Declarative Lifecycle Management in Digital Twins

Eduard Kamburjan
eduard@ifi.uio.no
University of Oslo
Oslo, Norway

Nelly Bencomo
nelly.bencomo@durham.ac.uk
Durham University
Durham, United Kingdom

Silvia Lizeth Tapia Tarifa
Einar Broch Johnsen
sltarifa@ifi.uio.no
einarj@ifi.uio.no
University of Oslo
Oslo, Norway

Abstract

Together, a digital twin and its physical counterpart can be seen as a self-adaptive system: the digital twin monitors the physical system, updates its own internal model of the physical system, and adjusts the physical system by means of controllers in order to maintain given requirements. As the physical system shifts between different stages in its lifecycle, these requirements, as well as the associated analyzers and controllers, may need to change. The exact triggers for such shifts in a physical system are often hard to predict, as they may be difficult to describe or even unknown; however, they can generally be observed once they have occurred, in terms of changes in the system behavior. This paper proposes an automated method for self-adaptation in digital twins to address shifts between lifecycle stages in a physical system. Our method is based on declarative descriptions of lifecycle stages for different physical assets and their associated digital twin components. Declarative lifecycle management provides a high-level, flexible method for self-adaptation of the digital twin to reflect disruptive shifts between stages in a physical system.

CCS Concepts

• **Software and its engineering → Abstraction, modeling and modularity**; Software architectures.

ACM Reference Format:
Eduard Kamburjan, Nelly Bencomo, Silvia Lizeth Tapia Tarifa, and Einar Broch Johnsen. 2024. Declarative Lifecycle Management in Digital Twins. In *ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS Companion '24), September 22–27, 2024, Linz, Austria.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3652620.3688248

1 INTRODUCTION

Digital twins are typically used in cyber-physical systems (CPS), where the physical part, the so-called *physical twin*, consists of different physical assets, and the digital part, the *digital twin* itself, consists of components that monitor, analyze and control the behavior of these physical assets [11]. The digital and physical twins interact: observations flow from the physical to the digital twin and control decisions flow back from the digital to the physical twin. Digital twins are model-centric: they maintain a live replica of the physical twin by continuously reflecting on observations of the physical system in a model that can be used for analysis and decision-making. Following an external approach to self-adaptive systems [38], the digital twin appears as a managing system and the physical twin as the managed subsystem; the managing subsystem may be realized by a MAPE-K feedback loop [7, 23].

As the digital twin needs to adapt to reflect changes in the physical twin, then digital twins need to be kept in sync with the twinned physical system to avoid model drift [24, 39, 36]. The feedback loop of the digital twin enables adjustments to fine-tune the controllers, thereby realizing *behavioral self-adaptation* [6]. However, disruptive shifts in the behavior of the physical system or in the associated requirements may invalidate the *DT components* used by the feedback loop of the digital twin to manage the physical system, including the different requirement analyzers and controllers. Whereas *behavioral self-adaptation* amounts to adjusting the behavior of the physical twin, *architectural self-adaptation* corresponds to reconfiguring the digital twin itself to reflect changes in the structure and requirements of the physical system. The goal of architectural self-adaptation in a digital twin is to maintain *architectural coherence* with the physical twin; that is, the different assets of the physical system should be matched correctly in the digital twin by the corresponding DT components.

We can often understand disruptive shifts in a physical system, which require architectural self-adaptation in the digital twin, in terms of transitions between different stages in the lifecycle of the physical system. As such, physical assets in engineering pass through stages, from design or commissioning to decommissioning (via construction and operation). For example, (i) a production line in a factory might be well-functioning until some part suddenly breaks down, potentially requiring a completely different management strategy for the entire factory; (ii) a plant in a greenhouse, which used to be healthy, might get infected, and suddenly require pesticide and a different watering regime than a healthy plant.

The physical system shifts through different stages that influence which DT components the digital twin should use. The correct configuration of the digital twin depends on the current stages of different assets in the physical system. The triggers of transitions between different stages of a physical asset might be unknown. Therefore, the transitions between the lifecycle stages of physical assets may be hard or even impossible to predict accurately by the digital twin at runtime.

This paper proposes a two-layered self-adaptive architecture (depicted in Fig. 1) that enables a digital twin to adapt to the lifecycle stages of the assets it manages. The novel notion of *stage management* is used for architectural self-adaptation of the digital

MODELS Companion '24, September 22–27, 2024, Linz, Austria

ACM ISBN 979-8-4007-0622-6/24/09
https://doi.org/10.1145/3652620.3688248

twin to reconfigure the behavioral self-adaptation layer, thereby reflecting lifecycle changes in the twinned physical system. Thus, our work separates behavioral (e.g., adjusting an existing controller to better maintain a desired policy) and architectural self-adaptation (e.g., adding a new controller to the configuration in Layer 1), as advocated in, e.g., the MORPH reference architecture [6].

Our focus in this paper is on how stage management can address architectural self-adaptation and the corresponding notion of *architectural coherence* between the digital and physical twin. Together, architectural self-adaptation and architectural coherence provide the means to answer relevant questions about the digital twin; e.g., is the digital twin's behavioral feedback loop currently using the correct DT components? Does the twin's *analyzer* component correctly evaluate the current requirements? How well do the controllers that manipulate the actuators of the physical assets (in the *executor* component) reflect the current control policy?

In addition to the behavioral self-adaptation previously discussed, the proposed architectural self-adaptation will be realized by a second MAPE-K feedback loop (depicted in blue in Fig. 1). In this additional feedback loop, the analyzer is concerned with the coherence of the digital twin concerning the physical twin, while the execute component is concerned with repair functions over DT components; i.e., architectural self-adaptation consists of reconfiguring or replacing the DT components of the first feedback loop. We need to *express* the relation between the asset and the DT components at any point in its lifecycle, *detect* the stage of an asset in its lifecycle and *adapt* to stage changes by reconfiguring the DT components. To provide the reasoning and querying capabilities for stage management, we consider the use of knowledge graphs.

Our approach to stage management is based on so-called *declarative stages*. Crucially, it is not required that the conditions that trigger transitions between stages are known. Instead, conditions are specified that express when an asset in a system should be considered to be in a certain stage, stage analyzers are used to detect these conditions, and compatibility conditions between stages are identified to ensure that self-adaptation does not fail. We illustrate our self-adaptation framework for architectural reconfiguration with a case study of an automated greenhouse. We also discuss how to apply the framework to other systems.

Contributions. The main contributions of this paper are

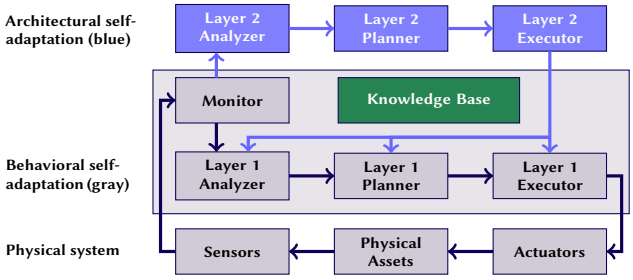

Figure 1: A two-layered self-adaptive digital twin architecture with MAPE-K feedback loops for behavioral (gray) and architectural (blue) self-adaptation.

Figure 2: A mini-greenhouse with basils and sensors.

(1) A *two-layered self-adaptive architecture for digital twins*, in which a feedback loop for architectural self-adaptation is used to reconfigure the twin's behavioral feedback loop;
(2) *Declarative stages*: a design method for modeling change in terms of the lifecycle stages of physical assets, and its formalization as a semantic model that can be used in knowledge bases; and
(3) *Stage management*: an automated method for architectural self-adaptation that selects the appropriate configuration of a digital twin, based on the modeled declarative stages.

We also provide a prototype implementation of the self-adaptive architecture, based on the GreenhouseDT exemplar [21], and an evaluation to assess the overhead when using semantic technologies to implement declarative stages over a time-consuming manual implementation.

2 MOTIVATING EXAMPLE

To illustrate self-adaptation with declarative stages, consider a digital twin for "smart mini-greenhouse" with a basil plant (see Fig. 2). The physical twin consists of the basil plant, NVDI[1] and soil moisture sensors, two pumps (one for pesticide, one for water) and a camera. The digital twin consists of a model of the basil, the pumps and the camera, together with requirement analyzers for the moisture and NVDI sensors, and requirement analyzers and controllers for each pump.

