Methods

Daniel Lehner*, Sabine Sint, Martin Eisenberg and Manuel Wimmer

A pattern catalog for augmenting Digital Twin models with behavior

Ein Musterkatalog zur Erweiterung von digitalen Zwillingsmodellen um Verhaltenssichten

https://doi.org/10.1515/auto-2022-0144
Received November 5, 2022; accepted March 20, 2023

Abstract: Digital Twins are emerging as a solution to build and extend existing software systems to make better use of data produced by physical systems. For supporting the development of Digital Twins, several software vendors are offering dedicated tool support, often referred to as Digital Twin platforms. The modeling capabilities of these platforms are mostly concerned with structural viewpoints, i.e., providing an overview of available components including their current and historical sensor values. However, behavioral viewpoints did not yet receive much attention on these platforms. As behavioral models are often used during the design processes, e.g., for simulation and synthesis, it would be beneficial for having them included in Digital Twin platforms, e.g., for reasoning on the set of possible next actions or for checking the execution history to perform runtime validation. In this paper, we present a catalog of modeling patterns for augmenting Digital Twin models with behavioral models and their corresponding runtime information without requiring any extension of the code bases of Digital Twin platforms. We demonstrate the presented modeling patterns by applying them to the Digital Twin platform offered by Microsoft, in an additive manufacturing use case of a 3D printer in a production line.

Keywords: behavior modeling; Digital Twins; language engineering; model-driven engineering; modeling patterns.

Zusammenfassung: Digitale Zwillinge entwickeln sich zu einer Lösung für den Aufbau und die Erweiterung bestehender Softwaresysteme, um die von physischen Systemen erzeugten Daten besser nutzen zu können. Um ihre Entwicklung zu unterstützen, bieten mehrere Softwareanbieter spezielle Tools an, die oft als Digital Twin-Plattformen bezeichnet werden. Die Modellierungsmöglichkeiten dieser Plattformen beziehen sich hauptsächlich auf strukturelle Gesichtspunkte, d.h., sie bieten einen Überblick über die verfügbaren Komponenten einschließlich ihrer aktuellen und historischen Sensorwerte. Da Verhaltensmodelle häufig während des Entwurfsprozesses verwendet werden, z. B. für Simulationen, wäre es von Vorteil, wenn sie in Digital Twin-Plattformen enthalten wären, z. B. um die möglichen nächsten Aktionen zu bestimmen oder die Ausführungshistorie zu überprüfen, um eine Laufzeitvalidierung durchzuführen. In diesem Beitrag stellen wir einen Katalog von Modellierungsmustern vor, mit denen die Modelle des Digitalen Zwillings um Verhaltensmodelle und die entsprechenden Laufzeitinformationen erweitert werden können, ohne dass eine Erweiterung der Codebasen erforderlich ist. Wir demonstrieren die Muster anhand eines 3D Drucker Anwendungsfalls, indem wir sie auf die von Microsoft angebotene DT-Plattform anwenden.

Schlagwörter: Verhaltensmodellierung; Digitale Zwillinge; Language Engineering; Model-Driven Engineering; Modellierungsmuster.

1 Introduction

Digital Twins (DTs) are becoming important ingredients in realizing software-defined manufacturing [1]. Based on a commonly used definition that was initially proposed by Kritzinger et al. [2], the DT provides a virtual representation of the physical system that enables bi-directional synchronization between it and its physical counterpart. DTs promise to support a multitude of tasks of the whole system life-cycle [3, 4]. In this context, more and more software

***Corresponding author: Daniel Lehner**, JKU Linz Christian Doppler Laboratory for Model-Integrated Smart Production (CDL-MINT), Department of Business Informatics – Software Engineering, Johannes Kepler University Linz, Altenberger Straße 69, 4040 Linz, Austria,
E-mail: daniel.lehner@jku.at
Sabine Sint, **Martin Eisenberg and Manuel Wimmer**, JKU Linz Christian Doppler Laboratory for Model-Integrated Smart Production (CDL-MINT), Department of Business Informatics – Software Engineering, Johannes Kepler University Linz, Altenberger Straße 69, 4040 Linz, Austria,
E-mail: sabine.sint@jku.at (S. Sint), martin.eisenberg@jku.at (M. Eisenberg), manuel.wimmer@jku.at (M. Wimmer)

development companies provide a set of tools, also referred to as DT platforms, to automate common DT creation and maintenance tasks.

Current DT platforms offer dedicated support for structural modeling aspects, such as defining components and topologies of systems to represent current and historical runtime states of systems. However, behavioral aspects are most often a second concern [5], although these behavioral aspects are usually required for modeling systems, especially when modeling DTs [6]. Such behavioral models are used during the design and generation processes, e.g., for simulation and synthesis, and thus also available in current system modeling languages, such as SysML[1] or AutomationML.[2] Enabling this type of modeling in current DT platforms as well would allow reflection on the runtime state of systems from behavioral viewpoints. As a result, a more systematic analysis of generated runtime data is achieved, e.g., comparing simulation traces with actual execution traces, to mention just one possible use case.

Therefore, the research goal of this paper is to find means to augment existing DT models with behavior without requiring heavyweight extensions of such platforms. With the term heavyweight extensions, we mean the necessity of changing the code base of platforms for introducing additional modeling capabilities. Instead, we are interested in representing behavioral models directly using the current modeling capabilities of existing platforms. By finding proper structures, structural DT models are augmented with behavior. Such augmentation also includes a representation of the historical traces of behavior execution, i.e., the runtime of the system. In this paper, we focus on discrete behavior models and leave continuous ones as subject to future work.

To achieve this research goal, we present a modeling pattern catalog for explicating how behavioral models, including their runtime traces, can be represented in DT platforms without having to rely on heavyweight extensions of the code bases of these platforms. This enables us to leverage provided features of these platforms, e.g., scalability while reusing information from behavioral models which may be already in use in the early engineering process.

We demonstrate the modeling pattern catalog by its application to a 3D printer use case. In particular, we demonstrate how to realize the presented patterns on top of the Microsoft DT platform.[3] The results of this study indicate that the presented patterns can be realized to represent behavioral models with their runtime traces in current DT platforms which already provide certain structural modeling capabilities.

By proposing the pattern catalog and demonstrating the realization of the patterns in the Microsoft DT platform, this work contributes to theoretical results about patterns for software language engineering, practical tooling results for the Microsoft DT platform, and takeaways for researchers, tool builders, and platform operators.

The remainder of this paper is structured as follows. In Section 2, we explain the background of this work, i.e., Model-Driven Engineering and DTs. Section 3 introduces a running example of a 3D printer used throughout the paper. Section 4 introduces the different modeling patterns that allow extending current DT platforms having a fixed structural modeling language with behavioral viewpoints. In Section 5, we demonstrate the usage of the presented patterns for the Microsoft DT platform through our running example. Section 6 provides a critical discussion of the presented work. Finally, in Section 7, we present and discuss related work, before we conclude the paper with an outlook on future work in Section 8.

2 Background

This section summarizes relevant background information for this work. We briefly outline Model-Driven Engineering (MDE), Digital Twins, and the combination of both.

2.1 Model-Driven Engineering

In Model-Driven Engineering (MDE), models are considered as the central artifacts in the engineering process [7]. Engineering problems are attempted to be solved by using formal models, i.e., machine-readable and processable representations, which allow for different viewpoints on the systems. By this, the abstraction power of models is used to cope with the increasing complexity of current systems [7, 8] which is, of course, also of interest for software-defined manufacturing. To be able to represent the observed reality in formal models, dedicated modeling languages, such as the Unified Modeling Language (UML[4]) or Domain-Specific Modeling Languages (DSMLs) are used. In this context, a four-layer metamodeling stack (M0 – M3) [7, 9] is often used for defining how metalanguages, languages, models, and runtime traces are related to each other. Besides, there exist approaches for multi-level modeling [10], where one goal is to reduce complexity in domain models by having

1 https://sysml.org.

2 https://www.automationml.org.

3 https://azure.microsoft.com/products/digital-twins.

4 http://uml.org.

several instantiation levels and where a level can influence elements in more than just the level immediately below it [11–13]. For the work presented in this paper, we are focusing on the metamodeling stack which is illustrated in Figure 1.

At the M3 layer, meta-metamodels define meta-languages (Meta-Language (ML)), which specify the constructs for languages and their relations at the next layer, the so-called M2 layer. At this layer, metamodels define languages (Language (L)), which describe the formalisms for models/domain models (Domain Model (DM)) one layer lower at M1. Domain models specify the general concepts within the given domain to finally describe the system and Runtime Trace (RT) at the M0 layer. In Figure 1, we define a Structural Language (SL) and a Behavioral Language (BL) based on the Meta Object Facility (MOF) for defining the concepts for Structural Models (SM) and Behavioral Models (BM). These SM and BM are instances of the belonging SL and BL and are associated with each other. At the M0 layer, the Structural Trace (ST) and the Behavioral Trace (BT) represent the elements and their interactions during system runtime.

2.2 Digital Twins and Digital Twin platforms

DTs are software systems comprising data, models, and services to interact with physical systems for a specific purpose [2, 14, 15]. To enable the efficient and systematic development of DTs, various vendors provide dedicated tool support, the so-called Digital Twin platforms [16]. Examples include Microsoft Azure,[5] and Amazon Web Services,[6] or tool platforms, such as Eclipse.[7] These platforms provide predefined services for establishing a bi-directional connection with physical assets to collect data via sensors and control actuators. They also provide the ability to connect these assets to tools, such as time series databases or visualization dashboards. The services help to reduce the number of repetitive tasks, while also supporting certain quality properties such as scalability and interoperability [16]. One common feature of these different platforms is that they provide dedicated modeling support for the creation of DTs. An investigation of their modeling capabilities [5] shows that they support a variety of commonalities.

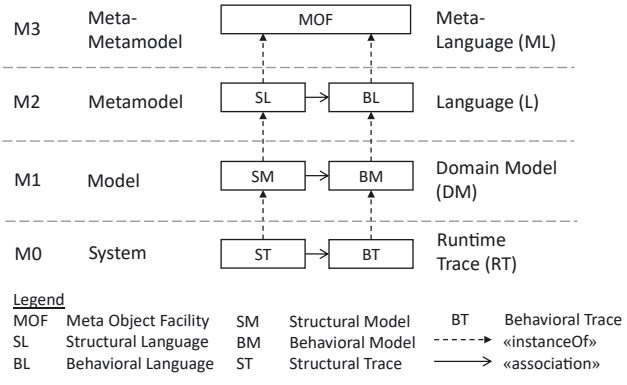

Figure 1: Four-layer metamodeling stack.

