A Method for Template-based Architecture Modeling and its Application to Digital Twins

Daniel Lehner*, Jérôme Pfeiffer†, Stefan Klikovits*, Andreas Wortmann†, and Manuel Wimmer*
*CDL-MINT, Institute for Business Informatics - Software Engineering, Johannes Kepler University Linz, Austria, firstname.lastname@jku.at
†Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW), University of Stuttgart, Germany, firstname.lastname@isw.uni-stuttgart.de

ABSTRACT
Digital Twins (DTs) have recently emerged to support domain experts in engineering and operating Cyber-Physical Systems (CPSs). As a result, software vendors started to create DT services offering advanced functionality to support the development and operation of DTs in the industry. However, the integration of services into a DT architecture is challenging. Services typically rely on specific software and modeling languages that are often not interoperable with other services. Hence, they have to be manually integrated which requires a significant, repetitive effort. Thus, currently, it is tedious to extend the DT's underlying architectures with new services or exchange individual service implementations.
In this paper, we propose a tool-supported method for architecture modeling and its application for digital twins. The presented method provides several steps to manage the complexity of current DT architectures. First, DT reference architectures are assembled by connecting DT templates, which provide an abstraction for a set of similar DT services. Second, dedicated DT modules are used to wrap existing services which provide concrete realizations of the DT templates. Third, a product-line-based generator supports the configuration of reference architectures into concrete architectures by selecting an appropriate set of modules for the used templates which are finally used in the derived integration solution. The transition from reference architecture modeling to the product-line-based configuration is supported by a dedicated model transformation. Our evaluation shows that the proposed DT templates enable the efficient modeling of different DT reference architectures and integration of new DT services into already existing architectures.

KEYWORDS Digital Twin, Model-Driven Engineering, Architecture Modeling, Modeling Method, Software Language Engineering.

1. Introduction

With the emergence of Cyber-Physical Systems (CPSs), more and more applications are being developed that utilize the collected runtime data, ranging from monitoring to predictive or self-adaptive solutions. To handle the complexity induced by the variety of available CPSs and applications, Digital Twins (DTs) have emerged. A DT provides access to runtime data of a CPS (Kritzinger et al. 2018). This data can then be used by various services to realize applications that use collected runtime data. To support the implementation of such DTs in a scalable way, several DT platforms, e.g., Microsoft Azure Digital Twins[1], Eclipse Vorto[2], or AWS TwinMaker[3], have emerged recently (Lehner et al. 2022).

These platforms enable the exchange of data with a physical twin based on structural models, which are reminiscent of the core of UML class diagrams (Pfeiffer et al. 2022). Nevertheless, their provided functionality is not sufficient to manage the full complexity of DTs (Atkinson & Kühne 2021). To provide advanced functionality, the addition of further *DT services*, such

JOT reference format:
Daniel Lehner, Jérôme Pfeiffer, Stefan Klikovits, Andreas Wortmann, and Manuel Wimmer. *A Method for Template-based Architecture Modeling: and its Application to Digital Twins*. Journal of Object Technology. Vol. 23, No. 3, 2024. Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2024.23.3.a8

[1] https://azure.microsoft.com/en-au/products/digital-twins/
[2] https://www.eclipse.org/vorto/
[3] https://aws.amazon.com/iot-twinmaker/

as simulation or monitoring, is needed (Dalibor, Jansen, et al. 2022). For example, to add simulation functionalities, we may use a model execution engine as a service that leverages behavioral models of the respective physical twin. To utilize runtime data within this simulation, we have to integrate the model execution engine with a DT platform which is able to provide the runtime data. We also have to integrate the used behavioral modeling language (e.g., state machines or rule-based languages) with the data modeling language used by the DT platform. Typically, manually performing such integration is labor-intensive and error-prone as the required glue code is, of course, not reusable. This situation is further aggravated when a DT architecture evolves, e.g., when it is extended with new services or an existing service or when a DT platform is exchanged with an alternative. Thus, the pairwise integration of each service with each other drastically increases the complexity of creating and evolving DT architectures.

To better cope with the complexity of extending different DT platforms with services that provide additional functionality into DT architectures (i.e., architectures that leverage runtime data of physical twins based on DT platforms), we propose a comprehensive tool-supported method to systematically manage a set of DT architectures and their evolution, based on ideas first outlined in (Pfeiffer et al. 2023). In Phase 1 of our method, we leverage a templating mechanism to generalize services with similar functionality. In Phase 2, we integrate different templates into a Reference Architecture Definition (RAD). Based on this RAD and the available modules implementing the selected templates and wrapping existing services, we automatically derive a product line of available service combinations for the selected RAD. After choosing a set of services from this product line in Phase 3, we generate a DT architecture that integrates the selected services. In this phase, the necessary glue code is generated automatically based on the RAD specification. With our method, we go beyond the state-of-the-art as the template-based reference architecture modeling method applies to any reference architecture, and thus, is not restricted to one particular reference architecture such as MAPE-K architectures as it is the case in (Dalibor, Heithoff, et al. 2022). We evaluate the presented method against the state-of-the-art by investigating several scenarios for modeling DT architectures we defined during discussions with an industry partner in the SofDCar project. Our evaluation results indicate that even though our template-based approach requires the definition of templates as an additional step, it provides better scalability concerning an increasing number of DT architectures or RADs.

The rest of this paper is structured as follows. Section 2 introduces a motivating example consisting of scenarios for modeling variants of DT architectures that we derived from discussions with an industry partner. Then, Section 3 presents our template-based architecture modeling method. In Section 4, we instantiate our method in the context of DTs. Section 5 evaluates our method using the scenarios introduced in Section 2. Section 6 discusses related work. Finally, Section 6 concludes the paper.

2. Motivating Example

We now proceed with the motivation for dedicated support for modeling DT architectures using two scenarios that we derived from discussions with an industry partner. The scenarios are described with the help of a representative example.

2.1. Software-Defined Car Research Project

The Software-Defined Car[4] (SofDCar) project is a government-sponsored research program. It is sponsored by the German Federal Ministry of Economic Affairs and Climate Action (BMWK). One of its four workstreams is concerned with building a digital twin for vehicles. The digital twin should tackle the challenges of constantly increasing and changing customer demands and expectations on modern vehicles. Our industry partner, one of the members of this workstream, plans to use the resulting software solutions to provide architectures for such DTs of cars to their customers. However, the specific requirements of individual customers lead to different DT architecture variants that have to be managed by our industry partner. Additionally, the evolution of services and their functionality also required the adaptation of existing DT architecture variants over time. Next, we describe a representative example of such DT architecture variants, followed by a set of scenarios required to provide DT solutions, that we defined together with our industry partner.

2.2. Motivating Example

Fig. 1 depicts two DT architectures that showcase different use cases of DTs in the context of the SofDCar project. The first use case (cf. DT architecture Variants 1 and 2) concerns temperature control for smart cars, where the driver sets a target temperature. Based on the actual temperature measured in the car, either air conditioning or a seat heater is activated until the chosen target temperature is reached. The second use case (cf. DT architecture Variant 3) is the reconfiguration of an electric car depending on the battery charging level and the remaining distance to the destination. On this basis, energy consumption is adjusted, e.g., by limiting acceleration, setting maximum speed, or turning off air conditioning or the entertainment system.
In the first use case, we extend different DT platforms with a Planner service that computes a sequence of actions for realizing a certain goal (i.e., target situation) for a given initial situation. The planner is used to maintain optimal temperatures as described above. In the second use case, the DT platform functionality is extended with a Deviation Checker service to identify reconfiguration needs, together with a planner to calculate required actions to adjust the energy consumption. All used services provide modeling languages for configuration, i.e., a DT Platform language describes structural aspects of data omitted by physical twins, a Planner language defines the goals and optimization rules of the planner, and a Deviation Checker language describes deviation checking rules.