Observe that multiple lifecycles, each with different stages, can be needed to describe the assets of this physical system. We here consider the following lifecycles, and the sensors used to detect their stages:

(1) The basil is either *healthy* or *sick*, which can be detected using the NVDI measurement,
(2) the basil is either a *seed* or a *plant*, which can be detected using image classification,
(3) the pumps are either *unreliable* or *reliable*, which can be detected by the pump's requirement analyzer.

The basil requires a different target moisture level m depending on its stage, say $m \leq 0.5$ for sick plants and $m \leq 0.7$ for healthy plants, and thus a different controller. Similarly the pumps will need

[1]Normalized difference vegetation index (NVDI) is a measure that can be used for the health analysis of vegetation [26].

different controllers with varying degrees of precision, depending on their stages. The example shows how multiple lifecycles can be useful to describe a physical system, and may even involve the same physical component (e.g., the basil in our example). As these may evolve independently, the digital twin's self-adaptive behavior will need to adapt to combinations of stages from the different lifecycles.

3 BACKGROUND

3.1 Self-Adaptive Layers in Digital Twins

We here briefly explain the idea of behavioral self-adaptation in the context of digital twins. Although the literature does not provide a uniform definition of a digital twin, some definitions have been proposed [9]. Meanwhile, the MAPE-K feedback loop [7, 23, 38] has been used to underpin properties of digital twins [13, 19, 35, 30, 10, 29]. For example, Flammini [13] considers a digital twin framework using run-time predictive models for self-healing and trustworthy autonomy of cyber-physical systems (CPS), Kamburjan *et al.* [19] propose twinning-by-construction to ensure correctness for self-adaptive digital twins, Splettstößer *et al.* [35] propose a self-adaptive digital twin reference architecture to improve process quality and Pfeiffer *et al.* [29] use MAPE-K to support digital twin product line architectures.

Our work presented here also uses the MAPE-K loop for digital twins. We assume that a physical twin consists of a number of assets, and sensors and actuators associated to these assets. Behavioral self-adaptation is concerned with adjusting the behavior of the assets in the physical twin, based on the analysis of a model of these assets. Behavioral self-adaptation is realized in terms of a MAPE-K feedback loop with the following DT components: *monitor* components that collect streams of observations from the assets of the physical twin and updates the digital twin's knowledge model of these assets (sometimes called the *digital shadow*), *analyzer* components that analyze these streams with respect to a set of requirements and triggers the need of behavioral self-adaptation when these requirements are not longer met, *planner* components that determine which adaptation actions are needed to meet the requirements, and *executor* components that execute the changes in the behavior of the *controller* that control the physical twin via given actuators. Digital twins are model-centric, as these components interact with a *knowledge base* that maintains a model of the physical system. Behavioral self-adaptation corresponds to the Layer 1 feedback loop depicted in gray in Fig. 1.

3.2 Semantic Technologies

We introduce knowledge graphs and ontologies by example, with a focus on the underlying technologies and their capabilities. In particular, we focus on the Web Ontology Language (OWL) [15] and its Manchester syntax [18], which we use for modeling and queries.

Knowledge Graphs. A knowledge graph consists of a set of triples and an ontology (discussed below). In a *triple* (s, p, o), the subject s is an individual or class name, p a predicate and object o an individual, class name or data value.

Example 1. *We can express that an asset* `ast` *is a basil and has the nvdi value* 4 *with the two triples*

(ast1,nvdi,4) (ast1, *a*, Basil).

Here, ast1 *is an individual,* nvdi *a property,* 4 *a data value,* Basil *a class name and a a property for class membership.*

Ontologies. Ontologies have been proposed for knowledge representation in numerous digital twins applications [22]. An *ontology* is a set of axioms over classes C and properties P, describing conditions that must hold for all triples in a knowledge graph. We briefly introduce OWL classes and properties, using the standard OWL 2.0.[2] OWL also has data types, for now we will only need the type int for integers. OWL distinguishes between object properties, that are relations between individuals, and data properties, which are relations between individuals and data values. Below, the notation $(\cdot)$? denotes optional elements.

A *class* is a named set of individuals. For our needs, a class C has the following form, where D, D_i are other class names:

Class: C **SubClassOf**: D
(**DisjointWith**: $D_1,\ldots,D_n$)? (**EquivalentTo**: Ax)?
(**DisjointUnionOf**: $D_1,\ldots,D_n$)?

The **SubClassOf**: D clause expresses that the class C is a subclass of D, while **DisjointWith**: expresses disjointness. The **EquivalentTo**: clause describes how to derive the membership of an individual to C and **DisjointUnionOf**: describes that class C is a union of certain other classes, which are all pairwise disjoint. The symbol Ax is a conjunction (using the symbol **and**) of class names and restrictions, which take the following forms: P **some** T[> n], P **some** T[≤ n] or P **some** C (where P is a property name). Each restriction expresses that an individual i has at least one triple with predicate P and an object described by C or T, possibly with a numeric restriction.

An *object property* is a named relation between two classes; for example, an object property P with a domain C_1 and a range C_2 can be expressed by

ObjectProperty: P (**SubPropertyOf**: S)?
Domain: C_1 **Range**:C_2.

where the optional clause **SubPropertyOf**: S expresses that it is a subset of another object property S. Similarly, a *data property* is a named relation between a class and a data type; for example, a data property P with a domain C and a range T can be expressed by

DataProperty: P **Domain**: C **Range**: T
(**Characteristics**: **functional**)?

The optional clause **Characteristics**: **functional** additionally expresses that the relation P is a function.

Queries. Queries are answered by logical deduction over the defined structure. Given a knowledge graph, a *membership query* member(C) returns all individuals that are members of the class C. Given an ontology, a *disjointness query* disjoint($C_1, \ldots, C_n$) decides whether the classes $C_1, \ldots, C_n$ share individuals.

4 DECLARATIVE STAGES

A *declarative stage* provides a description of how a lifecycle stage of a physical system can be recognized and how the digital twin needs to adapt when the physical system transitions into this lifecycle stage, but not why the transition into the stage occurs. For our

[2] For details on OWL 2.0, see https://www.w3.org/TR/owl2-overview/.

work presented in this paper, it is key that this design realizes a clear separation of concerns between describing lifecycle stages and describing temporal aspects such as the triggering conditions for changes between these stages.

Self-Adaptation. During the life of the system, different assets may change between different stages in their respective lifecycles, and the digital twin will need to adapt *its architectural configuration* accordingly. This adaptation may involve removing DT components that are no longer valid and starting up new DT components that may be needed. For simplicity, we will focus the discussion on requirement analyzers and controllers.

A digital twin that uses declarative stages must make them operationally; i.e., the declarative stages steps need to be incorporated in an algorithm for self-adaptation. To this end, we consider a MAPE-K feedback loop associated with declarative stages as follows.

- **Monitor:** The digital twin collects streams of observations concerning the physical twin's assets and updates the digital twin's knowledge model of these assets.
- **Analyzer:** For a given declarative stage, all assets at that stage are analyzed for architectural coherence; for example, the digital twin checks that a plant in a sick stage has the correct requirement analyzer and controller for that stage.
- **Planner:** For each inconsistent asset, the digital twin identifies adaptation actions for the detected inconsistencies; for example, a sick plant might have an associated requirement analyzer for $m \leq 7$ when it should have a requirement analyzer for $m \leq 5$. Other DT components are handled analogously (e.g., controllers).
- **Executor:** The digital twin executes the required changes, including the initialization and removal of requirement analyzers and controllers.

Already at this point, we can elicit some requirements.

(1) The knowledge base will need to not only include information about the asset, but also about the DT components. Additionally, the digital twin needs to provide reasoning and query capabilities to deduce the declarative stage to which an asset belongs.
(2) We will need to ensure some basic properties about the declarative stages; in particular, different stages in a lifecycle should not overlap, and an asset that is part of multiple lifecycles cannot be inconsistent.

An asset is considered *inconsistent* if it can be in two stages of the same lifecycle at the same time.

Observe that declarative stages apply to assets that can be identified independently of the current stage in their lifecycle, in terms of their *asset class*. For example, a plant has a different asset class than, let us say, a pump.

Definition 1 (Asset classes). *An* asset class $\mathcal{A}$ *is a set of assets. We assume that the assets in different asset classes are disjoint, i.e.,* $\mathcal{A}_1 \cap \mathcal{A}_2 = \emptyset$.

For example, the class of all plants in our example is $\mathcal{A}_{\mathsf{Plant}}$ and each asset has a distinct identifier, e.g., $\mathsf{plant1} \in \mathcal{A}_{\mathsf{Plant}}$.