2.3 MDE for Digital Twins

By applying MDE to DTs, the strengths of both fields can be combined. Figure 2 shows an architecture for this combination. At the Modeling Layer the structural and behavioral models (SM and BM) are modeled. They conform to their structural and behavioral language (SL and BL). At the Realization Layer the actual System and its digital counterpart (Digital Twin) are located. Based on the machine-readable models and available transformations, the system or/and the DT can be automatically generated. In previous work, we have shown this transformation for structural information incorporated in SM, such as AutomationML models [17] or UML class diagrams [5]. This structural information comprises information on which components are running in a system, which properties can be sensed, and particular values of these properties at a given point in time.

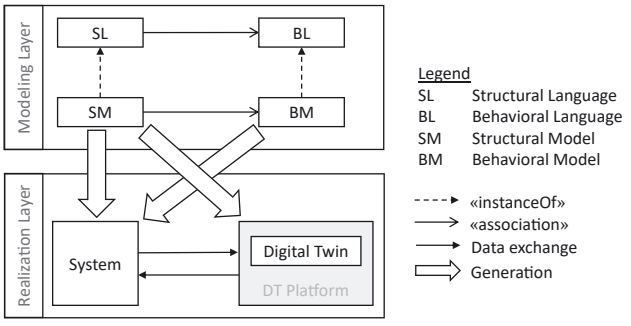

Figure 2: MDE for DTs: From design models to Digital Twin artefacts.

5 IoTHub: https://azure.microsoft.com/services/iot-hub/, DT service: https://azure.microsoft.com/services/digital-twins/, and TSI database: https://azure.microsoft.com/services.

6 AWS IoT Greengrass https://docs.aws.amazon.com/greengrass/v2/developerguide/what-is-iot-greengrass.html.

7 Eclipse Hono: https://www.eclipse.org/hono/, Vorto: https://www.eclipse.org/vorto/, and Ditto: https://www.eclipse.org/ditto/.

3 Running example: 3D printer

As a running example for this paper, we consider a production line where additive manufacturing is used to improve flexibility, support complex geometries, and simplify fabrication. As part of this production line, 3D printers are used for printing different items needed for the final product. In this context, we will focus on the 3D printer and identify open challenges at the end of this section.

3.1 3D printer models

Figure 3a shows a simplified structure for the used 3D printers in terms of a UML class diagram. A 3DPrinter has a unique id, a temperature value temp showing the current processing temperature of the printer, and the three different axes *x*, *y*, *z* to indicate the current position of the print head. Besides these attributes, the 3D printer can process signals as input (receive), such as connect, sendCADFile, print, pause, stop, and reset and as output (send), such as finished for indicating the end of a print job and error to indicate errors.

In addition to the structural information summarized in the class diagram, the 3D printer can take different states during its operation. Figure 3b shows the behavioral model of the 3D printer as a UML state machine. As starting point, it will be in the state StandBy waiting for any connection. After connecting a device to the printer, the state will change to Connected and the printer is ready to receive a print file. After receiving the file to print, the 3D printer is in the state Operational, does some preheating, and then is ready to print. When it starts printing, the operational state will be Printing. During printing, there is the opportunity to pause the printing (PausePrinting) and restart it again or cancel the print job (CancelPrinting). When the print job is finished, the 3D printer will go back to the StandBy state. If an error occurs at any time, the printer changes to the state Error. After resetting it, it will change back to StandBy.

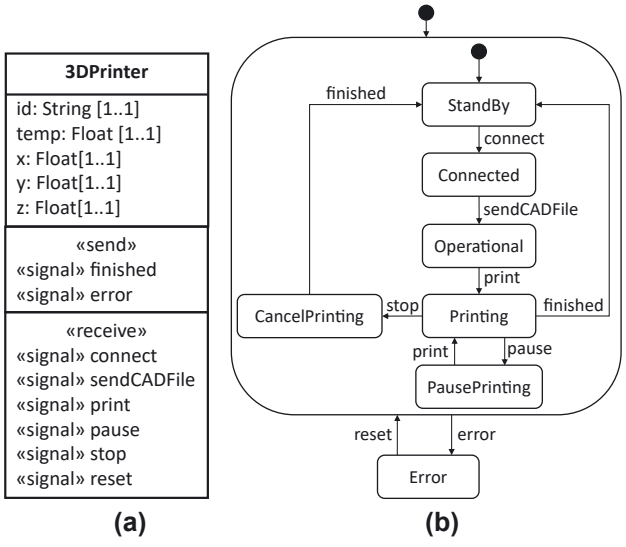

Figure 3: Structural and behavioral views of a 3D printer. Please note that we use a custom shortcut notation concerning sending and receiving signals. (a) Structural Model as Class Diagram, (b) Behavioral Model as State Diagram.

Based on the defined structure and behavior in the models, properties, and states of the 3D printer can change during runtime. Figure 4 shows an excerpt of a possible runtime execution trace. From snapshot t_1 to t_2 temperature (from 35 to 80), state (from operational to printing), and *y*-position (from 0 to 1) change. From snapshot t_2 to t_3 the change is no longer linear but depends on the signal being processed. Depending on the signal, the printer changes to another state (StandBy or PausePrinting) with other specified values for the properties. Snapshot t_3 does not show a view produced by execution but shows opportunities that could be analyzed by using state machines to reason about the potential next states, e.g., in what-if analysis.

This runtime execution trace can be divided into one structural and one behavioral trace. In the structural execution trace, the value change of the properties of the class are propagated, i.e., how the position and the temperature change (Figure 4, change of temp and *y* values). In the behavioral execution trace, behavior is observed over time. For instance, the status of the 3D printer changes from operational to printing and then to standby or pause printing (Figure 4). Based on the behavior and associated state, the system can respond differently to interactions.

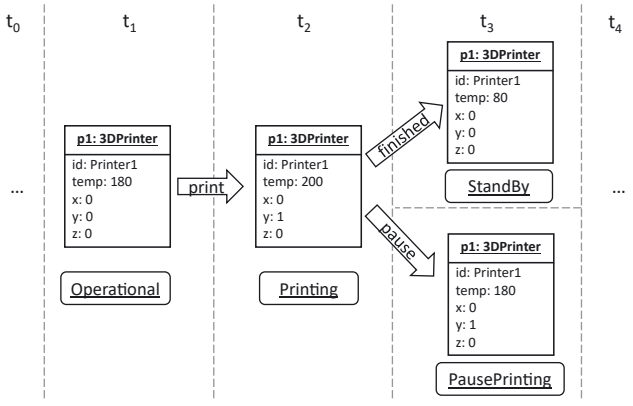

Figure 4: Excerpt of a runtime execution trace of the 3D printer.

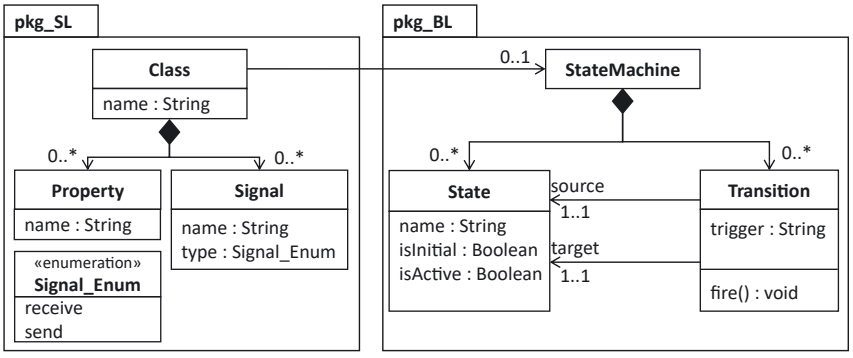

Figure 5: Example metamodels of a simplified structural and behavioral language.

The presented models are expressed using a structural and behavioral language defined at the metamodeling layer. For this purpose, Figure 5 shows a simplified example based on UML class diagrams and state machines. A `Class` can have properties (`Property`) and `Signals`, which could be of type `receive` or `send`. A class can reference a `StateMachine`. This state machine consists of `States` and `Transitions`. Each transition has a `trigger`, a defined `source` and `target` state, and an operation to indicate if the transition fires (`fire()`). States have a `name` and different booleans which indicate if the state is the initial or final state or if it is active or not. In the representation of the running example, we used a composite state (Figure 3b) to simplify the syntactical view on the state machine where every state can switch to an error state. However, this composite state can be flattened/reduced (i.e., from each state there is a transition to the error state), therefore we do not consider it further in our approach. Please note that in Figure 5, both the design and runtime views for the models are shown as is required for executing the models, e.g., using simulations [18]. Thus, it is essential that also methods, such as fire(), and boolean attributes, such as is Active, are modeled.

3.2 3D printer Digital Twin

Based on the defined language and the structural model, the system can also be realized for a DT platform. Listing 1 shows an excerpt of the concrete syntax representation of the 3D printer interface for the Azure DT platform. In this listing, the different properties and signals of the printer (cf. Figure 3a) are expressed in the Azure Digital Twin Definition Language (DTDL) which is based on the JSON-LD[8] syntax.

Listing 1 Excerpt of the 3D Printer interface in Azure DTDL.

```
{"@type": "Interface",
"displayName": "3DPrinter",
"@id": "dtmi:com:cdl:3DPrinter;2",
"contents":[{
        "@type": ["Telemetry", "Temperature"],
        "schema": "float",
        "unit": "degreeCelsius",
        "name": "temperature" }, {
        "@type": ["Telemetry"],
        "schema": "float",
        "name": "x" },
        ..., {
        "@type": ["Command"],
        "name": "connect" },
        ....,{
        "@type": ["Command"],
        "name": "reset" }],
}
```

The listing shows the structural representation of the system, but the mapping of behavioral models is still an open issue, which is also highlighted in the following two challenges.

- **Challenge 1:** Currently, DT platform languages neither provide native support for behavioral descriptions nor do they have the ability for extending the offered modeling support. How can we still use behavioral viewpoints for DTs without heavyweight extensions, i.e., changes to the code base, of the platforms?
- **Challenge 2:** There is no representation of histories of such behavioral descriptions since there is not yet a behavioral viewpoint in DT platforms. If we would have behavioral viewpoints in DT platforms, how can they be utilized for historical information similar to what was done for structural models?