2.3. Scenarios for building and evolving DT architectures

Scenario 1: Building new DT architecture variants from scratch. In a typical situation, specific requirements of individual cus-

[4] https://sofdcar.de/language/en/

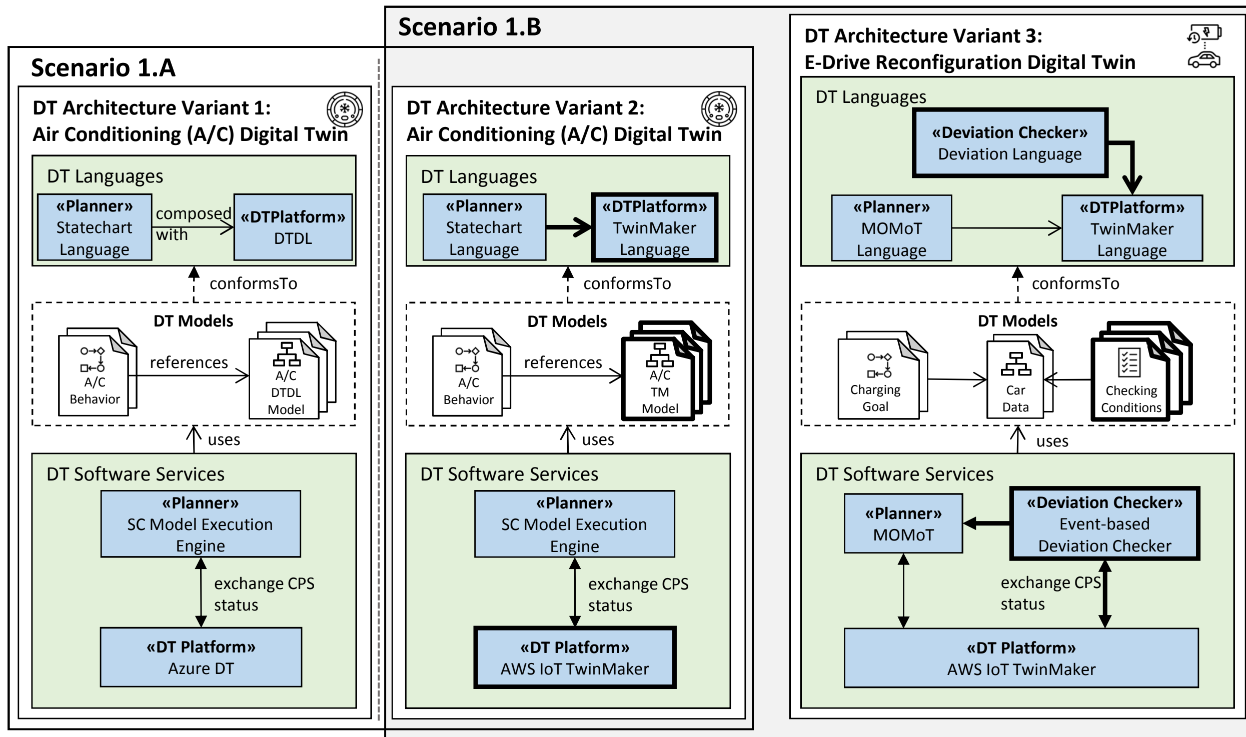

Figure 1 Scenario 1 from our industry partner: Building new DT architecture variants from scratch.

tomers lead to different variants of DT architectures realizing similar functionality (cf. *Scenario 1.A* in Fig. 1), or variants that realize different functionality, but still share certain commonalities (e.g., they all make use of a DT Platform, cf. *Scenario 1.B* in Fig. 1). For example, Variants 1 and 2 in Fig. 1 both realize the air conditioning use case, but Variant 1 makes use of the Azure DT platform and its offered Digital Twin Definition Language (DTDL)[5], whereas Variant 2 uses the AWS IoT TwinMaker service as DT Platform, together with its TwinMaker language. Such variants often emerge because of existing infrastructure (e.g., one customer might already rely on Azure as a cloud provider, whereas another customer already uses AWS cloud infrastructure), or specific restrictions (e.g., if a customer requires the use of on-premise solutions for legal reasons, cloud-based services cannot be employed). In addition, different use cases require individual service combinations. For example, while both use cases presented in Fig. 1 make use of a DT platform and Planner service, the Deviation Checking service is only needed for Variant 3.

As a result, our industry partner is forced to create and manage a wide range of DT architecture variants, based on the specific requirements of different customers. Currently, providing such a set of different DT architectures requires manual integration of each available service that provides a certain functionality with each service that makes use of this functionality. Such manual integration of services is tedious and error-prone, as it involves a lot of redundant work.

[5] https://github.com/Azure/opendigitaltwins-dtdl/blob/master/DTDL/v2/dtdlv2.md

Scenario 2: Evolving existing variants of DT architectures. Over time, most organizations iteratively evolve their employed DT architectures by switching services (*Scenario 2.A*) or adding new functionality(*Scenario 2.B*). For instance, clients might ask to add a new time-based deviation checker to replace the default event-based one or raise the need for a new DT platform (e.g., Eclipse *Ditto*[6]) (cf. Scenario 2.A in Fig. 2). Over time, continuous innovation also leads to the development of new services that customers want to add to their existing DT architectures (*Scenario 2.B*). For instance, the organization wants to leverage multiple DT platforms and, hence, requires a platform integrator that builds a federated DT platform to interact with both DT platforms (cf. Scenario 2.B in Fig. 2).

Adding a new service to existing DT architectures (Scenario 2.A) currently requires to integrate this new service with each DT architecture configuration, while making sure that both the software interfaces match, and the service-specific languages are made compatible. This effort increases with the number of available services. Introducing new service functionality into existing DT architectures (Scenario 2.B) adds another level of complexity to the integration process, as this communication with existing services needs to be defined for all different configuration options.

2.4. Limitations of existing approaches

To integrate services comprising software and related modeling languages, we proposed in previous work the notion of *language plugins* (Dalibor, Heithoff, et al. 2022), which we will refer to

[6] https://www.eclipse.org/ditto/

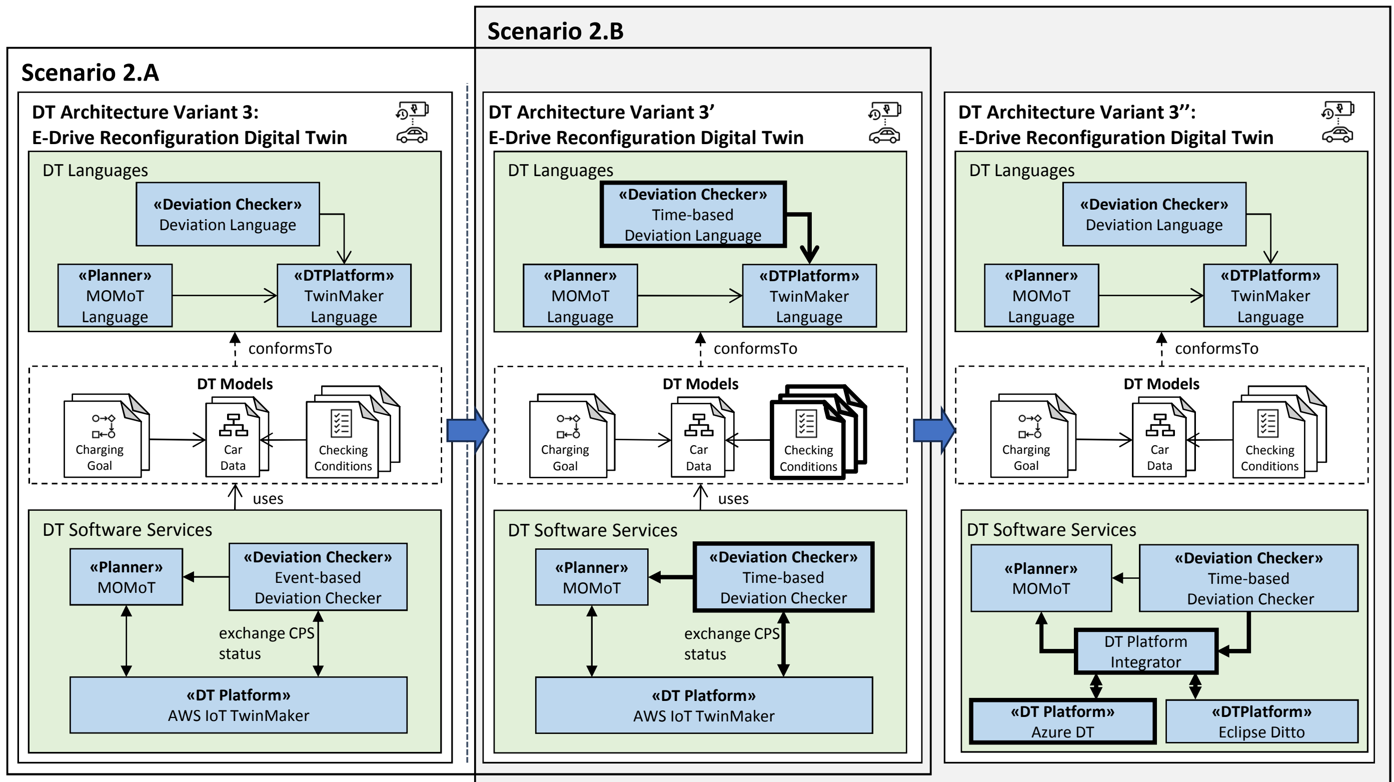

Figure 2 Scenario 2 from our industry partner: Evolving existing variants of DT architectures.

as modules in this paper. These modules can be integrated pairwise using a component and connector (C&C) architecture description language for the software level, and a language component notation to enable language composition based on an explicit language interface. To this end, we also provided a predefined set of high-level software and language interfaces that are integrated into a reference architecture realizing the MAPE-K (Kephart & Chess 2003) pattern. By implementing these interfaces, specific modules could be plugged into this predefined reference architecture. However, when building and evolving a set of DT architecture variants (cf. Scenario 1.A and 2.A), we realized that the pairwise integration of each module is not feasible. From the discussions with our industry partner, we also learned that plugging modules into a predefined reference architecture is not sufficient to support architectures that realize different applications (cf. Scenario 1.B) or add service functionality to existing architectures (cf. Scenario 2.B).