Declarative stages. A declarative stage in a lifecycle describes the conditions that determine that an asset is in the stage, which requirement analyzer and controller the digital twin needs for the asset in this stage, and how to generate the requirement analyzer and the controller when the asset enters this stage in the lifecycle. The central notion we use is a *consistent twin*: a digital twin is consistent if the set of DT components for each asset is correct according to its current stage in the lifecycle.

We now provide definitions to make this notion precise, where we impose no restrictions on the exact nature of a DT component for our purposes:

Definition 2 (Component class). *A* component class C *is a set of assets. We assume that components in different classes are disjoint, i.e.,* $C_1 \cap C_2 = \emptyset$.

In our examples, we focus on two kinds of DT components: requirement analyzers and controllers. Let us denote by $\mathcal{I}$ the *observational inputs* to the digital twin, such as the streams of sensor data. To access the current status of an asset a with respect to an input stream i, we write $i(a)$.

Example 2 (Requirement analyzers and controllers). *A* requirement analyzer $ranl : \mathcal{I} \to \mathbb{B}$ *is a function from input streams to Booleans, modeling the verdict. A* controller $ctrl : \mathcal{I} \to \mathbb{R}$ *is a function from input streams to a real number modeling the control decision.*

We define declarative stages formally as follows.

Definition 3 (Declarative stages). *Let* $\mathcal{A}$ *be an asset class,* $\overline{C} = \{C_1, \ldots, C_n\}$ *a set of component classes and* $C = \bigcup_i C_i$ *their union. A* declarative stage $\mathsf{D}_{\mathcal{A},\overline{C}}$ *is a pair*

$$\mathsf{D}_{\mathcal{A},\overline{C}} = \langle member, consistent \rangle,$$

where

- $member \subseteq \mathcal{A}$ *is a set of assets;*
- *the relation* $consistent \subseteq member \times 2^C$ *describes for a given asset all sets of DT components with which it is consistent.*

We let X range over sets of DT components and say that a declarative stage $\mathsf{D}_{\mathcal{A},\overline{C}}$ is C_i*-requiring* for some $C_i \in \overline{C}$ if

$$\forall a \in \mathcal{A}, X \in 2^C.\ (a, X) \in consistent \to \exists c \in C_i.\ c \in X.$$

This means that a member of C_i is required for consistency. Similarly, a declarative stage $\mathsf{D}_{\mathcal{A},\overline{C}}$ is C_i*-invariant* for some $C_i \in \overline{C}$ if

$$\forall a \in \mathcal{A}, X \in 2^C.\ (a, X) \in consistent \to \forall c \in C_i.\ c \in X.$$

For a C_i*-invariant* declarative stage, consistency does not depend on members of C_i.

In the sequel, we assume membership in the sets $member_{\mathsf{D}_{\mathcal{A}}}$ and $consistent_{\mathsf{D}_{\mathcal{A}}}$ to be decidable. We only use additional subscripts to disambiguate declarative stages and their elements if these are unclear from the context.

Example 3. *We illustrate the specification of declarative stages by considering sick and healthy plants. The corresponding declarative stages* $\mathsf{D}_{\mathsf{Sick}}$ *and* $\mathsf{D}_{\mathsf{Healthy}}$ *are shown in Fig. 3. The NVDI measurement is used to determine whether a plant a is sick of healthy. The two stages use different moisture settings in the corresponding requirement analyzers. Here, we use* $i_{\mathsf{nvdi}}(a)$ *to denote the value of the NVDI signal from plant a,* $i_{\mathsf{m}}(a)$ *to denote the value of the moisture signal from plant a, and* $\mathsf{ranl}_m^{\leq x}(a)$ *to denote the requirement analyzer for* $i_{\mathsf{m}}(a) \leq x$*. As before, X ranges over sets of DT components.*

$$
\begin{aligned}
\mathsf{D}_{\mathsf{Sick}} &= \{member_{\mathsf{Sick}}, consistent_{\mathsf{Sick}}\} \\
member_{\mathsf{Sick}} &= \{a \mid i_{\mathsf{nvdi}}(a) \leq 0.5\} \\
consistent_{\mathsf{Sick}} &= \{(a, X) \mid a \in member_{\mathsf{Sick}}, \\
&\qquad \mathsf{ranl}^{\leq 5}_{m}(a) \in X\} \\
\mathsf{D}_{\mathsf{Healthy}} &= \{member_{\mathsf{Healthy}}, consistent_{\mathsf{Healthy}}\} \\
member_{\mathsf{Healthy}} &= \{a \mid i_{\mathsf{nvdi}}(a) > 0.5\} \\
consistent_{\mathsf{Healthy}} &= \{(a, X) \mid a \in member_{\mathsf{Healthy}}, \\
&\qquad \mathsf{ranl}^{\leq 10}_{m}(a) \in X\}
\end{aligned}
$$

Figure 3: Declarative stages for Example 3.

We can easily see that the two stages of Example 3 are disjoint, yet the two stages cover all plants. This illustrates the concept of a *lifecycle* as a set of disjoint stages that cover a particular kind of asset, formally defined as follows:

Definition 4 (Lifecycle). *Let $\mathcal{A}$ be an asset class and I an index set. A* lifecycle $\mathsf{L}_{\mathcal{A}}$ *for $\mathcal{A}$ consists of a set of declarative stages $(\mathsf{D}_{\mathcal{A},\overline{C},i})_{i\in I}$ such that the following conditions hold:*

- $\mathcal{A} = \bigcup_{i\in I} member_{\mathsf{D}_{\mathcal{A},\overline{C},i}}$
- $\forall i, j \in I.\ i \neq j \Rightarrow member_{\mathsf{D}_{\mathcal{A},\overline{C},i}} \cap member_{\mathsf{D}_{\mathcal{A},\overline{C},j}} = \emptyset$

*A lifecycle is C_i-*ensuring *if all its stages are C_i-requiring.*

Observe that an asset can be part of multiple lifecycles, in which case basic compatibility must be ensured. For example, if the plant is additionally part of a lifecycle for commissioning, operations, maintenance and decommissioning, it must be possible to configure the digital twin for all 8 combinations of declarative staged from these lifecycles. Ensuring lifecycles are especially critical for controllers: For a single lifecycle, it is easy to check that there is always exactly one controller. However, when multiple lifecycles apply to the same asset, two stages that can occur at the same time cannot require different controllers. From the perspective of self-adaptation, even if an asset is in two different declarative stages, a plan to reconfigure the twin must exist.

Definition 5 (Compatible stages and lifecycles). *Let D_1 and D_2 be declarative stages of some asset class $\mathcal{A}$ and component classes $\overline{C} = \{C_1, \ldots, C_n\}$.*

- *The stages D_1 and D_2 are* compatible *if, for all $a \in member_{\mathsf{D}_1} \cap member_{\mathsf{D}_2}$ there is some $X \subseteq \overline{C}$ such that $(a, X) \in consistent_{\mathsf{D}_1}$ and $(a, X) \in consistent_{\mathsf{D}_2}$.*
- *Two lifecycles are* compatible *if all their declarative stages are pair-wise compatible.*

Using declarative stages, the transitions between the stages in a lifecycle need not be modeled; it suffices to describe architectural coherence for each stage. A complete model of the lifecycles is not needed, it suffices to provide a declarative characterization of aspects relevant for architectural coherence of the digital twin.

Remark that declarative stages describe how to *deduce* whether an asset is at a certain stage, not how to remove and create DT components once a stage change has been detected. If the asset is inconsistent, then deduction is not enough — instead, the system must *abduce* which DT components could explain the assumption that the asset is consistent, under the assumption that it is indeed at the given stage. For example, if $a \in member_{\mathsf{Sick}}$ is given, then it is easy to see that there must be some $\mathsf{ranl}^{\leq 5}_{m}(a) \in X$ to explain $(a, X) \in consistent_{\mathsf{Sick}}$.

To track an asset through its lifecycle and establish architectural coherence at the current stage, the MAPE-K loop of the digital twin needs the following deductive capabilities:

(1) the digital twin needs a *knowledge model* that can be queried for the current declarative stages of the assets;
(2) the digital twin must be able to *determine its architectural coherence* (i.e., that the digital twin's DT components such as the requirement analyzers and controllers correctly reflect the declarative stages of the assets);
(3) the digital twin must be able to *decide on stage membership* for the assets based on observational inputs; and
(4) the digital twin must be able to *explain inconsistencies* and derive plans to reestablish consistency.

