Declarative Lifecycle Management for Self-Adaptive Systems

Eduard Kamburjan[1,2*], Nelly Bencomo[3*], Einar Broch Johnsen[2*] and Silvia Lizeth Tapia Tarifa[2*]

[1]IT University of Copenhagen, Copenhagen, Denmark.
[2]Department of Informatics, University of Oslo, Oslo, Norway.
[3]Department of Computer Science, Durham University, Durham, UK.

*Corresponding author(s). E-mail(s): eduard.kamburjan@itu.dk; nelly.bencomo@durham.ac.uk; einarj@ifi.uio.no; sltarifa@ifi.uio.no;

Abstract

Self-adaptive systems can be realised as layered systems with a feedback loop: a managing system monitors a managed system, updates an internal model, and adjusts the managed system by means of controllers to maintain given requirements. For example, a digital twin coupled with its physical twin constitute such a self-adaptive system. As the managed system shifts between different stages in its lifecycle, these requirements, as well as the associated analysers and controllers, may need to change. The exact triggers for such shifts in a managed system are often hard to predict: they may be difficult to describe or even unknown. However, the shifts can generally be observed once they have occurred, in terms of changes in the system behaviour. This paper proposes an automated method for self-adaptation in self-adaptive systems to address shifts between lifecycle stages in a managed system. Our method is based on declarative descriptions of lifecycle stages for assets in a managed system and their associated counterparts in the managing system. Declarative lifecycle management provides a high-level, flexible method of self-adaptation for self-adaptive systems to reflect disruptive shifts between stages in a managed system.

Keywords: self-adaptation, autonomous systems, software architectures, lifecycle management, digital twins

1 Introduction

This paper presents a declarative method to specify the adaptation logic for autonomous, self-adaptive systems (SASs) in terms of lifecycle changes. A lifecycle change affects the overall system requirements that the SAS aims to maintain or how the system aims to maintain these requirements; i.e., lifecycle changes affect the adaptation logic of the SAS. The triggers for these lifecycle changes may be unknown, or difficult to specify. A declarative approach shifts the focus of self-adaptation from how the environment changes to how to react when the environment has changed, enabling a higher level of abstraction [1] in the description of the adaptation logic. The paper further provides a self-adaptive architecture that enables changes to the adaptation logic of the SAS and an automated method for declarative lifecycle management that handles such changes.

Autonomous systems are subject to different forms of uncertainty, such as changes in system requirements, evolution or failures of the system's assets (i.e., the system's components of interest, which may be physical as well as in software), or unexpected changes in their environment. These

uncertainties can be addressed by giving the autonomous system self-adaptive capabilities. *Self-adaptive systems (SASs)* form a rapidly growing field of research [2–4], with an increasing industrial penetration [5], ranging from e.g., IoT systems, robotic systems and, more recently, digital twins.

A SAS can be organised as a layered system with a feedback loop between a managing and managed system, where that managed system is realising the application logic, and a managing system is realising the adaptation logic [4]. The feedback loop of the SAS enables adjustments to fine-tune the controllers for the managed system, thereby realising *behavioural self-adaptation* [6]. Thus, the managing system of the SAS can adapt its managed system to changes in the environment or in the managed system itself, to ensure that overall system requirements are still met.

However, disruptive shifts in the behaviour of the managed system, its environment or in the associated overall system requirements, may invalidate not only configurations of the managed system but also their associated components in the managing system, triggering the need for so-called *architectural self-adaptation*. These components may include requirement analysers and controllers used by the feedback loop to realise the adaptation logic. Whereas *behavioural self-adaptation* amounts to adjusting the behaviour of the managed system, *architectural self-adaptation* corresponds to reconfiguring the managing system itself, to reflect changes in the structure and requirements of the managed system.

These disruptive shifts can often be understood as transitions between different stages in the lifecycle of the managed system. For example, (i) a data warehouse under a security attack may shift its overall system requirements from making its data easily available to making them inaccessible; (ii) physical assets in engineering pass through stages, from design or commissioning to decommissioning (via construction and operation) that need to be matched in the digital twin; (iii) 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; and (iv) 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.

Uncertainties in the transitions between different lifecycle stages of the managed system may be hard or even impossible to predict accurately by the managing system at runtime. In fact, the triggers for the transitions between different stages of this lifecycle might be unknown; e.g., a failure that causes degradation in the behaviour of a physical component. The correct configuration of the managing system depends on the current components of the managed system and their lifecycle stages.