In the next section, we tackle these two challenges by introducing dedicated patterns to add behavioral viewpoints to

8 https://www.w3.org/TR/json-ld.

DT models which also allow for representing the history of a system using these new viewpoints.

4 A pattern catalog for representing behavioral models in Digital Twin platforms

In this section, we present a catalog of patterns, which can be employed for overcoming the challenges mentioned in the previous section. In particular, we present patterns for extending the capabilities of Digital Twin Modeling Languages (DTMLs) as offered by existing DT platforms by reducing three meta-modeling levels into two levels. Furthermore, we show how to represent histories of behavior by exploiting different temporal modeling patterns. The concrete realization of the patterns is shown in Section 5 for Microsoft Azure.

4.1 Augmenting Digital Twin models with behavior

As DTMLs are currently neither defined by dedicated meta-languages nor provide multi-level modeling techniques that allow explicit language extensions [11–13], they, unfortunately, cannot be extended at the language level. This would indeed break compatibility with the supporting platform (cf. Challenge 1 as discussed before). Therefore, in order to still be in the position to extend these languages with behavioral viewpoints, we need to perform these extensions at a lower level of the meta-modeling stack, i.e., on levels M0 and M1, since M2 is fixed. For this purpose, we propose two patterns that serve as a blueprint for augmenting DT models with behavior descriptions as visualized in Figure 6. It is important to consider that due to the reduction of metamodeling layers, we do not keep the strict separation between levels as shown in Figure 1 in the background, but have to use them in a more flexible way.

The application of these patterns enables the modeling of behavioral aspects in combination with structural aspects, leveraging the existing infrastructure (i.e., modeling tools, APIs, code generators, and runtime environments) already offered by DT platforms. The main idea of the patterns is that the behavioral aspects, including the language to express these aspects, are defined as instantiations of the already available modeling language for the structural aspects – in our setting, DTML. By this, we aim to compensate for the one missing level of instantiation. The two proposed patterns offer two alternatives on how to augment behavior to DT models: one is substituting the instanceOf relationships with associations (the behavioral model (BM) is referring to language elements (BL) residing on the same level) while the other is merging the artifacts of two levels (BM and its traces (BT)) into one artifact on one level (BM + BT) which is a direct instantiation of BL.

As a prerequisite to augment current DT models with behavior for both patterns, we introduce SL+ (Figure 6), a base that is used by structural models in our blueprint as a link to behavioral aspects. In particular, all entities of the structural model (SM) which should be equipped with behavior descriptions should inherit from a particular base class.

The description of the individual patterns, the trade-offs between these two alternatives, their implementation, and their application are described in the following based on the well-known pattern description template [19]. For the implementation, we use the well-established UML notation before moving on to more specific implementations for a DT platform in Section 5.

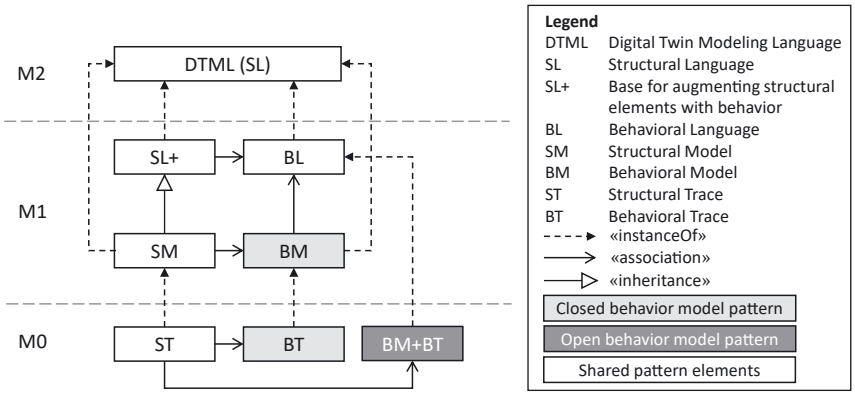

Figure 6: Macro-view on open versus closed behavior model patterns.

4.1.1 Closed behavior model pattern

Purpose: The Closed Behavior Model Pattern has the purpose of describing the behavior of a closed system, meaning that details on all model elements are already known at design time, and defined as a schema for particular object types. At runtime, this pattern ensures that the behavioral model elements are instantiated only in a particular way based on this defined behavioral schema.

Structure: In this pattern (cf. Figure 6), the behavioral model (BM) is created as an instantiation of the DTML, and in addition, an association relation is used to refer to the elements of the behavioral language (BL). This relation substitutes the instanceOf relation as shown in Figure 1. The structural model (SM) has an association with the behavioral model detailing the behavior of the corresponding structural entities. In the behavioral model, behavioral language is used to specify constraints given by the particular domain. These constraints are used to guide the development of the behavioral trace (BT) which is a direct instance of the behavioral model.

Consequences: In this pattern, the behavioral language serves as semantic meta-information of the behavioral traces, as these traces are instantiated directly from the behavioral model. Explicitly modeling domain-specific constraints in the behavioral model requires some modeling effort, but enables the validation of these constraints on the behavioral trace. However, this strict instantiation comes with reduced flexibility on M0, e.g., to add new types of behavioral traces. This is only possible by adapting the model on M1.

Implementation: Figure 7 shows the implementation of this pattern. A given `DomainBehavior` is linked to the respective `Behavior` element via an association. This association is annotated as unsettable, meaning that instantiations of this DomainBehavior on M0 (e.g., DomainBehavior4StructuralTrace1 in Figure 7) cannot use this association to link to a particular element on M0. Thus, this association serves as semantic meta-information to refer to the corresponding `Behavior` of the `DomainBehavior` used to instantiate a particular behavioral trace. `Structure` and `Behavior` are both abstract elements to ensure that traces are instantiated only from the domain-specific parts, i.e. `DomainStructure` and `DomainBehavior`, with the defined constraints.

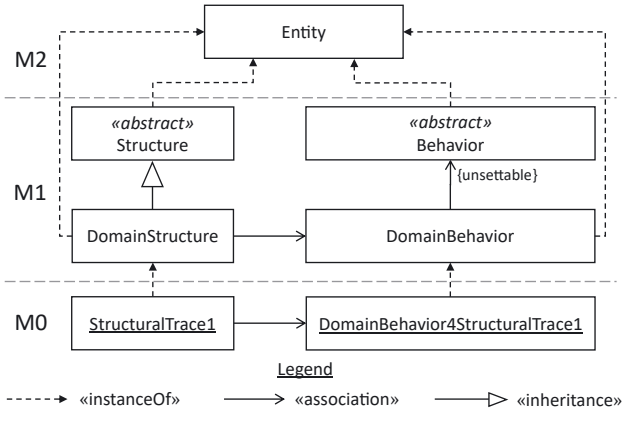

Figure 7: Implementation of the closed behavior model pattern.

Instantiation on the running example: Instantiating the implementation of this pattern to the running example of this paper, leads to the elements as illustrated in Figure 8. Based on the structural and behavioral languages as described in Figure 5 (note that the fire() operation of the transition in Figure 5 is represented as fire attribute in Figure 7), the 3DPrinter contains an inheritance relationship to `Class` and an outgoing association to the respective state machine describing the 3D Printer behavior (cf. `PrinterStateMachine` in Figure 8). Figure 8 visualizes the contents of this `PrinterStateMachine` using the example of the states `Operational` and `Printing`, which are connected via the transition `Print`. `PrinterStateMachine`, `Operational`, `Printing`, and `Print comprise` unsettable associations to the `StateMachine`, `State`, or `Transition` that contain the respective meta-information on the language level (i.e., attributes and associations). The domain-specific definitions of these language elements define (*i*) further restrictions on attributes (e.g., the `isStart` is initialized with a value, which cannot be changed on M0, by marking the corresponding attributes as final), (*ii*) constraints on associations (e.g., specify that the `Operational` can only be linked to `Printing` via `Print`), and (*iii*) guidelines (e.g., the name attribute of a `State` or `Transition` is not defined within the elements of `DomainBehavior` as values of attributes, but rather represented by the respective class names such as `Operational` or `Print`). By this explicit modeling of domain-specific constraints on M1, the structural and behavioral dependencies on M0 can be controlled. It results, for example, that the attribute `isStart` of the states has not to be considered in the behavioral traces anymore.

4.1.2 Open behavior model pattern

Purpose: The purpose of the Open Behavior Model Pattern is the behavioral description of systems in which the behavior of the system might be changed during runtime, i.e., systems with open behavior. The behavior can be specified individually on the object level, but not on the type level.

Structure: In this pattern (Figure 6, white and dark gray marked part of the macro view), the M1 level

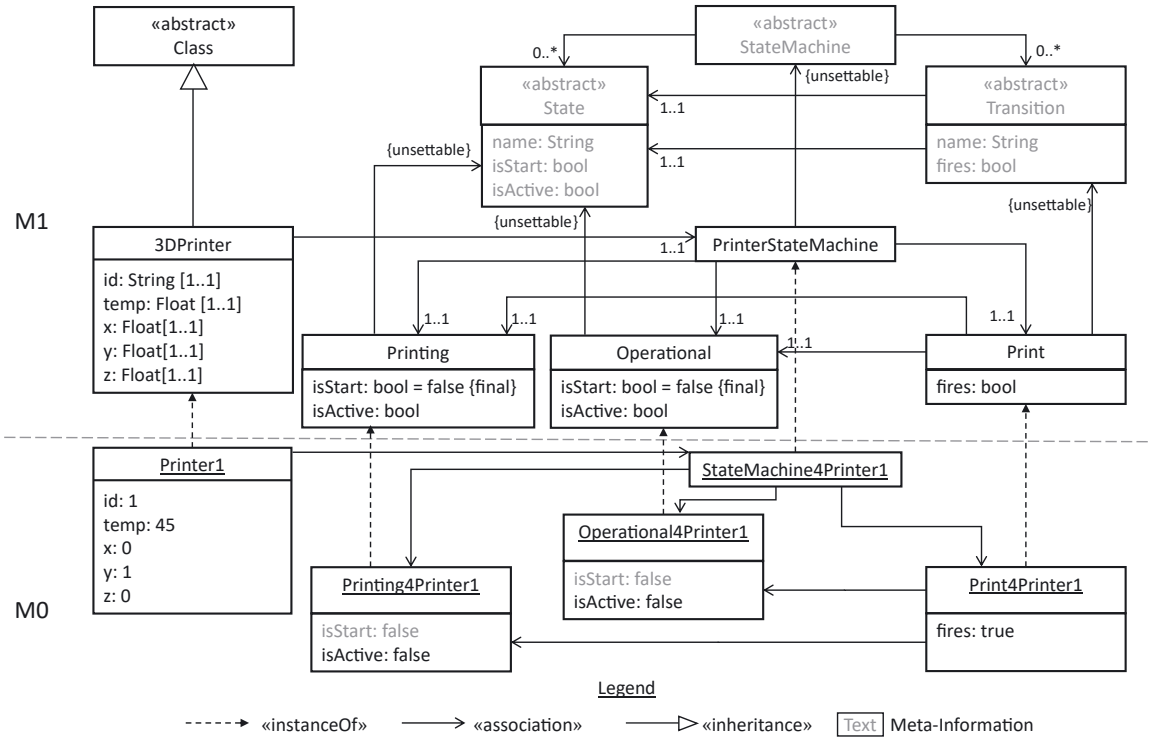

Figure 8: Closed behavior model pattern applied to the running example.

contains the information about the behavioral language (BL), whereas the domain-specific details of the behavioral model (BM) are merged with the behavioral traces (BT) on the M0 level. Thus, the behavioral model and the behavioral trace are both directly instantiated from the behavioral language.