In short, we need a uniform representation of asset information and of the digital twin's architectural configuration. The next sections show that these capabilities can be realized by means of semantic technologies.

5 A SEMANTIC REPRESENTATION OF DECLARATIVE STAGES

The formalization of declarative stages in Sect. 4 defines a general and abstract framework that forms the basis of our self-adaptive architecture. As discussed above, this framework relies on being able to (a) *model* stage membership, (b) efficiently *reason* about stage membership and consistency between stages in terms of this model, and (c) update the stage membership model when changes to the lifecycle stages of assets are detected. In this section, we discuss how a knowledge base with these deductive and abductive capabilities can be realized in terms of *knowledge graphs* [17], using *ontologies* and associated *semantic technologies*.

A knowledge graph enables a uniform representation of declarative stages and provides a technological platform for queries and reasoning tasks. We now introduce basic terminology to express properties of assets, DT components, and their relations. For our greenhouse example, we will consider an ontology with classes for requirement analyzers and controllers.

Definition 6 (Stage Ontology). *The* stage ontology *expresses the existence of assets and DT components, as well as their disjointness and relation.*

Class: Asset **DisjointWith**: Component
ObjectProperty: assignedTo
Domain: Asset **Range**: Component

In the sequel, we assume that every ontology we consider contains the axioms from Def. 6. Simple extensions of this ontology can provide the context of a specific system or environment. We further assume that for a set of component classes $\overline{C} = \{C_1, \ldots, C_n\}$, the axiom **Class**: Component **DisjointUnionOf**: $C_1, \ldots, C_n$ is added.

Example 4. *For the running example of a greenhouse, we need information about plants, the NVDI value, and the two requirement analyzers that we consider.*

Class: Component **DisjointUnionOf**: RAnalyzer, Controller
ObjectProperty: analyzedBy **SubPropertyOf**: assignedTo
Domain: Asset **Range**: RAnalyzer
ObjectProperty: controlledBy **SubPropertyOf**: assignedTo
Domain: Asset **Range**: Controller
Class: RAnalyzerMoistUnder5 **SubClassOf**: RAnalyzer
Class: RAnalyzerMoistUnder10 **SubClassOf**: RAnalyzer
Class: Basil **SubClassOf**: Asset **and** nvdi **some** int
DataProperty: nvdi **Domain**: Basil **Range**: xsd::int
Characteristics: **functional**

Here, we model the requirement analyzer classes as disjoint; using semantic technologies, we can also express subtype relations on requirement analyzer classes using subclass axioms.

A semantic stage is a representation of a declarative stage $\mathsf{D}_{\mathcal{A},\overline{C}} = \langle member, consistent \rangle$, where all conditions and properties of membership and consistency are expressed in terms of OWL axioms.

Definition 7 (Semantic Stages). *Let $\mathcal{A}$ be an asset class, C_i be subclasses of* Component. *A* semantic stage $\mathsf{S}_{\mathcal{A},\overline{C}} = \langle \mathsf{S}_{\text{member}}, \mathsf{S}_{\text{cons}} \rangle$ *is a pair of two OWL class names $\mathsf{S}_{\text{member}}$ and S_{cons}. The following axioms must hold for $\mathsf{S}_{\text{member}}$ and S_{cons}:*

Class: $\mathsf{S}_{\text{member}}$ **SubClassOf**: A
Class: S_{cons} **EquivalentTo**: $\mathsf{S}_{\text{member}}$ **and** SC

The symbol SC *is a concept that needs to be defined through the classes M_i and optionally over* C. *The definition has the following form, where $[\,\cdot\,]$ denotes an optional element.*

SC **EquivalentTo**:
analyzedBy **some** RA_1 **and** … **and** analyzedBy **some** RA_n
[**and** controlledBy **some** C]

Additionally, we demand that $\mathsf{S}_{\text{member}}$ is independent of RAnalyzer *and* Controller, *i.e, neither* RAnalyzer, Controller *nor their subclasses occur in any axiom for $\mathsf{S}_{\text{member}}$.*[3]

In a semantic stage S, we say that $\mathsf{S}_{\text{member}}$ is the *membership class* and S_{cons} the *consistency class* of the stage S.

Example 5. *We consider semantic stages corresponding to the declarative stages from Fig. 3. First the healthy stage*

$$\mathsf{S}_{\text{Healthy}} = \langle \text{Healthy}, \text{HealthyCons} \rangle$$

can be formalized as follows:

Class: Healthy **SubClassOf**: Basil **and** nvdi **some** int [> 5]
Class: HealthyCons **EquivalentTo**: Healthy
and analyzedBy **some** RAnalyzerMoistUnder10

Next, the sick stage $\mathsf{S}_{\text{Sick}} = \langle \text{Sick}, \text{SickCons} \rangle$ is formalized by

Class: Sick **SubClassOf**: Basil **and** nvdi **some** int[≤5]
Class: SickCons **EquivalentTo**: Sick
and analyzedBy **some** RAnalyzerMoistUnder5

A *lifecycle* consists of semantic stages that have disjoint members. To define semantic lifecycles, we first formalize compatibility and disjointness of semantic stages as follows:

[3]This condition can be made more precise with ontology modules [32]: None of these classes should be in the module of $\mathsf{S}_{\text{member}}$.

Definition 8. *Let $\mathsf{S}_1 = \langle \mathsf{S}^1_{\text{member}}, \mathsf{S}^1_{\text{cons}} \rangle$, $\mathsf{S}_2 = \langle \mathsf{S}^2_{\text{member}}, \mathsf{S}^2_{\text{cons}} \rangle$ be semantic stages. Then*

- *S_1 and S_2 are* disjoint *if their membership classes are disjoint, i.e.,* $\text{disjoint}(\mathsf{S}^1_{\text{member}}, \mathsf{S}^2_{\text{member}})$, *and that*
- *S_1 and S_2 are* compatible *if their consistency classes are not disjoint, i.e.,* $\neg\text{disjoint}(\mathsf{S}^1_{\text{cons}}, \mathsf{S}^2_{\text{cons}})$.

Definition 9 (Semantic Lifecycle). *A* semantic lifecycle *for an asset class $\mathcal{A}$ is a set of stages $\mathsf{S}_i = \langle \mathsf{S}^i_{\text{member}}, \mathsf{S}^i_{\text{cons}} \rangle$ for $\mathcal{A}$ such all consistency classes are disjoint:*

$$\text{disjoint}(\mathsf{S}^1_{\text{cons}}, \ldots, \mathsf{S}^n_{\text{cons}}).$$

Remark that patterns that capture the relations between two classes and the axioms required for them, as used above, are not directly expressible in OWL. Instead, they are formalized using another semantic technology, namely *ontology templates* [34]. The templates for our system are given in the auxiliary online material.[4] The conditions for entailment and satisfiability are standard tasks for OWL and DL reasoners, and supported by efficient implementations.

To use equivalence axioms together with an open-world assumption, the knowledge graph may not store membership to any $\mathsf{S}_{\text{member}}$. Otherwise, the reasoner could deduce the existence of requirement analyzers, even if they are not explicitly present. This issue can be solved in several ways. One can require that membership to stages is not stored (which is our solution), one could add additional axioms that essentially enforce a closed-world semantics, or one could use a different formalization for stages that does not require open-world reasoning, such as SHACL shapes [37].

Note that repair of the digital twin configuration corresponds to so-called ABox abduction [25] in the knowledge graph: it is sufficient to abduce the presence of individuals, and these must be members of a fixed set of possible classes.