3. A Method for Template-based Architecture Modeling

To overcome the aforementioned limitations, we provide a method to abstract a set of modules into so-called templates. In particular, we use these templates to (i) define arbitrary reference architectures by connecting the software and language ports of available templates, and (ii) generate a product line from such a reference architecture description that allows the selection of available modules for each template.

Fig. 3 gives an overview of the individual phases of this method, which we elaborate on in the remainder of this section. In this overview, grey boxes indicate steps in which we draw on existing work (Dalibor, Heithoff, et al. 2022), whereas white boxes mark the steps that are added based on the template-based nature of the method proposed in this paper. In our explanations, we will focus on these new parts. The following section gives an overview on the individual steps of our method, focusing on these new parts. Section 4 then instantiates these steps in the context of digital twins, using a specific example.

3.1. Phase 1: Service Definition.

In this first phase, we define available services as modules and templates.

In Step 1.1, we define a module for each service that wraps the corresponding software and language components of this service utilizing the `Module Definition Language`. Software components are interfaces that can be used to interact with the respective service. For this interaction, each software component offers incoming ports to send data to the service, and outcoming ports to retrieve the data produced by the service. The functionality provided by a service can be configured using models. The languages that these models conform to are specified as language components. Each language component specifies a set of so-called language ports. We can connect different languages by connecting these ports using language composition (Butting et al. 2020). This definition of modules is compliant with what is proposed in (Dalibor, Heithoff, et al. 2022).

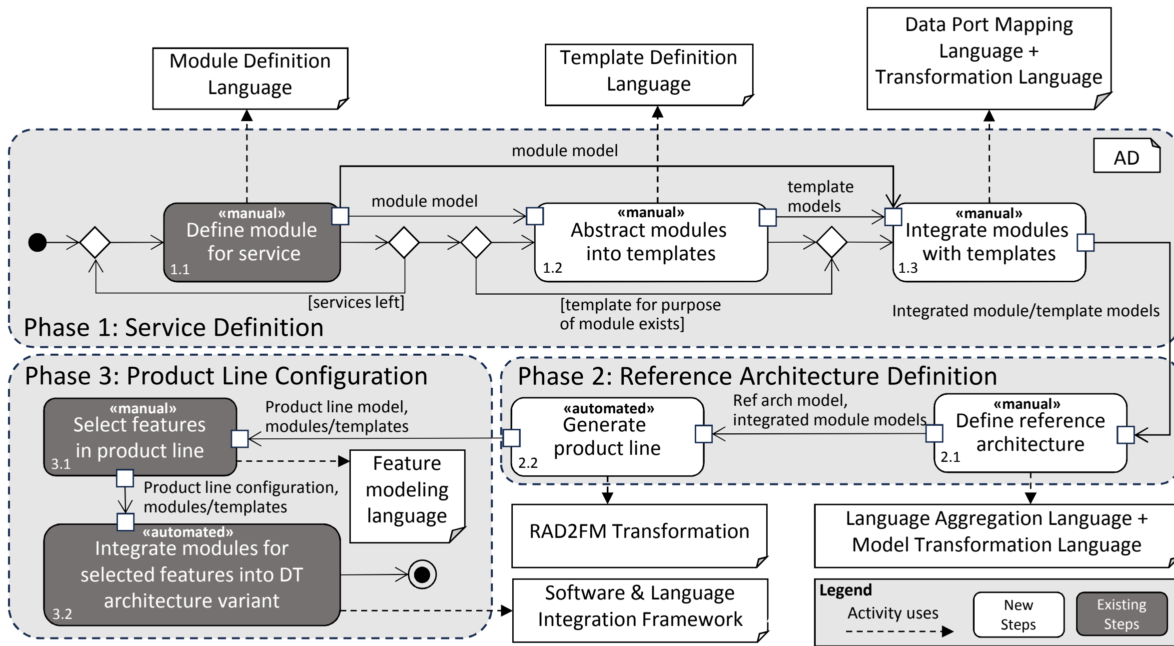

Figure 3 An overview of template-based digital twin architectures method. Phase 1: module and template definition and their integration; Phase 2: definition of reference architectures and generation of product lines. Phase 3: selection of DT variant from product line and composing software and language components.

In Step 1.2, we create templates as abstractions of the previously defined modules. These templates provide contracts for modules by specifying software and language ports that must be implemented by all modules that realize a certain template. These ports are specified together with a name for the template using a `Template Definition Language`
After Step 1.1, the modules provide specific ports based on the respective service. For a module to implement a certain template, these ports have to be wrapped to realize the contracts imposed by this template. Therefore, we define a mapping from the module specific port names and port types to the abstracted ports imposed by the templates in a `Data Port Mapping Language`. A similar mapping is also performed between the language components of the module and the template. For this mapping, we leverage the existing concept of language inheritance, and specify the required mapping rules using a `Transformation Language`.

3.2. Phase 2: Reference Architecture Definition (RAD)

Step 2.1 is concerned with the definition of so-called *reference architectures*. For this, a set of templates is connected to realize a certain reference architecture. This connection must be performed for all (software and language) components of the selected templates. For the software components, this means that outgoing software ports of a template must be connected to the incoming software port of another template. This connection also involves the definition of glue code to map the data type provided by the outgoing port to the data type required by the incoming port. For language components, this means that we connect each provided language port of selected templates to the required ports of another template. This connection affects that both languages are composed using language aggregation, i.e., a link between both language's ports is created, e.g., to check consistency between models conform to these languages. As a result of this Reference Architecture Definition (RAD) of templates, we obtain a `reference architecture model`, that is used in Step 2.2 to generate an architecture product line. The first level of this product line contains the list of templates that we already connected in the respective RAD model, followed by the modules that we connected to the respective templates in Step 1.3.

3.3. Phase 3: Product Line Configuration

The mentioned architecture product line, generated from the selected DT reference architecture model, allows the connection of available modules into different architecture variants. To select a specific variant from the product line in Step 3.1, we have to define a product line configuration model by selecting one available module for each template available in the product line.
Based on this product line configuration, in Step 3.2, we can integrate the selected modules into a finished architecture based on (i) the glue code between the respective templates defined in Step 2.1 in the RAD, and (ii) the inheritance relation between the modules and their template specified in Step 1.3. Such an integration is already shown in (Dalibor, Heithoff, et al. 2022) and this allows us to reuse existing tooling to achieve the following two integration solutions. The result of the integration of the software components is the operable DT architecture. For the language components, their integration results in a modeling editor that allows the instantiation of the software architecture for specific use cases.

The DT architecture can be instantiated by domain experts who can create models of the service-specific languages contained in the modules selected in Step 3.1. As these modules inherit from templates, and the templates are connected through language aggregation in the RAD, the created models are automatically integrated into one common model that contains the combined information of all created models. This model is then used to configure the respective services to a particular use case.

3.4. Tool Support

We provide a prototypical implementation of the presented method online[7]. This implementation builds upon the framework introduced in (Dalibor, Heithoff, et al. 2022), which integrates software components defined in the MontiArc ADL (Butting et al. 2017) and SCOLaR (Butting et al. 2020; Pfeiffer & Wortmann 2021, 2023) black-box language components into an operable software architecture and the respective modeling editor (i.e., Phase 3 of our method). The composition of two language components consists of two main activities: (1) Composing the components' interfaces; and (2) Composition of the comprised language definition constituents (grammar, well-formedness rules, code generators). In our approach, we leverage the two language composition operators language inheritance and language aggregation. Language inheritance extends an existing language, i.e., the generic template languages, with new concepts, i.e., the service-specific module language. Language aggregation takes two language definitions as input and creates a link between them. In contrast to inheritance, however, the models and the language definitions remain separate and are only connected by glue code on the language definition level. This is especially useful in our approach where we aim to allow independently developed languages to interact. For implementing our method, we provide dedicated languages to define modules and templates in Phase 1, as well as RADs, and product line configurations in Phase 2, using the MontiCore (Krahn et al. 2010) language workbench. We also implement a model-to-model transformation that translates RAD models defined in Phase 2 into the product lines used in Phase 3. Since both the RAD language and the language for product lines are defined using MontiCore, the transformation is implemented by transforming the abstract syntax tree (AST) of a parsed RAD model into an AST of the product line language according to the mapping defined in Section 3.2. The final AST is then pretty-printed into a product line model.