Consequences: The lack of a dedicated DomainBehavior description (cf. Section 4.1.1) in this pattern reduces the initial modeling effort for describing a system on M1. However, it increases the modeling effort on M0 as the domain behavior needs to be introduced on this level. Moreover, the lack of domain-specific constraints on M1 does not allow for the validation of correct instantiation for particular structural entity types and thus remains specific for the given instances. Nevertheless, these missing constraints increase the flexibility of the modeled behavior on M0, as it can be changed without the need for any adaptations on M1.

Implementation: The implementation of this pattern is visualized in Figure 9. Here, the behavioral traces are directly instantiated from the `Behavior`, requiring the behavior to not be abstract. This means that the correct modeling of domain-specific aspects (e.g., which transitions can be connected to which states) needs to be considered implicitly when modeling the behavioral

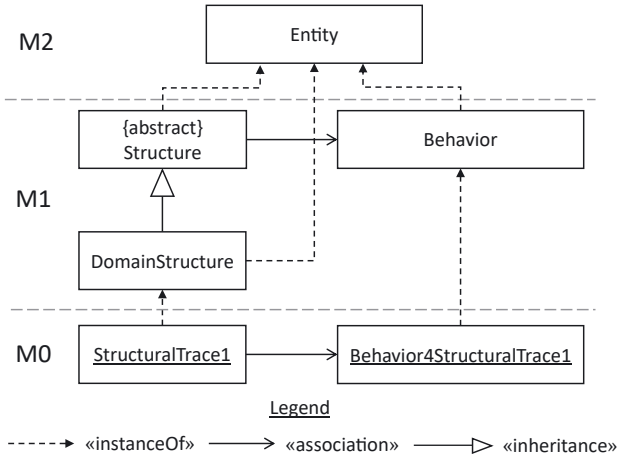

Figure 9: Implementation of the open behavior model pattern.

traces (`Behavior4StructuralTrace1`) on M0. The structural trace (`StructuralTrace1`) is connected by an association relation to this implicitly modeled domain behavior in the behavior trace.

Instantiation on the running example: Applying this pattern to the running example on M0, as visualized in Figure 10, the particular `Printer1` on M0 is connected to its `StateMachine4Printer1`, which is instantiated

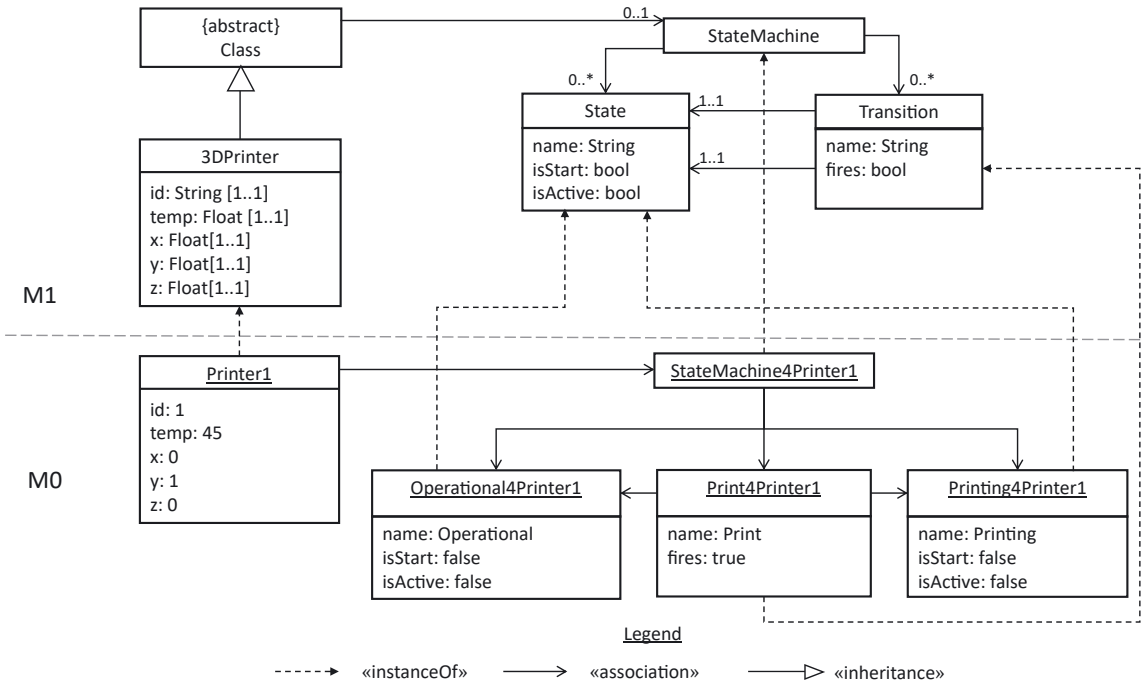

Figure 10: Open behavior model pattern applied to the running example.

directly from the `StateMachine`. In the same way, `Operational4Printer1`, `Printing4Printer1`, and `Print4Printer1` are instantiated from `State` or `Transition` of the behavioral language on M1. As a result, there are no restrictions on how to link these elements on M0. For instance, `Printer1` may be also connected to any other state machine, or `Operational` could be linked via a transition to the `StandBy` state, even though this is not reasonable semantically. In addition, the `isStart` attribute needs to be set for each instantiation of the individual states on M0. However, if the state machine from Figure 10 is extended by, e.g., another state called `Scanning`, this can easily be incorporated on M0 by creating a new instance of `State`, without requiring any adaptations on M1. The same applies to changes in the semantics of the state machine, e.g., changing the start state of the state machine from `StandBy` to `Operational`.

4.2 Representing behavioral traces in Digital Twin platforms

With the patterns described above, we have provided blueprints for augmenting DT models with behavior. However, although these patterns enable the representation of behavior as behavioral traces for one particular point in time (cf. Challenge 1), the history of these traces (cf. Challenge 2) is still an open question. In order to enable the representation of historical traces, we propose two patterns that can be used in addition to the open and closed behavior model patterns.

4.2.1 Temporal annotation-based history pattern

Purpose: The purpose of the Temporal Annotation-Based History Pattern is to represent historical execution traces of behavioral descriptions of DTs, with a focus on scalability concerning storage.

Structure: In this pattern, the used DTML has to be equipped with temporal annotations [20, 21] which are applicable for attributes of elements. In previous work, we have shown that such annotations are commonly available in DT platforms [5]. In case such annotations are not supported, Fowler [22] shows alternatives that can be used in order to implement this kind of support. The application of temporal annotations on the language and model level can be used in order to preserve the history of attribute values (cf. `Model (incl. TA)` in Figure 11). In the runtime traces, temporal annotated attributes contain beside the current model trace (`cT`) also the historical model trace (`hT`), whereas the non-annotated attributes simply contain the current attribute value at the current timestamp (`cT`). The historical model trace consists of a list of timestamps and associated attribute values.

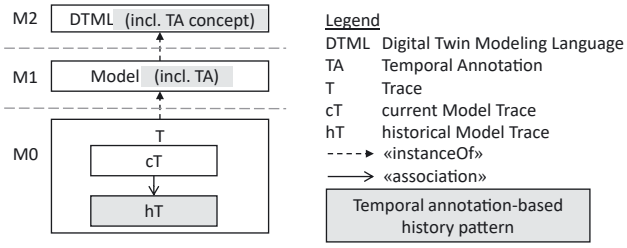

Figure 11: Structure of the temporal annotation-based history pattern.

Consequences: Applying this pattern leads to scalability with respect to storing models and performing certain queries, as the historical traces can be persisted in dedicated databases, such as time-series database (e.g., shown in [17, 23]). It is however more difficult to reproduce complete historical states from these traces, as they only contain the attribute values in a fragmented way, but not the system structure at a given point in time. Thus queries are limited to historical traces without reproducing the complete states as snapshots.

Implementation: Implementing this pattern (cf. Figure 12 for the pattern implementation for the closed behavior model pattern) means that attributes whose history should be traced on M0 have to be annotated as temporal in the `Behavior`. The same is applied to attributes of the `DomainStructure`. On M0, the respective traces contain the current values of attributes that are not annotated as temporal and a collection of historical values of the temporal annotated attributes. These implementation details apply to both the closed and open behavior model pattern. In the closed behavior model pattern, the annotations are simply reused by the `DomainBehavior` from the corresponding `Behavior` on the language level, whereas in the open behavior model pattern, the annotations are directly applied from the `Behavior` to the behavior traces on M0.

Instantiation on the running example: When applying this pattern to the running example, the `isActive` attribute of the `State` and the `fires` attribute of the `Transition` are annotated in the `Behavior` (please recall that the `fire()` operation from Figure 5 is represented by the `fire` attribute that is set to true every time the respective operation is called as we are in the observation setting which produces only data-oriented views). Additionally, the `temp` attribute of the `3DPrinter` in the `DomainStructure` is annotated as well. As a result, the structural and behavioral traces on M0 contain a reference to the historical values for these annotated elements (`isActive`, `fires`, `temp`).