6 A SELF-ADAPTIVE ARCHITECTURE FOR LIFECYCLE MANAGEMENT

The overall aim of the self-adaptive architecture is to maintain the architectural coherence of the digital twin while assets transition between the declarative stages in their lifecycles; i.e., every asset must have the correct associated DT components, reflecting the current declarative stages of the asset. The architecture of the self-adaptive digital twin is depicted in Fig. 4 for two generic DT components, such as requirement analyzers and controllers. It has the following architectural components; the first four constitute the *managing system*, while the DT components, together with the physical system, constitute the *managed system* of the second layer self-adaptive feedback loop. We introduce the architectural components and position them with respect to the MAPE-K feedback loops of Fig. 1:

- **Knowledge Base:** The knowledge base (KB) keeps track of information about DT components, assets and stages. The KB offers operations for the other components to manipulate and query this information.
- **Monitor:** The monitor component includes a semantic tagger that acts as the entry point for information about the

[4]https://github.com/Edkamb/StageAdapt

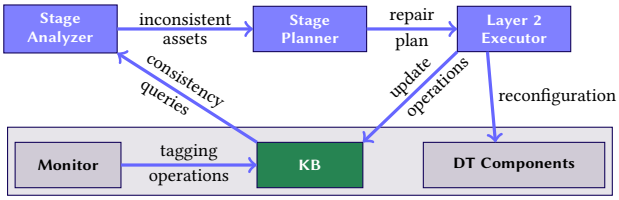

Figure 4: A self-adaptive architecture for stage-based lifecycle management.

assets of the twinned system, translates this information into semantic data and adds it to the KB. Although the Monitor component is used for both the behavioral and architectural feedback loop of Fig. 1, we here focus on its use for architectural self-adaptation.

- **Stage Analyzer:** The stage analyzer identifies changes to the lifecycle stages of the different assets by querying the KB for inconsistencies. This component is the Layer 2 Analyzer of Fig. 1.
- **Stage Planner:** The stage planner determines how to adapt the DT components of Layer 1 to resolve the inconsistencies in the KB. It identifies DT components that need to be added and removed. This component is the Layer 2 Planner of Fig. 1.
- **DT components:** The DT components address behavioral self-adaptation in the digital twins, including requirement analysis, planning and control for the different physical assets. These are the Layer 1 components of Fig. 1.

We now consider each component by itself and illustrate how declarative stages are used for self-adaptation in the architecture. We omit the DT components, which were discussed in Sect. 4, and the Layer 2 Executor component, which is straightforward given the output from the stage planner. In the sequel, we assume that the declarative stages are given a semantic representation (cf. Sect. 5).

Knowledge Base. The task of the KB is to manage knowledge about the assets, their states, as well as the architectural configuration of the digital twin; i.e., the KB tracks information about declarative stages, requirement analyzers and controllers. This requires a knowledge store for the information, operations to add and remove information, and a query interface that utilizes the managed knowledge to enable the stage analyzer to detect lifecycle changes and the stage planner to plan changes to the digital twin architecture.

For declarative stages that are given a semantic representation, the KB is exactly a knowledge graph with a set of predefined operations to manage information: while arbitrary reads can be allowed, the addition of new information must adhere to the patterns described by the ontology. A more subtle point is that OWL reasoning has an open-world semantics: it can deduce the existence of individuals that are not explicitly named in the knowledge graph. To circumvent such reasoning, we need to manually add axioms that list all individuals that belong to the class Asset or subclasses of Component. These *closing axioms* take the following form for a class C and individuals $i_1, \ldots, i_n$:

$$\text{C } \textbf{EquivalentTo } \{i_1, \ldots, i_n\}$$

$$\begin{aligned}
&[\![\mathcal{K} \sqcup \{\text{A } \textbf{EquivalentTo}\text{: } \{a_1, \ldots, a_n\}\}, \text{ADD(ast, A)}]\!] \\
&\quad = \mathcal{K} \cup \{\text{A } \textbf{EquivalentTo}\text{: } \{a_1, \ldots, a_n, \text{ast}\}\} \cup \{(\text{ast}, a, \text{A})\} \\
&[\![\mathcal{K} \sqcup \{\text{C } \textbf{EquivalentTo}\text{: } \{c_1, \ldots, c_n\}\}, \text{CLEAN(ast,C)}]\!] \\
&\quad = \big(\mathcal{K} \setminus \text{neighbor}_{\mathcal{K}}(\text{ast})\big) \\
&\qquad \cup \{\text{C } \textbf{EquivalentTo}\text{: } \big(\{c_1, \ldots, c_n\} \setminus \text{C}_{\mathcal{K}}(\text{ast})\big)\} \\
&[\![\mathcal{K} \sqcup \{(\text{ast}, a, \text{A}), \text{A } \textbf{EquivalentTo}\text{: } \{a_1, \ldots, a_n, \text{ast}\}\}, \\
&\quad \text{REMOVE(ast)}]\!] = \bigcap_{\text{Component class C}} [\![\mathcal{K}, \text{CLEAN(ast,C)}]\!] \\
&\qquad \cup \{\text{A } \textbf{EquivalentTo}\text{: } \{a_1, \ldots, a_n\}\} \\
&[\![\mathcal{K}, \text{UPDATE(ast, p, v)}]\!] \\
&\quad = \big(\mathcal{K} \setminus \{(\text{ast}, \text{p}, \text{X}) \mid \text{X} \in \text{dom(p)}\big) \cup \{(\text{ast}, \text{p}, \text{v})\} \\
&[\![\mathcal{K} \sqcup \{\text{C } \textbf{EquivalentTo}\text{: } \{c_1, \ldots, c_n\}\}, \text{ADD(ast, c, C)}]\!] \\
&\quad = \mathcal{K} \cup \{\text{C } \textbf{EquivalentTo}\text{: } \{\text{c}, c_1, \ldots, c_n\}\} \\
&\qquad \cup \{(\text{ast}, \text{assignedTo}, \text{c}), (\text{c}, a, \text{C})\}
\end{aligned}$$

Figure 5: Semantics of knowledge store operations.

Closing axioms are only added for the aforementioned classes — in contrast, for the classes describing the stages, we explicitly want to deduce membership and not store this information.

Example 6. *Consider the following knowledge graph $\mathcal{K}$ that models the semantic stages from the previous section, as well as one asset with one requirement analyzer and one controller.*

The set of stored triples $\mathcal{K}_{\text{store}}$ is as follows:

(ast1, *a*, Basil), (ast1, nvdi, 4)
(ast1, analyzedBy, rq1), (ast1, controlledBy, ctrl1)
(rq1, *a*, RAnalyzerMoistUnder5), (ctrl1, *a*, BasilController)

Note that the water level is not *stored in the KB. It is handled by the requirement analyzer and is not relevant for stage membership or consistency. In addition to the axioms of Def. 6 and Examples 4 and 5, the ontology $\mathcal{K}_{\text{ontology}}$ contains the following axioms:*

Asset **EquivalentTo** { ast1 }
RAnalyzer **EquivalentTo** { rq1}
Controller **EquivalentTo** { ctrl1 }

To manipulate the KB, we need to add and remove information. We implement a generic operation interface that supports these operations for assets, stages and DT components. The set of known declarative stages, however, remains static.

Definition 10 (Operations). *Let $\sqcup$ denote disjoint union,* A *an OWL asset class,* C *an OWL controller class, and* RA *an OWL requirement analyzer class. Given a knowledge graph $\mathcal{K}$ and an asset node* ast*, we let* neighbor$_{\mathcal{K}}$(ast) *denote the set of triples of the form* (ast, analyzedBy, ra), (ast, controlledBy, c) *in $\mathcal{K}$, and* ranalyzers$_{\mathcal{K}}$(ast) *denote the set of requirements analyzers ra with triples of the form* (ast, analyzedBy, ra). *Furthermore, let* control$_{\mathcal{K}}$(ast) *be the set*

$$\{(\text{ast}, \text{controlledBy}, \text{ctrl}), (\text{ctrl}, a, \text{C})\}$$

if such a ctrl *exists, or $\emptyset$ if not. The set of operations on knowledge graphs, and their effects, is defined in Fig. 5.*

Operation ADD(ast, A) adds a new asset to the KB. This requires to add it to both the closing axiom and to add a new triple that connects the asset ast with its asset kind A. Operation CLEAN(ast , C) removes all DT components of class C assigned to an asset ast. Operation REMOVE(ast) removes an asset, as well as all its DT components. Operation UPDATE(ast, p, v) updates the information

- ADD(ast,A) maps to

  `INSERT DATA { uri(ast) rdf : type uri(A) }`
- ADD(ast,c,C) maps to

  `INSERT DATA { uri(c) rdf : type uri(C). uri(ast) assignedTo uri(c) }`
- UPDATE(ast,p,v) maps to

  `DELETE WHERE { uri(ast) uri(p) ?a}`
  `INSERT DATA { uri(ast) uri(p) v}`
- REMOVE(ast) maps to

  `DELETE WHERE { uri(ast) rdf : type ?a }`

  Followed by CLEAN(ast,C)
- CLEAN(ast,C) maps to

  `DELETE WHERE { uri(ast) ?x ?y.}`
  `DELETE WHERE { ?a ?b uri(ast).}`
  `DELETE DATA { uri(C) rdf : type Component}`

Figure 6: Mapping operations to SPARQL queries. Operation `uri( · )` retrieves the URI of an asset, components, asset kind or other parameter.

modeled by property p for asset ast to value v. Finally, operation ADD(ast, C, M) adds a DT component to the KB, again updating the relevant closing axioms as well as adding a triple that connects the nodes and classes. The definition of these operations in terms of SPARQL queries is given in Fig. 6. Note that the interface is generic for declarative stages, while the implementation is specific for semantic stages.