4. Method Instantiation in the Context of Digital Twins

The reference architecture modeling method we introduced above is generally applicable to reference architectures from different application domains. Thus, as a prerequisite to applying this method to the scenarios introduced in Section 2, we have to instantiate it in the context of DTs. In the following, we answer the question of *"Which steps are required to model DT architectures using our template-based approach?"* by instantiating the proposed method for connecting different planning tools (i.e.,

[7] https://github.com/cdl-mint/DT-Product-Line-Architecture

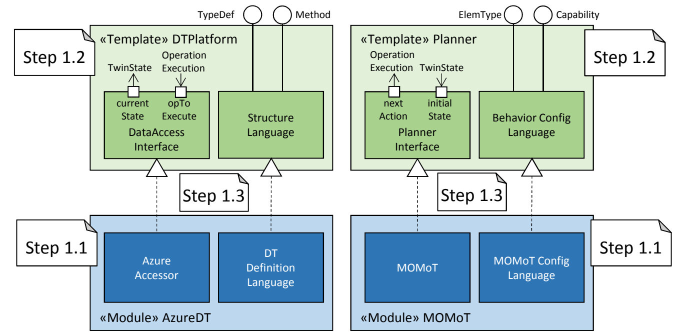

Figure 4 Integration of the AzureDT module with the DTPlatform template (Result after Phase 1).

the model-driven optimization tool MOMoT (Bill et al. 2019), and planners based on executing basic or timed statecharts) with commercial DT platforms offered by AWS and Azure, to create a reference architecture for DT-based reactive planning, as e.g. discussed in (Eisenberg et al. 2022).

The different planning services yield a plan that has to be executed on a system in order to achieve a certain goal (e.g., optimize overall energy consumption). Based on a model representing a system state, a planner returns the next action that needs to be performed in this system state, based on the stored plan. In the MOMoT planner, the plan is calculated by executing a set of predefined graph transformations on the input model using genetic algorithms. The list of graph transformations that achieve the planning target most efficiently is taken as the to-be-executed actions for the system. The used transformations must be specified using the Henshin graph transformation tool (Arendt et al. 2010), and the system state must be a model conforming to a meta-model specified in Ecore (Steinberg et al. 2008). In the statechart-based planners, the plan is stored as a basic or timed statechart, respectively. The next action in the plan is calculated based on the currently activated state and the input model. For instance, in a temperature control statechart, if the system is in *idle* state, but the actual temperature in the room is higher than the threshold value, the state is switched to *cooling*, and an action for activating the cooler is triggered.

To perform such planning based on live data rather than a persisted model, we need to connect these planners to DT platforms that host the running devices to derive the input model used for planning from live data and execute the action returned by the planner on the running devices. In this section, we use the Azure Digital Twins and AWS TwinMaker service as examples of such DT platforms. Both of these DT platforms provide a REST interface for interacting with running devices.

4.1. Phase 1: Service Definition

Part of the output of this phase is depicted in Fig. 4, namely an AzureDT module that implements the DTPlatform template as well as a MOMoT module implementing the planner template.

Step 1.1: Define modules In this first step, we define the modules for all available services. The goal of this step is to

bridge the specific technologies of the different tools (e.g., the MOMoT tool is stored as an executable Java Archive (JAR) file, whereas the DT platforms provide an HTTP interface) using Java code as an integration platform. Therefore, we provide a textual description of the in and out-ports of each module and use the MontiArc framework to generate a Java class based on this description. The resulting class contains an attribute for each port, a setter method for each incoming and a getter method for each outgoing port. On top of these port definitions, MontiArc also generates a `compute` method that needs to be implemented to bridge the specific technology of the respective tool and Java.

For example, in the module wrapping the MOMoT tool, the `inputModel` is stored as `EObject`, i.e., the Java class for interacting with models that conform to an Ecore meta-model together with a String representing the name of the respective meta-model, and the calculated `plan` is stored as a list of `UnitApplication` Java objects which contain execution information of a Henshin graph transformation. In the `compute` method of this class, these Java attributes are bridged with the jar-file containing the respective MOMoT configuration by executing this jar-file with the inputModel as the parameter, and storing the resulting plan in the `plan` attribute. The correct jar-file is located based on the meta-model name. The inputModel is stored as an xmi-file, and the path to this xmi-file is passed as the parameter when executing this jar-file.

For the AzureDT module wrapping the Azure platform and the AWS TwinMaker (AWSTM) module wrapping the AWS DT platform, the `compute` method of the respective Java class generated by MontiArc implements the interaction with the Azure digital twins or AWS TwinMaker service. To enable this interaction, we have to set the chosen service's web address. Based on this web address, we call the correct service methods and set the parameters accordingly. For instance, for updating the `currentState` of a DT in the AzureDT module, we call the `getById` method of the Azure DT webservice[8], passing the `id` of the twin we want to update as a parameter. The value of the `currentState` attribute is updated based on the Twin object received as a response. We perform this update of the `currentState` on every call of the compute method and also execute an operation on the system according to the value of the `command` attribute.

Step 1.2: Abstract modules into templates In this second step, we use the defined modules and abstract common functionality into the DT platform and planner templates. For example, for the DT platform template, this means that we generalize the specific languages used by different platforms into a common `Structure Language`. A simplified version of this language is shown in Fig. 4. It provides a `TypeDef` port that specifies type information on available twins, and a `Method` port defining operations that can be executed on twins. The provided software ports allow access to runtime data through the `currentState` port, and the execution of operations through the `opToExecute` port. We also need to define generic data types that are used

[8] https://learn.microsoft.com/en-us/rest/api/digital-twins/dataplane/twins/digital-twins-get-by-id

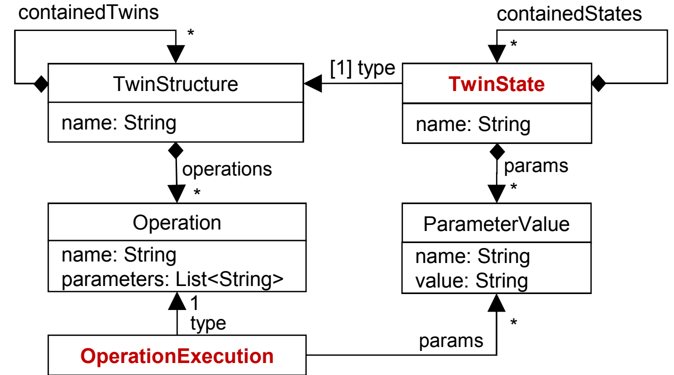

Figure 5 Structure of the `TwinState` and `OperationExecution` data types of the software ports used by the DT platform and planner templates (in class diagram syntax).

by these software ports. Fig. 5 gives an overview of the data types we define for both DT platform and Planner template (`TwinState` is used by the currentState and initialState ports, and `OperationExecution` is used by the opToExecute and nextAction ports).

Step 1.3: Integrate modules with templates The modules specified in Step 1.1 can now implement the generic ports of the respective templates, as shown in Fig. 4. Therefore, they have to provide the same ports as the software and language components of the template. However, most software and languages use specific software and language ports. For example, Azure uses the Twin datatype for the currentState port, instead of the TwinState type imposed by the DT platform template. Thus, we have to write so-called wrappers that map the specific ports of the module's software and language components to those imposed by the respective template. Fig. 6 gives an excerpt of the rules that have to be implemented in such a wrapper.

```
1  <-- Azure2DTPlatform Transformation
2  rule TwinToState{
3    from twin Azure!Twin
4    to state DTPlatform.TwinState(
5       name -> twin.name
6       type -> twin.type
7       params -> twin.properties
8       params -> twin.telemetries
9    )
10 }
11
12 ...
13
14 <-- DTPlatform2Azure Transformation
15 rule OpExecToCommand{
16   from op DTPlatform!OperationExecution
17   to cmd Azure.Command(
18      type.name -> cmd.name
19      params -> cmd.commandParameters
20   )
21 }
```

Figure 6 Excerpt of the transformation rules required by the AzureDT module generated from MontiArc for the integration with the DT platform template.