4.2.2 Snapshot-based history pattern

Purpose: The Snapshot-Based History Pattern has the purpose of representing historical execution traces of behavioral descriptions of DTs in cases where temporal annotations are not provided by DT platforms or the focus is on querying and reasoning on full model states, and the evolution of these states over time.

Structure: In this pattern (cf. Figure 13), snapshots (S) (as already introduced before for model validation and verification purposes, e.g., see [24, 25]) are integrated into the modeling capabilities of DT platforms using the

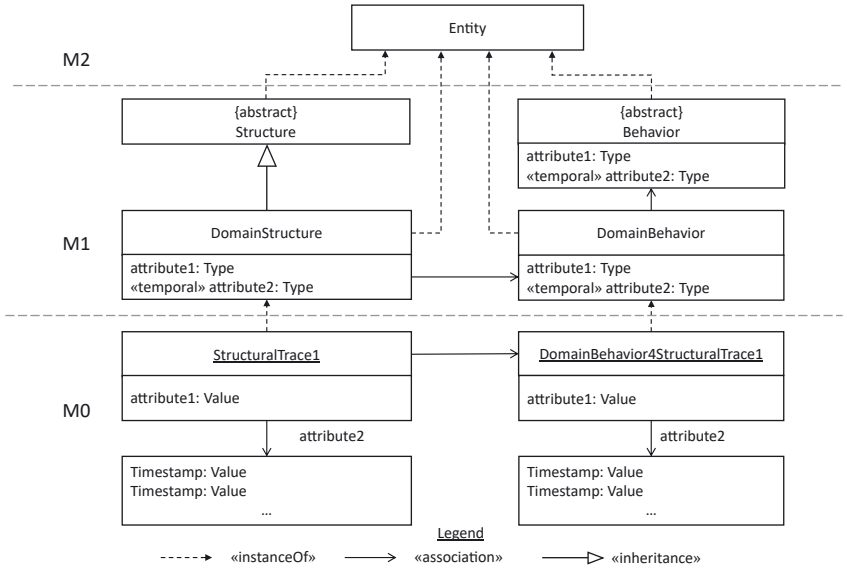

Figure 12: Implementation of the temporal annotation-based history pattern for the closed behavior model pattern.

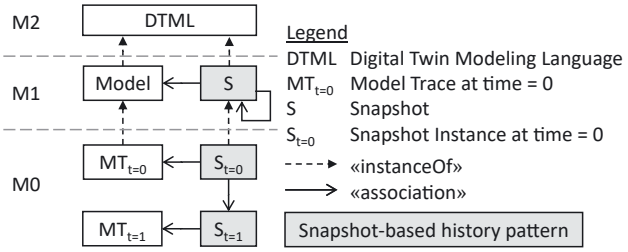

Figure 13: Structure of the snapshot-based history pattern.

blueprints proposed in Section 4.1. Each snapshot has a link to the elements in the model (representing structure or behavior derived from the DTML) and a link to another snapshot representing the previous model state. Thus, on M0, the instantiations of these snapshots for a particular point in time ($S_{t=0}$, $S_{t=1}$) contain the model trace for the respective point in time (i.e., $MT_{t=0}$, $MT_{t=1}$). In order to use this snapshot representation to log system traces, the individual snapshots on M0 are linked with each other to navigate from a specific snapshot to its predecessor and successor.

Consequences: Applying this pattern induces some additional modeling effort for creating snapshots. Additionally, all snapshots are instantiated and loaded in-memory, which means reduced scalability with respect to the size of the execution trace. However, explicitly storing connections between elements in historical states enables querying and reasoning about a particular model state or the evolution of model states over time.

Implementation: In this pattern, the snapshots need to be created on the M0 level based on their definition at M1. Therefore, structural and behavioral traces must be orchestrated to snapshots (cf. Figure 14 for details on the implementation of this pattern in combination with the closed behavior model pattern). In the closed behavior model pattern, the definition of the snapshot is associated with the `DomainStructure` and the `DomainBehavior` on M1. In the open behavior model pattern, these connections of the Snapshot on M1 are already defined at the language level (i.e., `Structure` and `Behavior`).

Instantiation on the running example: Applying the current pattern to the running example means that one snapshot contains the state of `Printer1`, the associated `StateMachine4Printer1`, and the contents of `StateMachine4Printer1` at a given point in time with a reference to the respective predecessor and successor snapshots. For instance, in one snapshot, the `Operational4Printer1` trace can be active, the Print4Printer1 trace is not firing, and the `Printing4Printer1` trace is not active. In the successor snapshot, the `Print4Printer1` is now firing, changing `Operational4Printer1` to non-active and `Printing4Printer1` to active.

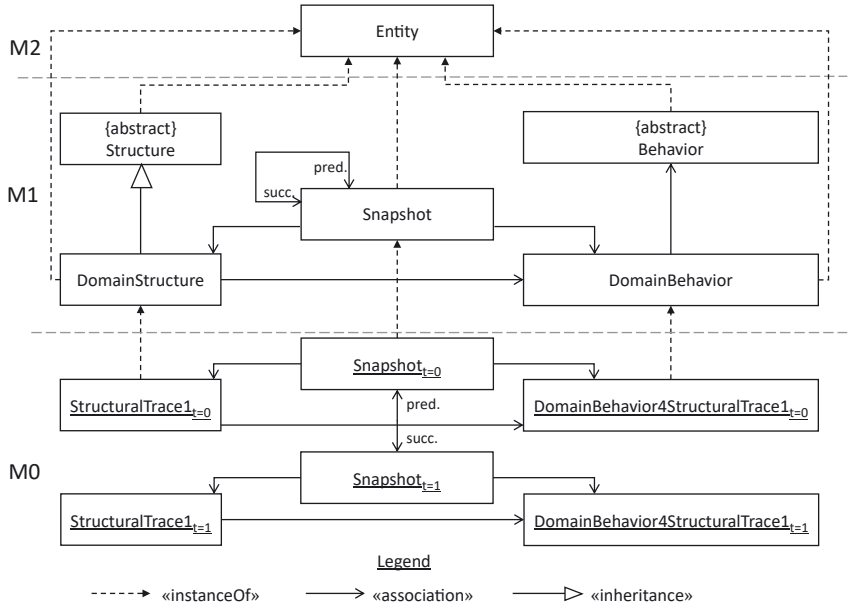

Figure 14: Implementation of the snapshot-based history pattern for the closed behavior model pattern.

5 Demonstration case: Digital Twin of a 3D printer

We applied the patterns presented above to two different use cases to validate their applicability, demonstrate their trade-offs, and provide the resulting models in an online repository.[9] The purpose of this section is to demonstrate the applicability of the patterns using an excerpt of the models in the mentioned repository. More precisely, we present the application of the open and closed behavior model patterns to the aforementioned 3D Printer running example and its realization in the Microsoft Azure DT platform. We also compare the modeling effort required in both patterns to create the 3D printer model as outlined in Section 3.

5.1 Setup

We create a reference implementation of the class and state machine diagram outlined in Figure 3 using the Eclipse Modeling Framework[10] (EMF). It serves as base information to be projected from the four-layer metamodeling stack (Figure 1) to the presented patterns. The EMF models are then translated to represent the same information in a DT platform. To this end, we show the application of both the closed behavior model pattern and open behavior model pattern by creating the respective DTDL models for the Microsoft Azure DT Platform.

In DTDL, a set of metamodel classes constitutes the elements for modeling the parts of the DT infrastructure and their interconnections. Thereby, the `Interface` class defines the `contents` of any DT, that is, e.g., its components, properties, and relationships. Take the definition of the 3DPrinter in Listing 1 for an example. It includes properties for the temperature and coordinates of the axis positioning system. Furthermore, the `Command` class is used to describe operations supported by the DT. This allows for a higher-level view of an asset's functionality that is decoupled from the underlying behavior, in this case, the state machine of the printer. A `Relationship` denotes a link to another DT. Temporal information, i.e., properties that will be emitted and ingested regularly at runtime, such as the temperature and axis properties, are declared using the in-built `Telemetry` class. To this effect, emerging data can be persisted and leveraged for analysis, e.g., using the Time Series Insights database.[11]

In the following, we describe the design-time elements required to model the behavior of the 3D printer and how it is linked to the structural model in Listing 1. We thereby apply the open and closed behavior model pattern in the DTDL, in contrast to modeling this example in EMF. Thereafter, the created DTDL models are imported into the Azure DTs Explorer[12] in order to validate their usage within the DT platform provided by Microsoft Azure. To assess the modeling efforts, we also determine the number of modeling elements required for the complete realization of the 3D printer example.

5.2 Twin model at design-time

Next, we describe the application of the closed and open behavior model pattern in combination with the temporal annotation history pattern for realizing the conceptual description of a system.

Closed behavior model. Listing 2 shows an excerpt of the interfaces necessary for modeling the 3D printer and its state machine according to the closed behavior model pattern. The class `3DPrinter` corresponds to the definition in Listing 1, but additionally derives from `Class` and holds a relationship to `PrinterStateMachine`. Consequently, printer instances can only refer to state machines of this type, whose function as such is defined via the `type` relation to `StateMachine`. Both 3DPrinter and PrinterStateMachine form the base elements to derive structural and behavioral twin representations, respectively. The latter is defined by relations to states and transitions, specifically, the ones shown in Figure 3b, e.g., `Operational`, which also contain the corresponding properties of interest. Again, the semantic assignment is done via type relations. Following the closed behavior model pattern, as with the restriction of printer instances to PrinterStateMachines, the relationships between states and transitions of the latter are rigid. Hence, based on the definitions of corresponding relationships for a transition, it can only lead from the defined incoming state to the defined outgoing state. This is ensured by the semantic type assignment of behavioral elements via corresponding relations (cf. {`unsettable`} in Figure 8) within class definitions rather than deriving from the general type classes. Although the inheritance of an interface is supported in DTDL, e.g., Operational as a descendant of `State` or PrinterStateMachine as a descendant of `StateMachine`, it would no longer limit the scope to the intended workflow for printer instances in the runtime model.

9 https://github.com/cdl-mint/at-journal.

10 https://www.eclipse.org/modeling/emf/.

11 https://azure.microsoft.com/en-gb/products/time-series-insights/.

12 https://azure.microsoft.com/products/digital-twins/.

Table 1: Elements required for modeling the structure and behavior of the 3D printer example in EMF on the meta-model (M2) and model (M1) level.