This paper proposes a two-layered self-adaptive architecture (depicted in Fig. 1) that enables a SAS to adapt to lifecycles changes in the components of a managed system. The novel notion of *stage management* is used for architectural self-adaptation of the SAS to reconfigure the adaptation logic of the managing system, thereby reflecting lifecycle changes in the components of the managed system. Thus, our work separates behavioural (e.g., adjusting existing controllers to better maintain a desired policy) and architectural self-adaptation (e.g., the coordinated addition of new controllers to the configuration in Layer 1), as advocated in, e.g., the MORPH reference architecture [6].

In this paper, we consider how declarative stage management can address architectural self-adaptation and the corresponding notion of *architectural coherence* between the managing and managed systems of the SAS. Together, architectural self-adaptation and architectural coherence provide the means to resolve uncertainties by answering important questions about the SAS; e.g., is the behavioural feedback loop of the managing system currently using the correct components? Does the *analyser* component correctly evaluate the current requirements? How well do the controllers that manipulate the assets of the managed system (in the *executor* component) reflect the current control policy?

Whereas behavioural self-adaptation can be realised by a MAPE-K feedback loop [2, 7] the proposed architectural self-adaptation will be realised by a second MAPE-K feedback loop (depicted in blue in Fig. 1). In this additional feedback loop, the analyser is concerned with the coherence of the SAS with respect to the managed system and the execute component is concerned with repair functions over the components of the managing system of the first MAPE-K feedback loop; i.e.,

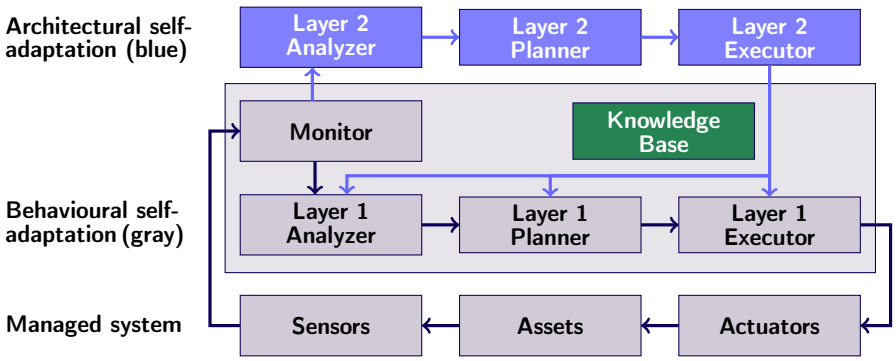

Figure 1: A two-layered self-adaptive architecture with MAPE-K feedback loops for behavioural (Layer 1, grey) and architectural (Layer 2, blue) self-adaptation.

architectural self-adaptation consists of reconfiguring or replacing the Layer 1 components. We need to *express* the relation between the assets (i.e., the components of the managed system) and the Layer 1 components at any point in its lifecycle, *detect* the stages of assets in their lifecycles and *adapt* to stage changes by reconfiguring the Layer 1 components.

The conditions that trigger transitions between lifecycle stages represent uncertainties in the SAS: the conditions that trigger these transitions need not be known. Instead, we specify conditions that express when an asset in the SAS should be considered to be in certain stages, and use stage analysers to detect these conditions. Compatibility conditions between stages are identified to ensure that self-adaptation does not fail. In our architecture, the knowledge base is used to concretely represent the specifications of the declarative lifecycle stages. To provide reasoning and querying capabilities for stage management, we consider the use of knowledge graphs in the knowledge base of our architecture.

Contributions

In short, the main contributions of this paper are:

1. A *two-layered self-adaptive architecture for SAS*, in which a feedback loop for architectural self-adaptation is used to reconfigure the SAS's behavioural feedback loop;
2. *Declarative lifecycle stages*: a design method for modelling change in terms of the lifecycle stages of the system's assets, and its formalisation as a semantic model that can be used in knowledge bases;
3. *Compositional, multi-asset lifecycle stages*: a generalisation of declarative lifecycle stages in which the changes to the adaptation logic of the SAS can depend on the lifecycle changes of multiple assets in the managed system, and that enables compositionality and reuse of stage specifications; and
4. *Lifecycle stage management*: an automated method for architectural self-adaptation that selects the appropriate configuration of a SAS, based on the modelled declarative stages.