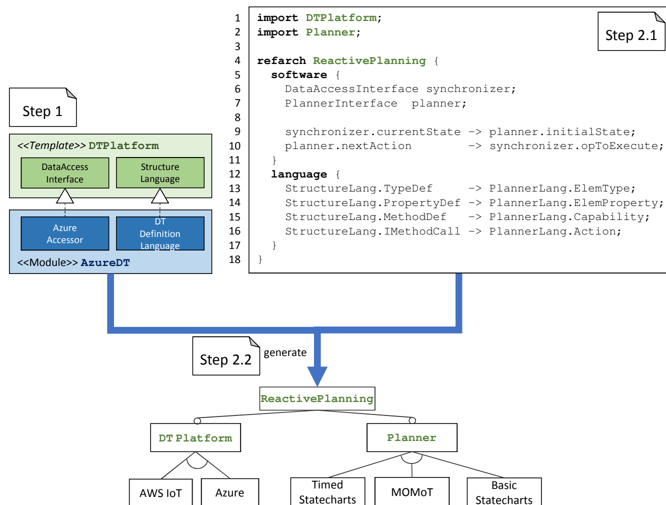

Figure 7 Definition of the DT RAD model using templates defined in Phase 1 (top), and visualization of the resulting DT product line with the chosen templates (green) and available modules (blue).

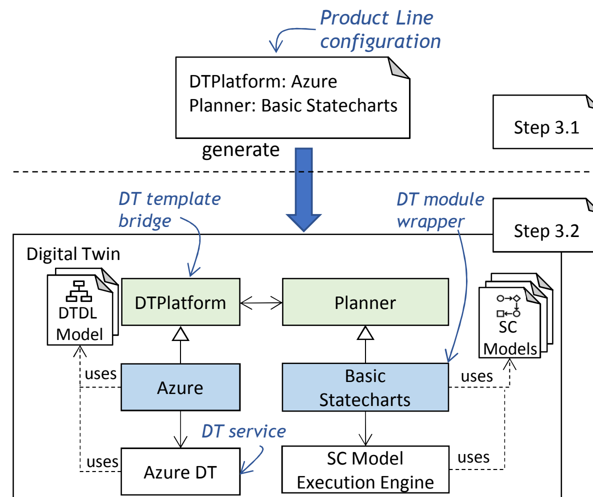

Figure 8 Configuration of product line generated in Phase 2 (top) and the DT architecture (bottom) resulting from the generation performed in Phase 3.

4.2. Phase 2: RAD

In Phase 2, we use the DT platform and planner templates defined in Phase 1 to generate a product line for our DT-based reactive planning reference architecture.

Step 2.1: Define Reference Architecture The upper part of Fig. 7 shows how we define the respective reference architecture in Step 2.1. This RAD model starts with the import of the templates to be used (lines 1-2). Then the keyword `refarch` followed by a name defines the architecture (line 4). Afterward, the mappings between the template's software components (lines 6-7) are defined, thereby, connecting all ingoing and outgoing ports (lines 9-10). For defining the composition of the template's language components, their provided and required interfaces need to be bound together defining which concepts should be linked using language aggregation (lines 13-16).

Step 2.2: Generate Product Line In Step 2.2, we use this RAD model to generate the product line depicted at the bottom of Fig. 7. In this product line, the root feature is derived from the name of the RAD, i.e., `ReactivePlanning`. The level 1 features are the names of the imported templates, i.e., `DTPlatform` and `Planner`. The leaves of the tree are all referring to modules that implement the respective parent feature's template. Hence, `AWS IoT TwinMaker` and `Azure DT` are children of the feature `DTPlatform`.

4.3. Phase 3: Product Line Configuration

By selecting one module for each available template, we configure our product line in Step 3.1, as shown in the upper part of Fig. 8. In the lower part of this figure, we can see the DT architecture generated from this product line configuration in Step 3.2. The integration of software components in this step is rather straightforward. The modules' software components can be replaced in the DT RAD model, yielding a new MontiArc architecture model that comprises the modules' MontiArc components and their connections. MontiArc can then generate executable Java code, connecting the selected software components. More precisely, *DT template bridges* are generated that contain the glue code that bridges the used templates based on the chosen DT RAD model. This glue code is inherited by *DT module wrappers* that perform the communication between the software components for the modules selected in the DT architecture configuration model. These DT module wrappers also wrap the actual software component of the selected module by mapping each specific port of this software component to the abstracted port imposed by the respective template. Thus, the specific software component of the module still runs decoupled from the Java code generated by our method.

For integrating available language components, we use two different language composition techniques, i.e., language aggregation between languages of templates (Pfeiffer & Wortmann 2021), and language inheritance between the languages of templates and modules. The former is performed based on the DT RAD model. When using language aggregation the composed languages remain independent of each other after the composition. Considering the running example of this section, the state machine language stays independent of the aggregated structure description language of the DTPlatform. Thus, both languages can be extended by module-specific languages without requiring adaptation of the aggregating language or vice versa. The languages of modules and templates are composed using language inheritance. Thereby, similar to object-orientation, the language of the module extends the language, in particular the contained language concepts, of the template. Consequently, the languages can be utilized by domain experts using the module-specific syntax but can be reused via the generic language of the templates that they extend at the same time.

4.4. Result: Integrated DT architecture

After receiving the integrated software architecture and the respective language to configure the architecture components as result of Phase 3, we can define models in the respective language to tailor our architecture to a particular use case. For the DT architecture in Fig. 8, we create models in the DT Definition Language used by the AzureDT module and statechart language used by the Basic Statechart module. As these two languages are integrated, the statechart models can reference types, attributes, and methods of the DTDL model. For this, we rely on previous work (Dalibor, Heithoff, et al. 2022).

5. Evaluation

We now evaluate the method proposed in this paper for DTs by using the scenarios introduced in Section 2. More precisely, we aim to answer the following research question.

Research Question (RQ): *What are the pros and cons of using the presented approach compared to the state-of-the-art?* To answer this RQ, we perform a comparative study of realizing the scenarios from our industry partner (see Section 2) with three different approaches. The `template-based approach` employs the method proposed in this paper. In the `module-only approach`, we manually integrate available modules directly with each other by using the approach presented in (Dalibor, Heithoff, et al. 2022). For the `MAPE-only approach`, we integrate modules into a given MAPE RAD, as done in (Dalibor, Heithoff, et al. 2022; Pfeiffer et al. 2023). Thereby, we investigate the impact that the different approaches on the overall effort for realizing the aforementioned scenarios (cf. Section 5.2 for Scenario 1 and Section 5.3 for Scenario 2). We finish this section with a discussion of the limitations of our evaluation and the proposed method of this paper (Section 5.5).

All necessary artifacts to create the described setup and execute the different scenarios are available in our online repository[7].

5.1. Setup

We now describe the setup of our study, namely the used templates and modules, the considered RADs, and how we apply the different methods for the scenarios of our industry partner.

5.1.1. Used templates/modules In this evaluation, we make use of the following templates and modules:

DTPlatform enables communication with industrial DT platforms. We specify modules for Microsoft *Azure DT* and AWS *TwinMaker*.

Planner provides planning functionality, implemented in the already described *basic* and *time-based* SC planner modules, next to the *MOMoT planner*.

DeviationChecker allows us to observe the runtime data (e.g., from a DTPlatform) and compares this data to simulation results retrieved from a simulation component which is also part of the DeviationChecker template. In the DT architecture variant 3 from Fig. 1, we use an *event-based deviation checker* that uses the full state of the running system and compares this state with the results of an event-based simulation. As an alternative, we introduce the *time-based deviation checker* module which uses time-based simulation as a basis for comparison.

Integrator builds a federated DTPlatform interface to interact with DTPlatforms that run in different instantiations of the DTPlatform template. It provides (i) the same software and language ports as the DTPlatform template to connect the Integrator to other templates based on the DTPlatform interface, and (ii) additional software and language ports that are connected to the DTPlatform templates that should be federated.

Fig. 9 gives three examples of the aforementioned templates and one module realization for each of them.

5.1.2. Considered RADs Using the aforementioned templates, we can build a variety of RAD models, including all of the RADs used by the DT architecture variants presented in Fig. 1 and Fig. 2. In particular, we use the following RADs: RAD 1 combines the DTPlatform with Deviation Checker and planner templates to realize a self-adaptation RAD based on the MAPE pattern (Kephart & Chess 2003). RAD 2 builds an alternative RAD that directly connects the DTPlatform with the planner template, without using the Deviation Checker. With RAD 3, we can create two instantiations of the DTPlatform template and federate them using the Integrator. By connecting the Integrator to the planner, we can realize a self-adaptation RAD for devices running in different DTPlatforms.

5.1.3. Evaluation Settings In the following, we implement different DT architectures using the three different approaches and compare their resulting implementations. For the `template-based approach`, we use the setup as introduced in Section 4. Next, we detail the setup of the `module-only approach` and `MAPE-only approach`.