Meta-level	# Elements
M2	6 classes/9 references/12 attributes
M1	37 objects/66 links/77 values

Table 2: Interfaces (top-level elements) and Interface Contents (Properties, Telemetries, Commands, and Relationships) for modeling the proposed open behavior model pattern (cf. open) and closed behavior model pattern (cf. closed) in DTDL.

Pattern	# Interfaces	# Interface contents
Open	5	18
Closed	28	123

Open behavior model. The interfaces for realizing the open behavior model pattern are defined in Listing 3. Here, the general types `Class` and `StateMachine` constitute the base elements to derive structural and behavioral twin representations. The former relates to the latter with the relationship `stateMachine`, thereby allowing for a behavior model to be defined for any descendants extending the Class interface, e.g., `3DPrinter`. Accordingly, those classes that represent the twin of an asset in the present domain are to be derived from Class. The state machine is described by relations to `State` and `Transition`, which carry the relevant properties and telemetries. However, using the open behavior model pattern, the actual behavioral elements depending on the domain are not specified at this level. The specification for the printer, i.e., the states and transitions reflecting the dedicated workflow, is only done at the instance level.

Comparison. Concerning modeling effort, the interfaces required for the closed behavior model pattern (Table 2) are similar to the objects required on M1 in EMF (Table 1). Furthermore, the required links and values add up to 143, which is roughly equal to the total number of content items in the interfaces, which are composed of relationships and properties. This results from mimicking the semantics of the EMF implementation as closely as possible. In the open behavior model pattern, this effort is reduced considerably, even leading to less interfaces being created than on M2 alone in EMF. The discrepancy in the modeling effort between open and closed behavior model pattern (Table 2) becomes even more evident when considering the number of interface contents that must be created in DTDL to create the model for the 3D printer example.

5.3 Twin instance at runtime

Regardless of whether the open or closed design pattern is used, the DT infrastructure will superficially host the same DT instances at runtime. However, each pattern yields different type assignments in the behavioral model and the therewith associated consequences discussed in Section 4. Figure 15 shows an excerpt respectively for the instantiated models provided in Listings 2 and 3, to realize the running example. In both cases the environment comprises a single printer “p1” whose internal workflow is represented by its state machine “psm1”. States and transitions of the latter such as Operational, Print, and Printing (cf. Figure 3b), are each dedicated to a DT instance. Following the closed behavior model pattern, Figure 15a shows these instances being initialized using their respective interfaces, and connected in conformance with the typed relations of the transition `print`, that is, to state instances `operational` and `printing`. In Figure 15b, the same is shown for the open behavior model pattern, with all the behavior-related

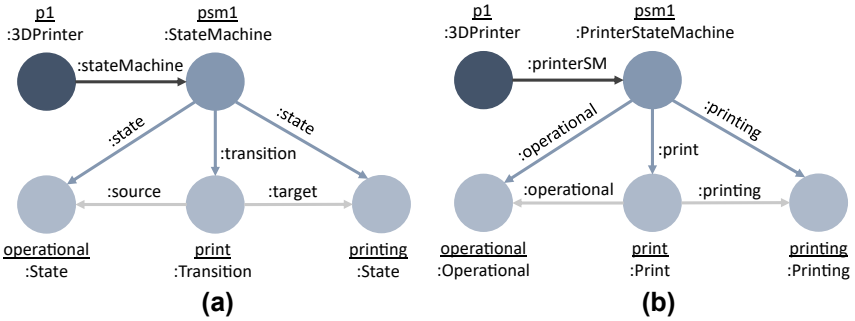

Figure 15: Excerpt of the twin instances for the 3D printer example in a graph-based notation. (a) Closed behavior model pattern, (b) Open behavior model pattern.

instances and relations based on the general classes `State`, `Transition`, and `StateMachine`.

Listing 2 JSON Code excerpt for the running example; closed behavior model pattern.

```
{"@type": "Interface",
 "displayName": "StateMachine",
 "@id": "dtmi:com:cdl:StateMachine;2" },
{"@type": "Interface",
 "displayName": "State",
 "@id": "dtmi:com:cdl:State;2" },
{"@type": "Interface",
 "displayName": "3DPrinter",
 "@id": "dtmi:com:3DPrinter;2",
 "contents":[{
   "@type": "Relationship",
   "target": "dtmi:com:PrinterSM;2",
   "name": "printerStateMachine" },
   ... ] },
{"@type": "Interface",
 "displayName": "PrinterStateMachine",
 "@id": "dtmi:com:PrinterSM;2",
 "contents":[{
   "@type": "Relationship",
   "target": "dtmi:com:StateMachine;2",
   "name": "type"}, {
   "@type": "Relationship",
   "target": "dtmi:com:Operational;2",
   "name": "operational"},
    ... ] },
{"@type": "Interface",
 "displayName": "Operational",
 "@id": "dtmi:com:Operational;2",
 "contents":[{
   "@type": "Relationship",
   "target": "dtmi:com:State;2",
   "name": "type" },
    ... ] }
```

Listing 3 JSON Code excerpt for the running example; open behavior model pattern.

```
{"@type": "Interface",
 "displayName": "Class",
 "@id": "dtmi:com:cdl:Class;2",
 "contents":[{
   "@type": "Relationship",
   "name": "stateMachine",
   "target": "dtmi:com:cdl:SM;2" }] },
{"@type": "Interface",
 "displayName": "3DPrinter",
 "extends": "dtmi:com:cdl:Class;2",
 "@id": "dtmi:com:cdl:D3Printer;2",
 "contents": [ ... ] },
{"@type": "Interface",
 "displayName": "StateMachine",
 "@id": "dtmi:com:cdl:SM;2",
 "contents":[{
   "@type": "Relationship",
   "target": "dtmi:com:cdl:State;2",
   "name": "state" }, {
   "@type": "Relationship",
   "target": "dtmi:com:cdl:Transition;2",
   "name": "transition" }] },
{"@type": "Interface",
 "displayName": "State",
 "@id": "dtmi:com:cdl:State;2",
 "contents": [ ... ] },
{"@type": "Interface",
 "displayName": "Transition",
 "@id": "dtmi:com:cdl:Transition;2",
 "contents":[ ... ] }
```

6 Discussion and limitations

This section critically discusses the presented patterns and the limitations of this work.

6.1 Pattern comparison

The demonstration of the presented patterns for a specific case using a current DT platform shows their applicability. In this respect, the 3D printer example leads to observations concerning initial modeling effort and model maintenance during design and runtime.

The closed behavior model pattern requires dedicated modeling effort on M1 by explicitly specifying the behavior model, which is not required by the open behavior model pattern. On M0, however, the effort is reduced by the information already available at M1, e.g., default values for properties, such as whether a state acts as the initial state for a state machine or not. In contrast, using the open behavior model pattern, it has to be set manually for each instantiated state machine for each structural entity on M0. This becomes more significant as the number of entities to be created increases. Thus, with respect to the modeling effort and rigor, there is a trade-off between the initial effort for modeling M1 and the long-term effort for modeling the runtime on M0. However, model transformations may be developed in order to automatically populate the model fragments used for representing the behavior models on M0 for both patterns.

Besides the modeling effort for creating an initial system, the patterns further differ with respect to the flexibility they imply towards evolving the created model (e.g., adding states or changing the starting state of a state machine). In this regard, the change effort varies for adapting the DT model to system changes that occur over time. Please note the flexibility of the open behavior model pattern as the runtime model can be changed directly on M0. With the closed behavior model pattern, such changes must first be carried out on M1. Performing these changes on M1, however, (*i*) requires redeploying the model to the DT platform, and (*ii*) breaks instance-level relations of the existing elements on M0 to the respective model elements on M1. Accordingly, elements on M0 have to be adapted as well in order to represent a valid model again.

As an alternative to temporal annotations to represent historical traces, a snapshot-based representation may be used. This can be achieved by defining an additional interface type on M1 that is associated with objects whose state should be recorded. Following this approach, objects would have to be replicated and linked to a new snapshot instance again and again. As a benefit, queries about object states including their behavioral states are supported, providing insights about the complete state of the system over time in a coherent structure. However, for each change in the system, the whole snapshot (i.e., the current structure and behavior of the system) must be stored. This may lead to a considerable overhead with respect to memory consumption compared to storing the value updates individually for each annotated property, as would be the case using the temporal annotation pattern.

With respect to data history and storage, the Azure DT platform has also implemented a realization.[13] However, it only deals with lifecycle events in a rudimentary way, which include creation and deletion of DTs. Through our presented patterns, the functionalities are extended and a variety of states can be reflected.

6.2 Limitations

We now point out limitations with respect to the scope of our study as well as the potential applicability of the presented patterns. For this, we discuss the following two threats to validity types.

Internal validity. In this work, we do not demonstrate the application of the patterns using a concrete runtime simulation nor do we evaluate the resulting solutions using software complexity metrics. Therefore, no conclusions can be made yet about the quality criteria such as performance, maintainability, and scalability, with respect to the involved DT infrastructures. Indeed, the extension of the DT model for behavioral properties leads to a higher data volume, which consequently leads to higher data storage and retrieval costs, especially with respect to the complete representation of the system state in the snapshot-based pattern. We plan to investigate these aspects and to utilize the obtained behavioral information in further work. Moreover, the patterns, especially the closed behavior model pattern, may benefit from multi-level modeling features. Substituting the instanceOf relationship with an association is clearly a workaround and only provides meta-information for the modeling elements but no support from a type system perspective. Furthermore, certain attributes should have only values on a certain level and not on others, e.g., the attribute isActive should only get a value assigned on M0. Future studies are required to evaluate the application of multi-level modeling concepts for the presented patterns such as having more flexible instantiation levels and concepts such

13 https://learn.microsoft.com/en-US/azure/digital-twins/concepts-data-history.

as potency to control the usage of attributes on dedicated levels.