Not every architectural component is supposed to use every operation. The operations on DT components are used only to reflect the changes during reconfiguration, while the operations on asset and state are updated based on incoming data.

Example 7. *We continue with Example 6. We update the NVDI value of the asset by the* UPDATE(ast1, nvdi, 4) *operation and add a new asset by the* ADD(ast2, Basil) *operation. After these operations, the updated knowledge graph* $\mathcal{K}^1_{\text{store}}$ *contains the following triples:*

(ast1, *a*, Basil), (ast1, nvdi, 6)
(ast2, *a*, Basil)
(ast1, analyzedBy, rq1), (ast1, controlledBy, ctrl1)
(rq1, *a*, RAnalyzerMoistUnder5), (ctrl1, *a*, BasilController)

Its ontology $\mathcal{K}^1_{\text{ontology}}$ *contains, in addition to the axioms of Def. 6 and Examples 4 and 5, the following axioms:*

Asset **EquivalentTo** {ast1, ast2}
RAnalyzer **EquivalentTo** rq1}
Controller **EquivalentTo** {ctrl1}

For queries, we allow OWL membership queries on Boolean combinations of OWL classes.

Definition 11 (Queries). *A* composed class *is the closure of OWL class names under conjunction (***and***) and negation (***not***). Let* C *be a composed OWL class and* $\mathcal{K}$ *a knowledge graph. We denote with* $[\![\mathcal{K}, \mathsf{C}]\!]$ *the set of individuals described by* C *according to the OWL semantics. Additionally, we let* $\mathsf{C}_{\mathcal{K}}(\mathsf{a})$ *denote the query that returns all the DT components of class* C *that are assigned to* a *in knowledge graph* $\mathcal{K}$.

```
1  while(true)
2   toGenerate := ∅, toAdd := ∅, toRemove := ∅
3   foreach stage S = ⟨S_member, S_cons⟩
4    // Analyzer: are there inconsistencies?
5    V := [[K, S_member and not S_cons]]
6
7    // With which stage should the asset be consistent?
8    foreach a ∈ V
9     toGenerate := toGenerate ∪{a, S}
10   end
11
12   // Planner: How to make the asset consistent?
13   K' := copy(K)
14   foreach asset a with class C
15    K' := [[K', CLEAN(a, C)]]
16   end
17   K' := abduce(K', C̄)
18
19   // What needs to be added or removed?
20   toAdd := K' \ K
21   toRemove := K \ K'
22  end
23 end
```

Figure 7: A self-adaptation algorithm combining stage analyzer and planner.

Example 8. *Retrieving all inconsistent, healthy basil plants requires querying for the class* Healthy **and not** HealthyCons.

For the running example, this query returns both assets: first, ast1 *has the wrong requirement analyzer (recall that the update performed in Example 6 moved it from its sick to its healthy stage), and second,* ast2 *has no requirement analyzer:*

$$[\![\mathcal{K}^1, \text{Healthy } \textbf{and not } \text{HealthyCons}]\!] = \{\text{ast1}, \text{st2}\}$$

Monitor. The monitor component takes asset observation streams as input and performs the following tasks:

- the monitor uses a semantic tagger as a *mapper* that translates the streams of asset observations into semantic information that are added to the knowledge graph, using predefined operations;
- the monitor acts as a *filter*: not all information is added to the KB, but only information needed for self-adaptation; and
- the monitor directly issues operations to remove an asset, its requirement analyzers and its eventual controller from the KB and system, if the asset observation stream notifies of its removal from the physical system.

Concretely, the monitor issues an UPDATE operation whenever information about an asset needs to be updated in the KB, an ADD operation whenever an asset is added to the system, and an REMOVE operation whenever an asset is removed.

Stage Analyzer. The stage analyzer is responsible for detecting inconsistencies between the current stage of an asset and its associated DT components in the digital twin. To this aim, the stage analyzer queries the KB for the assets that satisfy stage membership of different stages and checks whether the DT components of these assets comply with the consistency relation of the stage. The stage analyzer is realized by the first part of the algorithm in Fig. 7, which implements the MAPE-K loop for architectural self-adaptation. The algorithm iterates through all stages and finds the inconsistent assets (l. 5). It then collects all such assets in a set (l. 8) that records the stage with which the assets need to be consistent (l. 9).

Stage Planner. The stage planner is responsible for repairing the digital twin once an inconsistency is detected between the DT components and the stage associated with an asset. The stage planner requires a set of compatible lifecycles, and is realized by the second part of the algorithm in Fig. 7.

Repair is based on abducing an explanation for the required consistency. This explanation may add or remove DT components. Thus, abduction is performed on all inconsistencies at once. We formalize this procedure as follows. First, copy the KB (l. 13), then remove all DT components (l. 14). The abduction then derives the DT components that are needed to explain the consistency of this KB (l. 17). Finally, abduced components that are not already present,[5] are added to the digital twin and to the original KB (l. 20). DT components that are no longer needed, are similarly removed (l. 21).

7 Evaluation

We evaluate declarative stages and their realization as semantic stages to answer the following research questions:

RQ1 Can declarative stages be used to model an existing digital twin system?

RQ2 How does using semantic stages over non-semantic declarative stages effect performance?

RQ1 is qualitative and indicates the general applicability of declarative stages, while **RQ2** is quantitative and estimates the overhead caused by using semantic technologies to implement the declarative stages, compared to a direct (ad hoc) implementation. We performed two experiments, available in the auxiliary online material.

Experimental Design and Setup. Experiment **EX1** addresses **RQ1** and uses the example from Sect. 2, based on GreenhouseDT [21], a digital twin exemplar specifically designed for structural self-adaptation. We have implemented both semantic stages, as well as non-semantic declarative stages. Non-semantic stages were implemented using suitable data structures for the knowledge base, while semantic stages were implemented with an RDF knowledge graph based on Apache Jena[6] and its built-in reasoner. Thus, adding, e.g., a new stage requires to define it in the ontology and implement the corresponding class in the architecture, respectively.

EX2 consists of a series of configurations, based on the running example. The experiments consider n different stages in one lifecycle and m assets. Each stage defines one interval for the `nvdi` property, and is validated using the OTTR templates. For each configuration (n, m), we measure the time needed for the non-semantic and semantic stages to adapt from a random starting state, respectively. This experiment addresses **RQ2**.

Results. We can answer **RQ1** positively: we are able to model the stages and lifecycles needed for the GreenhouseDT, a digital twin exemplar for a smart greenhouse, and adapt this digital twin to changes in the NVDI values. We did not need to model the transitions between the stages and interactions between the lifecycles, as the checks for compatibility and the conditions for stages to form a lifecycles are sufficient.

Figure 8 shows results for **EX2**. The left figure shows the time (in s) needed for semantic stages (blue) and non-semantic stages (red) need for one self-adaptation cycle for stages $n \in \{1, 10, 20, 30, 40, 50\}$ and assets $m \in \{1, 2000, 4000, 6000, 8000, 10000, 12000\}$ (intermediate values interpolated for better visibility). The results show that the solution using semantic technologies scales better, both in the number of assets and in the number of stages. The plots to the right show the behavior for a fixed number of stages ($n = 10$); we can see that the overhead is only relevant for a low number of assets ($m < 5000$). The lower plot shows that for a fixed number of assets ($m = 14000$), semantic stages scale strictly better. To answer **RQ2**, the overhead of using semantic stages improves performance as the number of stages and assets rises.

Threats to Validity. A threat to *internal validity* is that our implementation of non-semantic stages is an ad-hoc implementation using the specific structure of stages in the KB, but it is not specifically optimized for performance. Nevertheless, we consider the comparison to be a realistic, as a highly specialized system would not be able to be reused and would amount to implementing a highly restricted knowledge graph and reasoner. A threat to *external validity* is that **EX1** is performed on a single case study and **EX2** on a synthetic benchmark. To mitigate this thread, we performed both experiments on the basis of a publicly available exemplar for self-adaptation, where reproduction studies and comparisons are possible.