Module-only approach. To describe the setup of the `module-only approach`, we use the example of connecting planners with DT platforms from the previous section but focus on the connection of the software components of the MOMoT planner with the Azure and AWS DT platforms. To perform this connection, we have to (1) manually write the glue code that transforms (1.1) the live data retrieved from the DT platforms to the input model required by MOMoT, and (1.2) transform the `UnitApplication` object calculated by MOMoT to the data format required by the DT platforms to execute this `UnitApplication` on a running device. Based on this mapping, we (2) connect the respective modules to finalize the connection of MOMoT to the DT platforms. The remaining steps to generate the respective DT architectures in this setting (i.e., module definition, product line configuration, and generation of a DT architecture based on a product line configuration) are similar to the respective steps in the `template-based approach`.

(1.1) Glue code for transforming live data into an EObject. In order to connect the live data retrieved from the DT platform (an object of type `Interface` for Azure and `Component` for AWS) with the input model of type EObject required by MOMoT, we need to write for each DT platform a dedicated `setter` method of the `inputModel` attribute in the respective class of the MOMoT module. For the Azure DT platform, this method requires a parameter of type `Interface`, and transforms this parameter into an `EObject` by implementing the transformation rules described in Fig. 10. The method for the AWS DT platform

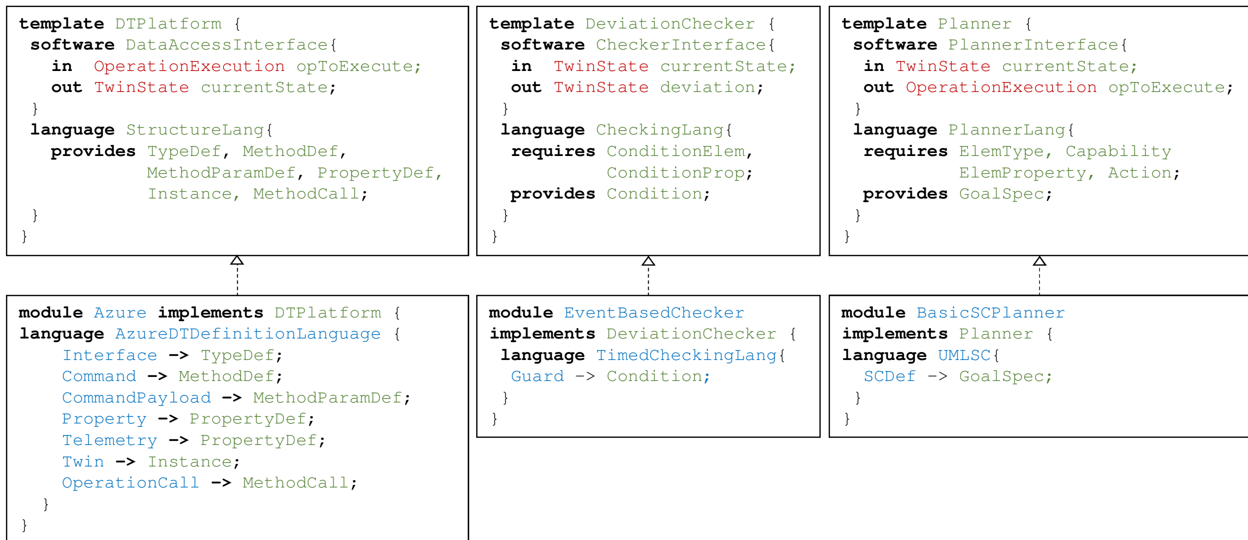

Figure 9 Excerpt of module and template definition for the evaluation setup.

requires a parameter of type `Component` instead, implementing the respective rules to transform this `Component` object into an element of type `EObject`.

(1.2) Glue code for executing UnitApplication on running devices. For mapping the `UnitApplication` object retrieved as `nextStep` from MOMoT to the respective element in the DT platform, we have to implement a dedicated getter method of this `nextStep` attribute for each DT platform. Each method returns the value of the `nextStep` attribute in the specific format required by the respective DT platform. Therefore, the method implements the rules required to transform the `UnitApplication` object of the `MOMoT` tool to a `Command` object for Azure (cf. Fig. 10) or a `Function` object for AWS.

(2) Connecting the modules. We can now integrate MOMoT with the Azure DT platform by regularly retrieving the `currentState` from the AzureDT module and passing it to the Azure setter method of the `inputModel` of the MOMoT module, and retrieving the `nextStep` of the MOMoT module via the getter method that returns the `Command` element that can be passed to the AzureDT module.

MAPE-only approach To perform a similar integration than described in the `MAPE-only approach`, we have to connect the MOMoT, AzureDT, and AWS modules with their respective components of the predefined MAPE RAD, similar to Step 1.3 of the method outline in Fig. 3. To perform this integration, we have to implement the same transformation rules as in the `template-based approach`, as outlined in Fig. 6. In contrast to the `template-based approach`, this integration only allows us to generate DT architectures that use the MAPE RAD, i.e., not all possible DT architecture variants are supported.

Next, we investigate how the differences in these individual approaches manifest in the context of the scenarios of Section 2.

5.2. Application to Scenario 1: Building DT architecture variants from scratch

In the following, we discuss how to model different reference architectures and their configurations to realize Scenario 1 using the three different approaches (i.e.,, `template-based approach`, `module-only approach`, and `MAPE-only approach`). Therefore, we assume that we already defined the available modules for the `template-based approach`, `module-only approach` and `MAPE-only approach`, and templates for the `template-based approach`, as described above.

5.2.1. Create a single DT architecture Using the `template-based approach`, the creation of a single DT architecture requires following the three phases described in Section 3. For example, to create a system that follows the MAPE RAD, we could use the DTPlatform, deviation checker, and planner template from above and connect them accordingly. In the product line, we select the Azure DT module as DTPlatform, the time-based variant for the Deviation Checker, and the basic SC module for the planner template. This allows us to generate the DT architecture, as described in Phase 3 of Fig. 3.

In contrast, as the `MAPE-only approach` imposes the MAPE RAD, it is only necessary to define and integrate the modules. In the `module-only approach`, the glue code to integrate the three required modules—Azure, time-based deviation checker, and basic SC planner—needs to be developed by hand. This integration requires connecting all incoming and outgoing ports of the individual modules manually.

> **Lessons Learned** Our approach requires definition of both templates and modules, before a DT architecture can be created. In comparison, the `module-only approach` can directly integrate the modules via the manual development of glue code. In the `MAPE-only approach`, the templates are already provided by the imposed MAPE RAD. We see that for creating a single DT architecture our method requires extra effort.

5.2.2. Scenario 1.A: Creating several DT architecture variants for one RAD Our partner might need to create

```
1  rule currentStateToInputModel{
2    from twin Azure!Twin
3    to eobj MOMoT.EObject(
4       name -> twin.name
5       type -> twin.type
6       eAttributes -> twin.telemetries
7       eAttributes -> twin.properties
8       ...
9    )
10 }
11
12 ...
13
14 rule UnitApplicationToCommand{
15   from ua MOMoT!UnitApplication
16   to cmd Azure!Command(
17      name -> ua.name
18      parameterValues
19      -> ua.executionParameters
20   )
21 }
22
23 rule ParamValueToCommandParam{
24   from param MOMoT!ExecutionParameter
25   to cmdParam Azure.CommandParameter(
26      name -> param.name
27      value -> param.parameterValue
28   )
29 }
```

Figure 10 Excerpt of transformation rules that need to be implemented by the MOMoT module generated from MontiArc for its integration with the Azure DT platform.

different variants of MAPE (i.e., particular DT architectures realizing the MAPE-based RAD) based on the specifications of different clients, as described in Scenario 1.A.

In our `template-based approach` this versatility is built in. This means that once a RAD is defined, it is possible to generate all combinations of modules for each client's DT architecture based on this RAD. For instance, for our MAPE-based RAD, we can generate 12 different DT architectures using the modules described in Section 5.1.

In the `MAPE-only approach`, we may reuse the same modules defined above. However, as MAPE is imposed by the method, it is not necessary to define a RAD.

For realizing the same example in the `module-only approach`, in the absence of templates, modules would have to be integrated manually for each of the 12 available DT architectures.

> **Lessons Learned** When generating several DT architecture variants for the same RAD, using templates and DT RAD models is more scalable than the manual integration of modules for each DT architecture. As the `MAPE-only approach` is optimized for integrating DT architectures for the MAPE RAD, it is also more efficient in building MAPE-based DT architectures.

5.2.3. Scenario 1.B: Creating DT architectures using different RADs

To realize Scenario 1.B in our `template-based approach`, we have to create a DT RAD model for each RAD according to the DT architectures to be built. Based on these DT RAD models, we can create a DT architecture configuration model for each required DT architecture. For example, in addition to the MAPE RAD (Scenario 1.A), we can generate six further DT architectures where we directly connect the three different planner modules with the two DTPlatform modules. Twelve further variants can be created by connecting the three available planners with the available integrators in combination with all possible combinations of instantiating two DTPlatforms. This results in 30 total DT architectures that can be built from these three RADs.