External validity. In our demonstration of the presented patterns for state machines, we present some general structures which may be also utilized for other behavioral modeling languages. However, the general suitability of the presented patterns for other languages such as other discrete behavior modeling languages, e.g., activity diagrams, sequence diagrams, or continuous behavior modeling languages has to be studied in future work. Especially for the latter, there are optimized modeling languages and tools for example Modelica[14] or Matlab Simulink.[15] These languages also have their own mechanisms and support for mapping DTs and handling behavioral simulations of systems.[16] Besides using other modeling languages, targeting other DT platforms with the presented patterns is required to generate further evidence that the presented patterns are general enough to be usable in other contexts. In particular, the patterns shown assume the availability of certain modeling concepts in the target DT platform. This becomes evident in our example about the implementation of the closed behavior model pattern on M1 in DTDL, which does not support any concept for defining abstract interfaces. In this respect, interfaces like "Class" can also be instantiated in the runtime model on M0, which is only intended to serve for the type assignment. Furthermore, it is neither possible to set property values on M1 nor to declare them as unchangeable, which means that the relevant property in the start state, i.e., "isStart" in our example, cannot be set on M1. Anticipating these limitations, the model as well as runtime instances could be created in EMF, which supports these concepts, and then be transformed into the DTML of the target platform ready for deployment. However, this raises the question of how the DT model will be updated as the system changes. For the actual reproduction, the patterns nevertheless require a minimum set of concepts, the absence of which must be specifically dealt with. Moreover, the applicability of the presented patterns in other DT platforms seems promising based on the results from a previous study [5] about the modeling features of DT platforms. However, additional studies are required to provide evidence that the patterns can be indeed realized by the modeling features of the other platforms. Finally, future studies are needed to validate the applicability and performance of the proposed patterns for the different DT types and architectures as described in [26].

14 https://modelica.org/.

15 https://www.mathworks.com/.

16 https://www.mathworks.com/discovery/digital-twin.html.

7 Related work

Regarding related work, we first consider work which is highlighting the need for behavioral models in DTs. Then, we discuss model-driven approaches for engineering DTs. Finally, we discuss semantic lifting approaches for runtime data. For a general overview of the various disciplines that have adopted DTs, and the applications therein, the interested reader is referred to [3, 6].

Addressing behavioral models for Digital Twins. The need for the availability of behavioral models in DT systems is becoming increasingly important. Rovere et al. [27] outline a supporting infrastructure for managing DTs, emphasizing the relevance of behavioral models for enabling simulations and the accessibility of these models throughout the factory lifecycle. Building on the DEVS formalism, Niyonkuru & Wainer [28] present an environment that includes DTs but also foresees a physical model to support simulations and study behavior under real-world conditions, e.g., time constraints in real-time systems. Using Reinforcement Learning, a model was learned from data collected at runtime and used to optimize behavior [29, 30]. Stary et al. [31] take a human-centric view of DTs by proposing behavioral models for capturing components and interactions in a Cyber-Physical System (CPS). They provide a subject-oriented approach for modeling both the structure and behavioral aspects of the system, aiming to provide a unified view for the various stakeholders involved in the design process of DTs. Tekinerdogan and Verdouw present a design pattern catalog for developing DTs [26]. Different from our mapping of specific object behaviors, the authors define the general behavior of DTs using Sequence Diagrams. In [32], Verdouw et al. discuss the typologies of different types of DTs which also utilize behavioral models in the context of smart farming. In contrast to our approach, the mentioned works address the requirements for DTs in terms of planning, validation, and implementation at a conceptual level rather than concrete blueprints for capturing DTs in existing DT platforms.

Model-driven Digital Twin engineering. In the area of MDE for DTs, there are several different approaches to facilitate their development and deployment. Whereas Bordeleau et al. [33] give an overview of opportunities and challenges for integrating MDE and DTs, there are also many concrete implementations available already. For instance, Muñoz et al. [34] engineer DTs with UML and apply structural system snapshots for the runtime information. Binder

et al. [35] present an approach that automatically transforms the logical architecture of a system into the technical implementation using AutomationML (AML) and evaluate it by a case study according to the concepts of the Reference Architecture Model Industry 4.0 (RAMI 4.0). Zhang et al. [36] also present an approach for Cyber-Physical Production Systems (CPPS) information modeling based on DTs and AML. They show an integration of various physical resources into CPPS by DTs and AML. Mazak et al. [37] take an approach in this area where the strengths of AutomationML and MDE are combined to reduce the manual effort required to implement a runtime data acquisition system and simplify subsequent analyses. In the area of communication and data exchange between systems and DTs, there are also other approaches [38, 39] that exploit the use of higher-level models (e.g., AML models). In addition, these models are used to generate web services and visualization.

Regarding the modeling of DTs, Tao et al. [6] give an overview of current modeling techniques and tools, and Atkinson and Kühne [40] point out shortcomings of current modeling standards for improving them with respect to (*i*) the rigid heterogeneous technology stack and (*ii*) the evolving nature of DTs. They propose a multi-level modeling approach with a universal language as one possible countermeasure, e.g., to preserve consistency while accommodating different abstraction levels and to enforce typing constraints along hierarchy levels. Our suggested patterns follow this general direction by providing a solution to overcome the hierarchical constraint one faces with current DTMLs that are set to a 2-level modeling approach for behavioral models. However, we consider the usage of multi-level modeling techniques for behavioral models as an interesting research line, e.g., how to integrate multi-level concepts in current platforms which then can be used for developing behavioral modeling support. Especially, the closed behavior model pattern would benefit from such approaches. In Bibow et al. [14] events are logged to detect when, e.g., properties are changed in a CPPS. For this purpose, a domain-specific language is specified that defines the communication between the system and its DT using OPC-UA. Brockhoff et al. [41] address the intersection of DTs and process mining, emphasizing the need for a common view regarding architectural design. In their proposed architecture they aim at self-adaptive DTs, therefore adopting a generative approach that enriches MDE capabilities with process mining techniques to exploit the obtained data. Similar to [14], they plan to react when predefined changes are detected. Our approach on the other hand aims at making temporal aspects traceable by integrating behavior models into the DT model and providing a general architecture for DTs with an explicit behavior view that could also support evolution in the future.

Semantically lifting runtime data. Processing recorded data against a knowledge base provides another way to comprehend a system's behavior or even respond to it's evolution. This is based on the formalization of domain semantics, e.g., using the Resource Description Framework[17] (RDF). In [42], the authors propose an RDF-based representation of the Asset Administration Shell, a standard for describing assets in Industry 4.0. Their vision entails automated integration of various assets as well as validation of constraints and reasoning over the data model. Kamburjan et al. [43, 44] use knowledge graphs to align the DT with its physical counterpart. To this end, the DT infrastructure and simulation models are defined using the Semantic Micro Object Language [45]. A language-based approach bridging simulators with formalized domain knowledge is shown in [43]. In [44], the DT infrastructure is integrated with asset models, which are descriptions of the physical assets, in order to form a knowledge graph. The runtime state will be lifted into this graph to detect changes, and determine and perform an appropriate reconfiguration of the DT. Adaptation of structure and simulation models takes place by querying and manipulating the knowledge graph. In our view, the behavior could also be tracked from the model properties and the knowledge graph using this method. With our pattern, in contrast, the injection of behavioral elements into the model is done directly, which allows mapping from other behavioral modeling languages. As a result, a flexible environment is provided for data processing in terms of querying, validation, and simulation.

Synopsis. While there have been several efforts of using model-driven engineering for DTs and interpreting runtime data on a semantically-enriched level, we are not aware of any existing work which is integrating the concept of behavioral models in current DT platforms. The presented work of this paper aims to fill this gap.

8 Conclusions and future work

In this paper, we have presented several patterns on how to augment DT models with behavior, covering design-time and runtime viewpoints of current DT platforms. The patterns have been illustrated for state machines and with a demonstration case using the Microsoft Azure DT platform.

17 https://www.w3.org/RDF.

In future work, we consider the following lines of research. First, we plan to perform studies about the scalability of the presented patterns concerning runtime data storage and query performance, e.g., of KPIs. Second, integrating other viewpoints such as process or organizational viewpoints would allow sophisticated interfaces to DTs for higher layers of the automation pyramid. Finally, the usage of multi-level modeling as discussed in [11, 40] within DT platforms becomes of interest as it would allow using several instantiation levels more explicitly. By this mechanism, more product-line-aware DTs which also incorporate behavioral models may be envisioned.

Author contributions: All the authors have accepted responsibility for the entire content of this submitted manuscript and approved submission.
Research funding: This work has been supported by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development (CDG).
Conflict of interest statement: The authors declare no conflicts of interest regarding this article.

