mkhinini, M. M., Labbani-Narsis, O. & Nicolle, C. Combining UML and ontology: An exploratory survey. *Comput. Sci. Rev.* **35**, 100223 (2020).

[66] Horridge, M. *et al. The manchester OWL syntax*, Vol. 216 of *CEUR Workshop Proceedings* (CEUR-WS.org, 2006).

[67] Wiens, V., Lohmann, S. & Auer, S. *Webvowl editor: Device-independent visual ontology modeling*, Vol. 2180 of *CEUR Workshop Proceedings* (CEUR-WS.org, 2018).

[68] IEEE Standard for Information Technology–System and Software Life Cycle Processes–Reuse Processes. *IEEE Std 1517-2010 (Revision of IEEE Std 1517-1999)* 1–51 (2010).

[69] Ghasemi, G., Müller, M. S., Jazdi, N. & Weyrich, M. *Quality Analysis Framework based on Complexity for Change Management Using Intelligent Digital Twin*, Vol. 120, 1516–1521 (Elsevier B.V., 2023). URL https://doi.org/10.1016/j.procir.2023.09.207.

[70] Lia, M. & Colella, D. D. *CkanFAIR: a digital tool for assessing the FAIR principles*, 3980–3984 (IEEE, 2023).
[71] Washizaki, H., Yamamoto, H. & Fukazawa, Y. *A metrics suite for measuring reusability of software components*, 211–223 (IEEE, 2003).
[72] Gomes, C. *et al.* Semantic adaptation for FMI co-simulation with hierarchical simulators. *Simulation* **95**, 241–269 (2019).
[73] Kamburjan, E. *et al.* *GreenhouseDT: An Exemplar for Digital Twins*, SEAMS'24 (ACM, 2024).
[74] ISO. Automation systems and integration — ontology based interoperability. Standard, International Organization for Standardization, Geneva, CH (2024).
[75] Drobnjakovic, M. *et al.* *The industrial ontologies foundry (IOF) core ontology*, Vol. 3240 of *CEUR Workshop Proceedings* (CEUR-WS.org, 2022).
[76] Xiao, G., Ding, L., Cogrel, B. & Calvanese, D. Virtual knowledge graphs: An overview of systems and use cases. *Data Intell.* **1**, 201–223 (2019).