Evidently, in the `module-only approach` the effort is much higher. To realize this example in the `module-only approach`, we would need to manually integrate module combinations in the 30 different DT architectures, regardless of the reference architecture used. This means writing glue code to connect the two DTPlatform modules with the different Integrator modules, and these Integrator modules with the three different planners, and these planners again with the DTPlatform modules, just to realize the DT architectures for one of the three RADs. To create the DT architectures of the other RADs, we need to connect all DTPlatforms with all Deviation Checker and planner modules, and the Deviation Checker with the planner modules.

The `MAPE-only approach` is only applicable to the twelve MAPE-based RADs. DT architectures for other RADs cannot be built using this approach.

> **Lessons Learned** The high variability of the integration of different RADs and DT architectures clearly shows the benefits of increased scalability of the `template-based approach` in comparison to the `module-only approach`. The `MAPE-only approach` is applicable for twelve out of the overall 30 DT architectures in this scenario, which shows that several DT architectures cannot be supported.

5.3. Application to Scenario 2: Evolving existing variants of DT architectures

In the following, we demonstrate the realization of Scenario 2 using the three different approaches (i.e.,`template-based approach`, `module-only approach`, and `MAPE-only approach`). More precisely, we showcase the addition of a new module and a new template into the DT architectures created for Scenario 1.B.

5.3.1. Scenario 2.A: Switch to new module in existing DT architectures

To add a new module using the `template-based approach`, we first define the service following the steps in Phase 1 (Section 3), which enables its selection in the product line. Then, to integrate it in existing DT architectures, we simply alter the respective DT architecture configuration models and re-generate the DT architectures. For example, the emergence of a new DT platform such as Eclipse Ditto can require us to replace the Azure DT module with a new Eclipse Ditto module. Therefore, we just replace the respective modules in all DT architecture variants created for Scenario 1.B. The compatibility with existing modules is already ensured by the respective RAD.

In the `MAPE-only approach`, we also replace the Azure DT with the new Eclipse Ditto module, but only for the subset of

12 out of 30 DT architectures that we were able to create using this approach in Scenario 1.B.

To perform this Scenario in the `module-only approach`, we would need to manually exchange the Azure *DT* module with Eclipse *Ditto* in all 15 DT architectures from Scenario 1.B that use the Azure *DT* module, asserting the compatibility of *Ditto* with all other modules in each DT architecture.

> **Lessons Learned** To include a new tool to replace an existing one, the `template-based approach` only requires the definition of the new module and regeneration of the required DT architectures. In the `MAPE-only approach`, we also perform this replacement by defining the new module, but as mentioned before, less than half of all DT architectures are supported. In the `module-only approach`, instead, we have to manually integrate the new module with all existing modules of each DT architecture.

5.3.2. Scenario 2.B: Add new functionality to existing RADs To extend the functionality of existing RADs, in the `template-based approach`, we need to first define a template that describes this new functionality via its interface, and integrate it into the existing RADs. This change in the DT RAD model implies that all existing DT architecture configuration models that use this RAD need to be extended accordingly. For example, we could extend the MAPE RAD by connecting the deviation checker and planner with the integrator template. Then, for each of the existing MAPE DT architecture configuration models from Scenario 1.B, we have to select two specific modules for the DTPlatform and one for the Integrator templates. After these adaptations, we can regenerate the DT architectures to include the newly added functionality.

The `MAPE-only approach` does not support this scenario, as the imposed MAPE RAD cannot be altered in this approach.

Realizing this example evolution in the `module-only approach` would mean to manually "glue" the DTPlatform to the Integrator module, and the Integrator to the existing modules for each MAPE-based DT architecture from Scenario 1.B.

> **Lessons Learned** When adding new functionality to existing RAD in the `template-based approach`, only the connections between the added template and existing templates of the considered DT RAD model need to be defined. In the `module-only approach`, the new functionality needs to be integrated with the existing modules of each DT architecture that uses the considered RAD. The `MAPE-only approach` does not allow altering the RAD at all.

5.4. Discussion

In the following, we discuss the answer to the research question posed at the beginning of this section, based on the evaluation results presented above. By implementing the scenarios from Section 2 using the method proposed in this paper, and comparing the resulting realization with two baseline approaches, we found that the definition of modules and templates evidently comes with a price. Especially for "one-off" developments that aim at the creation of a single DT architecture, this effort might not be worthwhile. Nonetheless, our template-based architecture modeling method can be used to flexibly generate DT architecture variants based on a wide range of RADs. Especially when several similar DT architectures need to be built, the amount of glue code required remains manageable, compared to the exponentially growing amount of glue code required for manual integration of modules.

We also see that the added effort of creating templates pays off as soon as the architectures have to be extended to include additional functionality. Our method also provides complete flexibility to compose templates into any RAD. This evolution of RADs, for instance, is clearly not foreseen in previous approaches (Pfeiffer et al. 2023) or leads to significant overhead when performed on the individual module level, as done in (Dalibor, Heithoff, et al. 2022).

5.5. Limitations

One major limitation of the evaluation presented in this paper is that it is based on scenarios from one industry partner. However, we provide an open-source implementation of the proposed method[7] that can be used by peers to apply the presented approach to their own case studies.

We also note that the technologies used to create the prototypical implementation of our method used for evaluation, i.e., MontiArc and SCOLaR, impose certain limitations. MontiArc generates Java code for integrating software components, which implies that these components need to either be written in Java or provide a compatible interface. The SCOLaR framework is developed in the MontiCore workbench, thus the languages used for the language components in our method also require a compatible definition in MontiCore.

6. Related Work

In the following, we discuss related work concerning the presented contributions of this paper.

Model-based Tool Integration. In this paper, we aim to integrate different tools into reference architectures to realize enhanced DTs. Several approaches have been defined in the past that tackle the challenge of integrating different modeling tools (Tratt 2005). For example, the Model BUS framework (Blanc et al. 2005) proposes a dedicated model exchange type describing the interface of a modeling service to ensure compatibility between inputs and outputs of different services and provide entry points for invocation of modeling services. The ModelCVS project (Kappel et al. 2006) also provided techniques to integrate different tools by realizing model exchange by model transformations. Another approach uses model transformations between file formats to realize information exchange between tools (Platenius-Mohr et al. 2020).

Whereas all of these approaches focus on tool integration during design time, i.e., one-time data exchange of design models, to solve the challenges presented in this paper, we need to orchestrate tools at runtime to have continuous data exchange between the different tools. This orchestration is achieved with the presented method by providing explicit modeling support for the architecture that a system uses during its runtime.

Tool Interoperability through Standardization. An alternative to tool integration as described in the previous paragraph is the standardization of communication interfaces and protocols between different tools. This is, e.g., successfully implemented for simulation tools with the Functional Mock-up Interface (Blockwitz et al. 2012). Another example is the standardized communication between industrial assets using standards such as OPC-UA[9] or MTConnect[10]. The Asset Administration Shell (AAS) (Arm et al. 2021; Tantik & Anderl 2017) or digital twin definition language by Microsoft[11] are also attempting to achieve similar standardization in the context of DTs. However, currently, available standards for managing the complexity of DTs are insufficient for most DT applications, as they only provide a structural representation of physical assets (Pfeiffer et al. 2022), and do not allow for natively extending the available modeling languages, as these standards lack an explicit metamodeling level (Atkinson & Kühne 2021). Previous work shows how different modeling languages can be summarized into a homogeneous exchange format on the data level with UML profiles for class diagrams (Pfeiffer et al. 2022), ontologies (Vila et al. 2021) or conceptual models (Becker et al. 2021). In addition, there are approaches towards homogenization of digital (twin) architectures, e.g., ontologies for digital platforms (Derave et al. 2020) aim at summarizing the functionalities and requirements of three different types of digital platforms.

The existing related work aims to integrate different DT services by standardizing the communication between software services, e.g., OPC-UA, MTConnect, or AAS, and the integration of DT languages by providing UML profiles or ontologies. However, none of these approaches is sufficient to solve the integration challenges on both the software and language levels. In our work, we achieve this integration by encapsulating both software and languages into modules. Different modules are abstracted into templates to overcome heterogeneity through common interfaces. For the integration between software services, we leverage a component & connector ADL and model transformations to convert incompatible types between their interfaces. For language integration, we employ the two language composition operators: inheritance and aggregation.