References

[1] C. Yang, S. Lan, W. Shen, L. Wang, and G. Q. Huang, "Software-defined cloud manufacturing with edge computing for Industry 4.0," in *Proc. of the 16th International Wireless Communications and Mobile Computing Conference (IWCMC)*, IEEE, 2020, pp. 1618—1623.
[2] W. Kritzinger, M. Karner, G. Traar, J. Henjes, and W. Sihn, "Digital Twin in manufacturing: a categorical literature review and classification," *IFAC-PapersOnLine*, vol. 51, no. 11, pp. 1016—1022, 2018.
[3] M. Dalibor, N. Jansen, B. Rumpe, et al., "A cross-domain systematic mapping study on software engineering for digital twins," *J. Syst. Softw.*, vol. 193, p. 111361, 2022.
[4] F. Tao, H. Zhang, A. Liu, and A. Y. C. Nee, "Digital twin in Industry: state-of-the-art," *IEEE Trans. Ind. Inf.*, vol. 15, no. 4, pp. 2405—2415, 2019.
[5] J. Pfeiffer, D. Lehner, A. Wortmann, and M. Wimmer, "Modeling capabilities of digital twin platforms — old wine in new bottles?" *J. Object Technol.*, vol. 21, no. 3, pp. 3:1—14, 2022.
[6] F. Tao, B. Xiao, Q. Qi, J. Cheng, and P. Ji, "Digital twin modeling," *J. Manuf. Syst.*, vol. 64, pp. 372—389, 2022.
[7] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," in *Synthesis Lectures on Software Engineering*, 2nd ed. Kentfield, Morgan & Claypool Publishers, 2017.
[8] J. Bezivin, "On the unification power of models," *Softw. Syst. Model.*, vol. 4, no. 2, pp. 171—188, 2005.
[9] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins, "Modeling software architectures in the unified Modeling Language," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 1, pp. 2—57, 2002.
[10] T. Kühne, "Matters of (meta-)modeling," *Softw. Syst. Model.*, vol. 5, no. 4, pp. 369—385, 2006.
[11] C. Atkinson, R. Gerbig, and T. Kühne, "Comparing multi-level modeling approaches," in *Proc. of the Workshop on Multi-Level Modelling co-located with MoDELS*, CEUR-WS.org, 2014, pp. 53—61.
[12] C. Atkinson and T. Kühne, "Reducing accidental complexity in domain models," *Softw. Syst. Model.*, vol. 7, no. 3, pp. 345—359, 2008.
[13] T. Kühne, "Multi-dimensional multi-level modeling," *Softw. Syst. Model.*, vol. 21, no. 2, pp. 543—559, 2022.
[14] P. Bibow, M. Dalibor, C. Hopmann, et al., "Model-driven development of a digital twin for injection molding," in *Proc. of the International Conference on Advanced Information Systems Engineering (CAiSE)*, Springer, 2020, pp. 85—100.
[15] J. C. Kirchhof, J. Michael, B. Rumpe, S. Varga, and A. Wortmann, "Model-driven digital twin construction: synthesizing the integration of cyber-physical systems with their information systems," in *Proc. of the ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MoDELS '20)*, ACM, 2020, pp. 90—101.
[16] D. Lehner, J. Pfeiffer, E. Tinsel, et al., "Digital twin platforms: requirements, capabilities, and future prospects," *IEEE Softw.*, vol. 39, no. 2, pp. 53—61, 2022.
[17] D. Lehner, S. Sint, M. Vierhauser, W. Narzt, and M. Wimmer," AML4DT: a model-driven Framework for developing and maintaining digital twins with AutomationML," in *Proc. of the 26th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2021*, IEEE, 2021, pp. 1—8.
[18] B. Combemale, O. Barais, and A. Wortmann, "language engineering with the GEMOC studio," in *Proc. of IEEE International Conference on Software Architecture Workshops, ICSA Workshops*, IEEE, 2017, pp. 189—191.
[19] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides, "Design patterns: abstraction and reuse of object-oriented design," in *Proc. of the 7th European Conference on Object-Oriented Programming (ECOOP)*, Springer, 1993, pp. 406—431.
[20] J. Cabot, A. Olivé, and E. Teniente, "Representing temporal information in UML," in *Proc. of the 6th International Conference on the Unified Modeling Language, Modeling Languages and Applications*, Springer, 2003, pp. 44—59.
[21] A. Gómez, J. Cabot, and M. Wimmer, "TemporalEMF: a temporal metamodeling framework," in *Proc. of the 37th International Conference on Conceptual Modeling (ER)*, Xi'an, China, Springer, 2018, pp. 365—381.
[22] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, Addison-Wesley, 2012.
[23] A. Mazak, S. Wolny, A. Gómez, J. Cabot, M. Wimmer, and G. Kappel, "Temporal models on time series databases," *J. Object Technol.*, vol. 19, no. 3, pp. 3:1—15, 2020.
[24] M. Gogolla, J. Bohling, and M. Richters, "Validating UML and OCL models in USE by automatic snapshot generation," *Softw. Syst. Model.*, vol. 4, no. 4, pp. 386—398, 2005.
[25] F. Hilken, L. Hamann, and M. Gogolla, "Transformation of UML and OCL models into filmstrip models," in *Proc. of the 7th International Conference on Theory and Practice of Model Transformations (ICMT)*, Springer, 2014, pp. 170—185.
[26] B. Tekinerdogan and C. Verdouw, "Systems architecture design pattern catalog for developing digital twins," *Sensors*, vol. 20, no. 18, p. 5103, 2020.

[27] G. Dal Maso, D. Rovere, P. Pedrazzoli, M. Alge, and M. Ciavotta, *A Centralized Support Infrastructure (CSI) to Manage CPS Digital Twin, towards the Synchronization between CPSs Deployed on the Shopfloor and Their Digital Representation*. Gistrup, River Publishers, 2019, pp. 317—335.
[28] D. Niyonkuru and G. A. Wainer, "A devs-based engine for building digital quadruplets," *Simul.*, vol. 97, no. 7, pp. 2021—2506, 2021.
[29] C. Cronrath, A. R. Aderiani, and B. Lennartson, "Enhancing digital twins through reinforcement learning," in *15th IEEE International Conference on Automation Science and Engineering (CASE)*, IEEE, 2019, pp. 293—298.
[30] N. Tomin, V. Kurbatsky, V. Borisov, and S. Musalev, "Development of digital twin for load center on the example of distribution network of an urban district," *E3S Web Conf.*, vol. 209, p. 02029, 2020.
[31] C. Stary, M. Elstermann, A. Fleischmann, and W. Schmidt, "Behavior-centered digital-twin design for dynamic cyber-physical system development," *Complex Syst. Informatics Model. Q.*, vol. 30, pp. 31—52, 2022.
[32] C. Verdouw, B. Tekinerdogan, A. Beulens, and S. Wolfert, "Digital twins in smart farming," *Agric. Syst.*, vol. 189, p. 103046, 2021.
[33] F. Bordeleau, B. Combemale, R. Eramo, M. van den Brand, and M. Wimmer, "Towards model-driven digital twin engineering: current opportunities and future challenges," in *Proc. of the First International Conference on Systems Modelling and Management (ICSMM)*, Springer, 2020, pp. 43—54.
[34] P. Muñoz, J. Troya, and A. Vallecillo, "Using UML and OCL models to realize high-level digital twins," in *Proc. of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS)*, IEEE, 2021, pp. 212—220.
[35] C. Binder, A. Calà, J. Vollmar, C. Neureiter, and A. Lüder, "Automated model transformation in modeling digital twins of industrial internet-of-things applications utilizing AutomationML," in *Proc. of the 26th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2021*, IEEE, 2021, pp. 1—6.
[36] H. Zhang, Q. Yan, and Z. Wen, "Information modeling for cyber-physical production system based on digital twin and AutomationML," *Int. J. Adv. Manuf. Technol.*, vol. 107, no. 3, pp. 1927—1945, 2020.
[37] A. Mazak, A. Lüder, S. Wolny, et al., "Model-based generation of run-time data collection systems exploiting AutomationML," *Autom.*, vol. 66, no. 10, pp. 819—833, 2018.
[38] G. N. Schroeder, C. Steinmetz, C. E. Pereira, and D. B. Espindola, "Digital twin data modeling with AutomationML and a communication methodology for data exchange," *IFAC-PapersOnLine*, vol. 49, no. 30, pp. 12—17, 2016.
[39] G. N. Schroeder, C. Steinmetz, R. N. Rodrigues, R. V. B. Henriques, A. Rettberg, and C. E. Pereira, "A methodology for digital twin modeling and deployment for Industry 4.0," *Proc. IEEE*, vol. 109, no. 4, pp. 556—567, 2021.
[40] C. Atkinson and T. Kühne, "Taming the complexity of digital twins," *IEEE Softw.*, vol. 39, no. 2, pp. 27—32, 2022.
[41] T. Brockhoff, M. Heithoff, I. Koren, et al., "Process prediction with digital twins," in *Companion Proc. of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS)*, IEEE, 2021, pp. 182—187.
[42] S. R. Bader and M. Maleshkova, "The semantic asset administration shell," in *Proc. of the 15th International Conference on Semantic Systems (SEMANTiCS)*, Springer, 2019, pp. 159—174.
[43] E. Kamburjan, and E. B. Johnsen, "Knowledge structures over simulation units," in *Annual Modeling and Simulation Conference, ANNSIM 2022*, IEEE, 2022, pp. 78—89.
[44] E. Kamburjan, V. N. Klungre, R. Schlatte, S. L. T. Tarifa, D. Cameron, and E. B. Johnsen, "Digital twin reconfiguration using asset models," in *Proc. of the 11th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, Springer, 2022, pp. 71—88.
[45] E. Kamburjan, V. N. Klungre, R. Schlatte, E. B. Johnsen, and M. Giese, "Programming and debugging with semantically lifted states," in *Proc. of the 18th International Conference on the Semantic Web (ESWC)*, Springer, 2021, pp. 126—142.

Bionotes

Daniel Lehner
JKU Linz Christian Doppler Laboratory for Model-Integrated Smart Production (CDL-MINT), Department of Business Informatics — Software Engineering, Johannes Kepler University Linz, Altenberger Straße 69, 4040 Linz, Austria
daniel.lehner@jku.at

Daniel Lehner is a PhD candidate at the Department of Business Informatics — Software Engineering, and also associated with the Christian Doppler Laboratory for Model-Integrated Smart Production (CDL-MINT), both at Johannes Kepler University Linz. His research interests include applying Model-Driven Engineering techniques and practices to Digital Twins. For more information, please visit https://se.jku.at/daniel-lehner.

Sabine Sint
JKU Linz Christian Doppler Laboratory for Model-Integrated Smart Production (CDL-MINT), Department of Business Informatics — Software Engineering, Johannes Kepler University Linz, Altenberger Straße 69, 4040 Linz, Austria
sabine.sint@jku.at

Sabine Sint is currently working as PhD student in the Christian Doppler Laboratory for Model-Integrated Smart Production (CDL-MINT) at the JKU Linz in the module Reactive Model Repositories. Her topic of interest is SysML-based modeling and execution of complex systems. Since 2013, she has been working in the Research Unit of Building Physics at TU Wien with a focus on project management and developing software solutions for energy-efficient planning of buildings. For more information, please visit https://se.jku.at/sabine-sint.

Martin Eisenberg
JKU Linz Christian Doppler Laboratory for Model-Integrated Smart Production (CDL-MINT), Department of Business Informatics — Software Engineering, Johannes Kepler University Linz, Altenberger Straße 69, 4040 Linz, Austria
martin.eisenberg@jku.at

Martin Eisenberg is currently a master's student in Computer Science. Since joining the CDL-MINT in 2019, he has partaken in research around model-driven technologies and AI applications. His research interests include algorithms for optimization purposes such as in machine processes and applied machine learning. For more information, please visit https://se.jku.at/martin-eisenberg.

Manuel Wimmer
JKU Linz Christian Doppler Laboratory for Model-Integrated Smart Production (CDL-MINT), Department of Business Informatics — Software Engineering, Johannes Kepler University Linz, Altenberger Straße 69, 4040 Linz, Austria
manuel.wimmer@jku.at

Manuel Wimmer is a Full Professor and Head of the Department of Business Informatics — Software Engineering at JKU Linz, Austria. Since 2019, he has been also the Program Director of the Business Informatics master study at JKU Linz. He received his Ph.D. and his Habilitation from TU Wien. He has been a research associate at the University of Malaga, Spain, a visiting professor at the University of Marburg, Germany, and at TU Munich, Germany, and an assistant professor at the Business Informatics Group (BIG), TU Wien, Austria. His research interests include Software Engineering, Model-Driven Engineering, and Cyber-Physical Systems. For more information, please visit https://se.jku.at/manuel-wimmer.