8 DISCUSSION AND RELATED WORK

8.1 Discussion

We briefly discuss how declarative and semantic stages relate to a two-layered self-adaptive robotics architecture using ontologies, and to ontological asset information models.

Metacontrol. Metacontrol is a two-layered self-adaptive framework, which has been used for, e.g., autonomous underwater robots [33] and builds on the TOMASys [8] ontology for autonomous systems [16]. Metacontrol adds a second self-adaptive feedback loop that manipulates and combines controllers to fulfill different combinations of requirements and adapt to unforseen changes in the underlying system. In contrast to our framework, it does not consider lifecycles, but focuses on exchanging the controller according to application-specific KPIs, not arbitrary components of the inner self-adaptive system.

Integration with Asset Information Models. Declarative stages need access to the requirements of assets and data streams. This is the information provided by industrial information models such

[5] This straightforward task of matching new with existing components has been omitted from the algorithm.
[6] https://jena.apache.org/

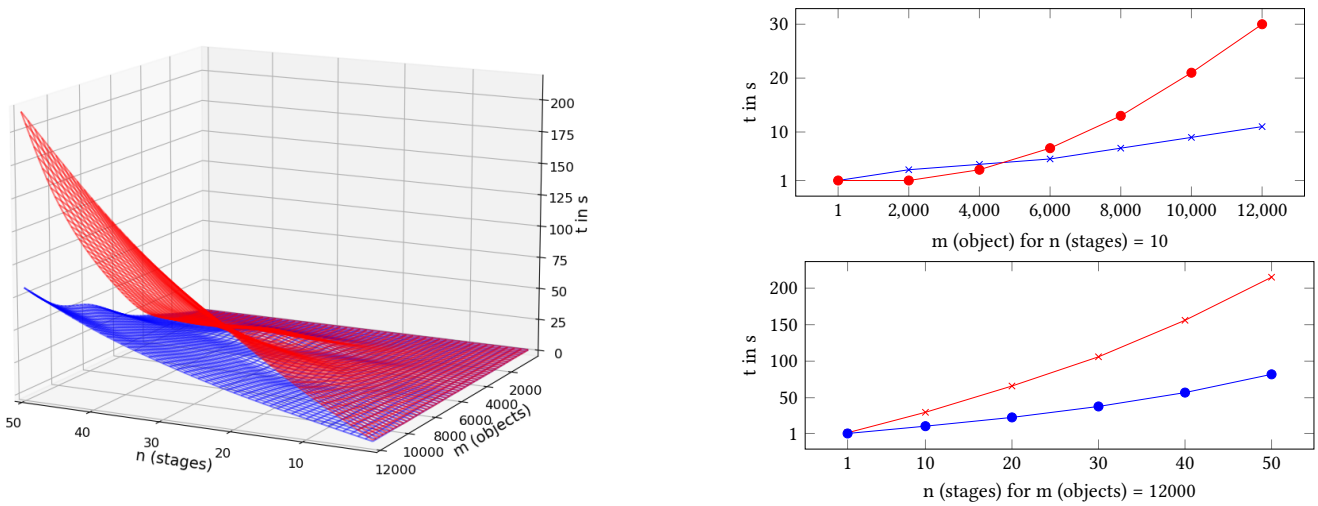

Figure 8: Evaluation Results for n stages and m assets, with time t measured in seconds s.

as the industrial data ontology (ISO 23726) or the integrated asset planning lifecycle ontology (ISO 15926-13), or wrappers on existing standards, such as the asset administration shell [2]. With the increasing use of such ontologies, an additional advantage of semantic stages becomes apparent: they do not require to start modeling lifecycles from scratch, but can build on data and concepts from these semantic frameworks.

8.2 Related Work

We position our work with respect to related work on self-adaptive systems and on digital twin architectures.

Goldsby *et al.* [14] organize self-adaptive systems as collections of steady-state systems. Only one steady-state system can be active at any time and adaptations are dynamic transitions from one steady-state system to another. This way, steady-state systems have similarities to our work on stages of physical systems, and the levels of RE (requirements engineering) identified in [14], including requirements for the runtime monitoring infrastructure and decision-making mechanism, complements our work on lifecycle stages. Whereas prior work on structural self-adaptation in digital twins [20] targets the relation between assets, our focus on lifecycle management and declarative stages is novel. In contrast to prior approaches, we represent asset information by modeling the digital twin's architectural configuration directly in the knowledge base, in a form of runtime models [3, 4, 5] to support runtime analysis.

MAPE-K loops have previously been used in a digital twin context. Feng *et al.* [12] integrated a MAPE-K loop in a digital twin for to provide self-adaptive decisions in a CPS case study. Flammini *et al.* [13], used a MAPE-K loop to perform digital twin functionalities such as behavior modeling and real-time data monitoring to support anomaly detection, using Conformance Checking (CC) and supervised Machine Learning using CC diagnoses. Other MAPE-K loop-based digital twin architectures exist [10, 30, 35, 19], but do not tackle the problem of architectural self-adaptation of the digital twins for lifecycle management addressed in our work.

Semantic technologies, including ontologies and knowledge graphs, have been recognized as crucial for information and data integration in digital twins [22, 40]. Sahlab *et al.* [31] use knowledge graphs to configure digital twins at design time, Li *et al.* [27] use ontologies to detect errors in simulator configurations, and Kiritsis *et al.* [28] use ontologies for data exchange and component configuration in digital twin platforms. Compared to our work, these approaches do not consider self-adaptation or (re)configuration at runtime and do not use semantic technology to model lifecycles. The digital twins of Abburu *et al.* [1] use knowledge graphs to adapt to unforeseen situations on the level of behavioral self-adaptation, i.e., the structure remains unchanged and is not relative to lifecycles.

9 CONCLUSION

It is highly challenging for digital twins to reflect the lifecycle evolution of physical assets, especially if an asset has multiple lifecycles for different aspects or parts. This paper provides a system to manage such lifecycle evolution by means of *architectural self-adaptation*, using declarative stages that do not require explicit modeling of the transitions between stages in the lifecycles. We further propose an implementation of declarative stages using semantic technologies and evaluate our approach in the context of a smart greenhouse digital twin.

In future work, we plan to consider dynamically changing declarative groups, i.e., allowing groups and requirements to evolve in response to, e.g., changing regulations, as well as hierarchical assets and groups that describe requirements for sets of assets by the further exploitation of runtime models.

10 ACKNOWLEDGMENTS

The work was partly supported by the EU project SM4RTENANCE (grant 101123423). We thank Leif Harald Karlsen for his help with the OTTR templates.

References

[1] S. Abburu, A. J. Berre, M. Jacoby, D. Roman, L. Stojanovic, and N. Stojanovic, "COGNITWIN - hybrid and cognitive digital twins for the process industry," in *ICE/ITMC*. IEEE, 2020, pp. 1–8.

[2] S. R. Bader and M. Maleshkova, "The semantic asset administration shell," in *SEMANTiCS*, ser. Lecture Notes in Computer Science, vol. 11702. Springer, 2019, pp. 159–174.

[3] N. Bencomo and L. H. Garcia Paucar, "RaM: Causally-connected and requirements-aware runtime models using Bayesian learning," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 2019, pp. 216–226.

[4] N. Bencomo, S. Götz, and H. Song, "Models@run.time: a guided tour of the state of the art and research challenges," *Softw. Syst. Model.*, vol. 18, no. 5, pp. 3049–3082, 2019. [Online]. Available: https://doi.org/10.1007/s10270-018-00712-x

[5] G. Blair, N. Bencomo, and R. B. France, "Models@ run. time," *Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[6] V. A. Braberman, N. D'Ippolito, J. Kramer, D. Sykes, and S. Uchitel, "MORPH: a reference architecture for configuration and behaviour self-adaptation," in *Proc. 1st International Workshop on Control Theory for Software Engineering (CTSE@FSE 2015)*, A. Filieri and M. Maggio, Eds. ACM, 2015, pp. 9–16. [Online]. Available: https://doi.org/10.1145/2804337.2804339

[7] Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. M. Kienle, M. Litoiu, H. A. Müller, M. Pezzè, and M. Shaw, "Engineering self-adaptive systems through feedback loops," in *Software Engineering for Self-Adaptive Systems*, ser. Lecture Notes in Computer Science, B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds., vol. 5525. Springer, 2009, pp. 48–70. [Online]. Available: https://doi.org/10.1007/978-3-642-02161-9_3

[8] C. H. Corbato, "Model-based self-awareness patterns for autonomy," Ph.D. dissertation, Universidad Politécnica de Madrid, 2013.