7. Summary and Future Work

This paper presents a method to facilitate the specification, realization, and evolution of DT architectures by lifting the integration of different components to the reference architecture modeling level. This is achieved by abstracting DT services into templates, which provide a common interface for a set of services. This reduces the manual effort for integrating new DT services. As templates are abstractions of particular services, the integration space is drastically reduced compared to integrating services directly. By providing a transformation from reference architectures to product lines already equipped with code generation facilities, we can derive the necessary glue code for integrating heterogeneous services into concrete architectures. Our evaluation showed that, although our method implies some overhead through the definition of reference architectures and templates, it provides high scalability for integrating DT architectures that use a variety of functionality.

We currently work towards further applications of our prototype with our industrial partner, where we plan to study how to implement a set of templates and modules in a more diverse set of real-world industry settings. In the future, we aim to further investigate the generalizability of these scenarios for other industrial partners, thereby extending our scope to domains other than the automotive area that we investigated in the context of the presented SofDCar project. Studying generalizability also includes the consideration of different reference architectures and services to implement these reference architectures that are commonly used by organizations in these domains.

Acknowledgments

This work has been supported by the German Federal Ministry of Economic Affairs and Climate Action (BMWK, Bundesministerium für Wirtschaft und Klimaschutz) under grant no. 19S21002L and by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development (CDG).

References

Arendt, T., Biermann, E., Jurack, S., Krause, C., & Taentzer, G. (2010). Henshin: Advanced concepts and tools for in-place EMF model transformations. In *International Conference on Model Driven Engineering Languages and Systems (MODELS)* (pp. 121–135). Springer.

Arm, J., Benesl, T., Marcon, P., Bradac, Z., Schröder, T., Belyaev, A., ... others (2021). Automated Design and Integration of Asset Administration Shells in Components of Industry 4.0. *Sensors*, *21*(6).

Atkinson, C., & Kühne, T. (2021). Taming the Complexity of Digital Twins. *IEEE Software*, *39*(2), 27–32.

Becker, F., Bibow, P., Dalibor, M., Gannouni, A., Hahn, V., Hopmann, C., ... others (2021). A Conceptual Model for Digital Shadows in Industry and Its Application. In *International Conference on Conceptual Modeling (ER)* (pp. 271–281).

Bill, R., Fleck, M., Troya, J., Mayerhofer, T., & Wimmer, M. (2019). A local and global tour on MOMoT. *International Journal on Software and Systems Modeling (SoSyM)*, *18*, 1017–1046.

Blanc, X., Gervais, M.-P., & Sriplakich, P. (2005). Model Bus: Towards the Interoperability of Modelling Tools. In *Model Driven Architecture: Foundations and Applications (MDAFA)* (pp. 17–32).

Blockwitz, T., Otter, M., Akesson, J., Arnold, M., Clauss, C., Elmqvist, H., ... others (2012). Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models. In *International Modelica Conference*.

Butting, A., Kautz, O., Rumpe, B., & Wortmann, A. (2017, May). Architectural Programming with MontiArcAutomaton. In *International Conference on Software Engineering Advances (ICSEA)* (pp. 213–218).

[9] https://opcfoundation.org/about/opc-technologies/opc-ua
[10] https://www.mtconnect.org
[11] https://github.com/Azure/opendigitaltwins-dtdl/blob/master/DTDL/v2/dtdlv2.md

Butting, A., Pfeiffer, J., Rumpe, B., & Wortmann, A. (2020). A Compositional Framework for Systematic Modeling Language Reuse. In *International Conference on Model Driven Engineering Languages and Systems (MODELS)* (pp. 35–46). ACM.

Dalibor, M., Heithoff, M., Michael, J., Netz, L., Pfeiffer, J., Rumpe, B., … Wortmann, A. (2022). Generating customized low-code development platforms for digital twins. *Journal of Computer Languages (CoLa)*, *70*, 101-117.

Dalibor, M., Jansen, N., Rumpe, B., Schmalzing, D., Wachtmeister, L., Wimmer, M., & Wortmann, A. (2022). A Cross-Domain Systematic Mapping Study on Software Engineering for Digital Twins. *Journal of Systems and Software (JSS)*, *193*.

Derave, T., Sales, T. P., Gailly, F., & Poels, G. (2020). Towards a Reference Ontology for Digital Platforms. In *International Conference on Conceptual Modeling (ER)* (pp. 289–302).

Eisenberg, M., Lehner, D., Sindelar, R., & Wimmer, M. (2022). Towards Reactive Planning with Digital Twins and Model-Driven Optimization. In *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA)* (pp. 54–70).

Kappel, G., Kapsammer, E., Kargl, H., Kramler, G., Reiter, T., Retschitzegger, W., … Wimmer, M. (2006). Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages. In *International Conference on Model Driven Engineering Languages and Systems (MODELS)* (pp. 528–542).

Kephart, J. O., & Chess, D. M. (2003). The vision of autonomic computing. *IEEE Computer*, *36*(1), 41–50.

Krahn, H., Rumpe, B., & Völkel, S. (2010, September). MontiCore: a Framework for Compositional Development of Domain Specific Languages. *International Journal on Software Tools for Technology Transfer (STTT)*, *12*(5), 353–372.

Kritzinger, W., Karner, M., Traar, G., Henjes, J., & Sihn, W. (2018). Digital Twin in manufacturing: A categorical literature review and classification. *Ifac-PapersOnline*, *51*(11), 1016–1022.

Lehner, D., Pfeiffer, J., Tinsel, E., Strljic, M. M., Sint, S., Vierhauser, M., … Wimmer, M. (2022). Digital twin platforms: Requirements, capabilities, and future prospects. *IEEE Software*, *39*(2), 53–61.

Pfeiffer, J., Lehner, D., Wortmann, A., & Wimmer, M. (2022). Modeling Capabilities of Digital Twin Platforms - Old Wine in New Bottles? *Journal of Object Technology (JOT)*, *21*(3), 3:1–14.

Pfeiffer, J., Lehner, D., Wortmann, A., & Wimmer, M. (2023). Towards a Product Line Architecture for Digital Twins. In *International Conference of Software Architecture - Companion (ICSA-C)* (pp. 187–190). IEEE.

Pfeiffer, J., & Wortmann, A. (2021). Towards the Black-Box Aggregation of Language Components. In *International Conference on Model Driven Engineering Languages and Systems - Companion (MODELS-C)* (pp. 576–585). IEEE.

Pfeiffer, J., & Wortmann, A. (2023). A Low-Code Platform for Systematic Component-Oriented Language Composition. In *International Conference on Software Language Engineering (SLE)* (pp. 208–213). ACM.

Platenius-Mohr, M., Malakuti, S., Grüner, S., Schmitt, J., & Goldschmidt, T. (2020). File- and API-based interoperability of digital twins by model transformation: An IIoT case study using asset administration shell. *Future Generation Computer Systems*, *113*, 94–105.

Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). *EMF: eclipse modeling framework*. Addison-Wesley.

Tantik, E., & Anderl, R. (2017). Integrated data model and structure for the asset administration shell in Industrie 4.0. *Procedia Cirp*, *60*, 86–91.

Tratt, L. (2005). Model transformations and tool integration. *Journal on Software and Systems Modeling (SoSyM)*, *4*(2), 112–122.

Vila, M., Sancho, M.-R., Teniente, E., & Vilajosana, X. (2021). Semantics for Connectivity Management in IoT Sensing. In *International Conference on Conceptual Modeling (ER)* (pp. 297–311).

About the authors

Daniel Lehner is a PhD candidate at the Institute of Business Informatics - Software Engineering at Johannes Kepler University Linz. His research interests include applying Model-Driven Engineering techniques and practices to Digital Twins. You can contact the author at daniel.lehner@jku.at or visit https://se.jku.at/daniel-lehner.

Jérôme Pfeiffer is a research assistant at the Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW) of the University of Stuttgart. His research interests include Software Language Engineering techniques and applied Model-Driven Engineering with a focus on digital twins and Industry 4.0. You can contact the author at jerome.pfeiffer@isw.uni-stuttgart.de or visit https://www.isw.uni-stuttgart.de/en/institute/team/Pfeiffer-00005/.

Stefan Klikovits is a university assistant at the Institute of Business Informatics - Software Engineering at Johannes Kepler University Linz. His current research interests include model-driven systems engineering, autonomous driving system verification, and quantum computation. You can contact the author at stefan.klikovits@jku.at or visit https://klikovits.net.

Andreas Wortmann is a Full Professor at the Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW) of the University of Stuttgart where he conducts research on model-driven engineering, software language engineering, and systems engineering with a focus on Industry 4.0 and digital twins. You can contact the author at andreas.wortmann@isw.uni-stuttgart.de or visit www.wortmann.ac.

Manuel Wimmer is Full Professor and Head of the Institute of Business Informatics – Software Engineering at Johannes Kepler University Linz. His research interests include Software Engineering, Model-Driven Engineering, and Cyber-Physical Systems. You can contact the author at manuel.wimmer@jku.at or visit https://www.se.jku.at/manuel-wimmer.