[9] M. Dalibor, N. Jansen, B. Rumpe, D. Schmalzing, L. Wachtmeister, M. Wimmer, and A. Wortmann, "A cross-domain systematic mapping study on software engineering for digital twins," *J. Syst. Softw.*, vol. 193, p. 111361, 2022.

[10] F. Edrisi, D. Perez-Palacin, M. Caporuscio, and S. Giussani, "Adaptive controllers and digital twin for self-adaptive robotic manipulators," in *SEAMS*. IEEE, 2023, pp. 56–67.

[11] R. Eramo, F. Bordeleau, B. Combemale, M. van Den Brand, M. Wimmer, and A. Wortmann, "Conceptualizing digital twins," *IEEE Software*, vol. 39, no. 2, pp. 39–46, 2021.

[12] H. Feng, C. Gomes, S. Gil, P. H. Mikkelsen, D. Tola, P. G. Larsen, and M. Sandberg, "Integration of the MAPE-K loop in digital twins," in *ANNSIM*. IEEE, 2022, pp. 102–113.

[13] F. Flammini, "Digital twins as run-time predictive models for the resilience of cyber-physical systems: a conceptual framework," *Philosophical Transactions of the Royal Society A*, vol. 379, no. 2207, p. 20200369, 2021.

[14] H. J. Goldsby, P. Sawyer, N. Bencomo, B. H. Cheng, and D. Hughes, "Goal-based modeling of dynamically adaptive system requirements," in *15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ecbs 2008)*, 2008, pp. 36–45.

[15] B. C. Grau, I. Horrocks, B. Motik, B. Parsia, P. F. Patel-Schneider, and U. Sattler, "OWL 2: The next step for OWL," *J. Web Semant.*, vol. 6, no. 4, pp. 309–322, 2008.

[16] C. Hernández, J. Bermejo-Alonso, and R. Sanz, "A self-adaptation framework based on functional knowledge for augmented autonomy in robots," *Integr. Comput. Aided Eng.*, vol. 25, no. 2, pp. 157–172, 2018.

[17] A. Hogan, E. Blomqvist, M. Cochez, C. d'Amato, G. de Melo, C. Gutierrez, S. Kirrane, J. E. L. Gayo, R. Navigli, S. Neumaier, A. N. Ngomo, A. Polleres, S. M. Rashid, A. Rula, L. Schmelzeisen, J. F. Sequeda, S. Staab, and A. Zimmermann, "Knowledge graphs," *ACM Comput. Surv.*, vol. 54, no. 4, pp. 71:1–71:37, 2022.

[18] M. Horridge, N. Drummond, J. Goodwin, A. L. Rector, R. Stevens, and H. Wang, "The Manchester OWL syntax," in *OWLED*, ser. CEUR Workshop Proceedings, vol. 216. CEUR-WS.org, 2006.

[19] E. Kamburjan, C. C. Din, R. Schlatte, S. L. Tapia Tarifa, and E. B. Johnsen, "Twinning-by-construction: ensuring correctness for self-adaptive digital twins," in *ISoLA*. Springer, 2022, pp. 188–204.

[20] E. Kamburjan, V. N. Klungre, R. Schlatte, S. L. Tapia Tarifa, D. Cameron, and E. B. Johnsen, "Digital twin reconfiguration using asset models," in *ISoLA*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds., vol. 13704. Springer, 2022, pp. 71–88. [Online]. Available: https://doi.org/10.1007/978-3-031-19762-8_6

[21] E. Kamburjan, R. Sieve, C. P. Baramashetru, M. Amato, G. Barmina, E. Occhipinti, and E. B. Johnsen, "GreenhouseDT: An exemplar for digital twins," in *Proc. 19th Intl. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2024)*, L. Baresi, X. Ma, and L. Pasquale, Eds. ACM, 2024, pp. 175–181. [Online]. Available: https://doi.org/10.1145/3643915.3644108

[22] E. Karabulut, S. F. Pileggi, P. Groth, and V. Degeler, "Ontologies in digital twins: A systematic literature review," *Future Generation Computer Systems*, vol. 153, pp. 442–456, 2024.

[23] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003. [Online]. Available: https://doi.org/10.1109/MC.2003.1160055

[24] R. Klinkenberg, "Learning drifting concepts: Example selection vs. example weighting," *Intell. Data Anal.*, vol. 8, no. 3, pp. 281–300, 2004. [Online]. Available: http://content.iospress.com/articles/intelligent-data-analysis/ida00170

[25] P. Koopmann, "Signature-based abduction with fresh individuals and complex concepts for description logics," in *Proc. Thirtieth International Joint Conference on Artificial Intelligence (IJCAI 2021)*, Z. Zhou, Ed. ijcai.org, 2021, pp. 1929–1935. [Online]. Available: https://doi.org/10.24963/ijcai.2021/266

[26] F. J. Kriegler, W. A. Malila, R. F. Nalepka, and W. Richardson, "Preprocessing Transformations and Their Effects on Multispectral Recognition," in *Remote Sensing of Environment, VI*. University of Michigan, Jan. 1969, p. 97.

[27] Y. Li, J. Chen, Z. Hu, H. Zhang, J. Lu, and D. Kiritsis, "Co-simulation of complex engineered systems enabled by a cognitive twin architecture," *International Journal of Production Research*, vol. 60, no. 24, pp. 7588–7609, 2022.

[28] J. Lu, X. Zheng, A. Gharaei, K. Kalaboukas, and D. Kiritsis, "Cognitive twins for supporting decision-makings of internet of things systems," *CoRR*, vol. abs/1912.08547, 2019.

[29] J. Pfeiffer, D. Lehner, A. Wortmann, and M. Wimmer, "Towards a product line architecture for digital twins," in *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2023, pp. 187–190.

[30] P. Pileggi, E. Lazovik, J. Broekhuijsen, M. Borth, and J. Verriet, "Lifecycle governance for effective digital twins: A joint systems engineering and IT perspective," in *2020 IEEE International Systems Conference (SysCon)*. IEEE, 2020, pp. 1–8.

[31] N. Sahlab, S. Kamm, T. Müller, N. Jazdi, and M. Weyrich, "Knowledge graphs as enhancers of intelligent digital twins," in *ICPS*. IEEE, 2021, pp. 19–24.

[32] U. Sattler, T. Schneider, and M. Zakharyaschev, "Which kind of module should I extract?" in *Description Logics*, ser. CEUR, vol. 477, 2009.

[33] G. R. Silva, J. Päßler, J. Zwanepol, E. Alberts, S. L. Tapia Tarifa, I. Gerostathopoulos, E. B. Johnsen, and C. H. Corbato, "SUAVE: an exemplar for self-adaptive underwater vehicles," in *SEAMS*. IEEE, 2023, pp. 181–187. [Online]. Available: https://doi.org/10.1109/SEAMS59076.2023.00031

[34] M. G. Skjæveland, D. P. Lupp, L. H. Karlsen, and H. Forssell, "Practical ontology pattern instantiation, discovery, and maintenance with reasonable ontology templates," in *ISWC (1)*, ser. Lecture Notes in Computer Science, vol. 11136. Springer, 2018, pp. 477–494.

[35] A.-K. Splettstößer, C. Ellwein, and A. Wortmann, "Self-adaptive digital twin reference architecture to improve process quality," *Procedia CIRP*, vol. 119, pp. 867–872, 2023.

[36] R. Torres, N. Bencomo, and H. Astudillo, "Addressing the QoS drift in specification models of self-adaptive service-based systems," in *Proc. 2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE 2013)*, 2013, pp. 28–34.

[37] W3C, SHACL Working Group, "Shapes constraint language," https://www.w3.org/TR/shacl/.

[38] D. Weyns, *An introduction to self-adaptive systems: A contemporary software engineering perspective*. John Wiley & Sons, 2020.

[39] G. Widmer and M. Kubat, "Learning in the presence of concept drift and hidden contexts," *Machine Learning*, vol. 3, p. 69–101, 1996. [Online]. Available: https://doi.org/10.1023/A:1018046501280

[40] X. Zheng, J. Lu, and D. Kiritsis, "The emergence of cognitive digital twin: vision, challenges and opportunities," *Int. J. Prod. Res.*, vol. 60, no. 24, pp. 7610–7632, 2022.