We also provide a prototype implementation of the self-adaptive architecture, based on the GreenhouseDT digital twin exemplar [8],[1] in which lifecycle stages are realised as *semantic stages* in a knowledge graph [9], and an *evaluation* to assess the overhead when using semantic technologies to implement declarative stages over a time-consuming and error-prone manual implementation by answering the following research questions:

RQ1: *Can declarative stages be used to model an existing SAS?*

RQ2: *Can multi-asset stages be used to model mutually dependent assets and their lifecycles in a SAS?*

RQ3: *How does using semantic stages over non-semantic declarative stages affect performance?*

This paper extends a paper that was originally presented at *EDTconf 2024* [10]. Building

[1] All auxiliary material is available as a Zenodo artefact: https://doi.org/10.5281/zenodo.18034484.

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

upon the results in [10], we here generalise declarative lifecycle management from an adaptation logic that is triggered by changes to single assets to an adaptation logic that can be triggered by multiple assets and that supports compositionality (Sections 5 and 6.1). We further broaden the scope of discussion from digital twins to a broader context of SASs based on external self-adaptation (Sections 1 and 7) and extended the implementation, evaluation and related work.

Overview

This paper is structured as follows. We present a motivating example in Sect. 2 and briefly introduce the state-of-the-art and preliminaries of digital twins, semantic technologies and asset models in Sect. 3. In Sect. 4 we explain *declarative stages* and in Sect. 5 their compositional extension to handle multiple assets. In Sect. 6 we describe their formalisation using semantic technologies. We detail a digital twin architecture with self-adaptation based on declarative stages in Sect. 7 and evaluate it in Sect. 8. We conclude in Sect. 10.

2 Motivating Example

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

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 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 behaviour will need to adapt to combinations of stages from the different lifecycles.

3 Background

3.1 Self-Adaptation

We now introduce self-adaptive systems, in particular feedback loops and digital twins. Digital twins (DTs) are a natural setting for adapting the configuration of a cyber-physical system, even though their exact definition remains elusive [12, 13]. Remark that an automatic behavioural control loop, a common aspect of DT definitions (e.g., according to Kritzinger *et al.* [14]), is not necessary for the framework considered here. We target general self-adaptive systems and use DTs as their most modern manifestation.

[2]Normalised difference vegetation index (NVDI) is a measure that can be used for the health analysis of vegetation [11].

3.1.1 Preliminaries

A SAS with external self-adaptation separates the application logic, the so-called *managed system* and the adaptation logic, the so-called *managing system* [4].

The MAPE-K feedback loop [2, 4, 7] has been used to underpin properties of SAS and will form the basis for our framework. Behavioural self-adaptation is realised in terms of a MAPE-K feedback loop with the following components: *monitor* components that collect streams of observations from the assets of the physical twin and updates the knowledge model of these assets, *analyser* components that analyse these streams with respect to a set of requirements and triggers the need of behavioural self-adaptation when these requirements are no longer met, *planner* components that determine which adaptation actions are needed to meet the requirements, and *executor* components that execute the changes in the behaviour of the *controller* that control the managed system via given actuators. SASs can be model-centric, as these components interact with a *knowledge base* that maintains a model of the physical system. behavioural self-adaptation corresponds to the Layer 1 feedback loop depicted in grey in Fig. 1.

Digital Twins

Our work presented here uses the MAPE-K loop for DTs, for which MAPE-K loops have been explored previously [15–20]. DTs can be used to illustrate the central concepts of SASs with an external layer for behavioural self-adaptation. DTs are dynamically updated with live data from the twinned system, have analytic capability, and inform decisions that realise value [21]. DTs are typically found and used in engineering disciplines, and they are commonly associated with cyber-physical systems (CPS), where the physical part, the so-called *physical twin*, consists of different physical assets, and the digital part, the *DT* itself, consists of *digital* components (hereafter, DT components), that capture the structure, context, and behaviour of the system they are twinning. The DT needs to be kept in sync with the twinned physical system to avoid model drift [22–24].

From the perspective of self-adaptation, a DT can be seen as a managing system and the physical twin as the managed system. The self-adaptive capabilities requires to monitor, analyse and control the behaviour of the physical assets [25]. Self-adaptation happens in a feedback loop while the digital and physical twins interact: observations flow from the physical to the DT and trigger self-adaptive decisions that flow back from the digital to the physical twin.

The physical twin forms the managed system of behavioural self-adaptation. Behavioural self-adaptation is concerned with adjusting the behaviour of the assets in the physical twin, based on the analysis of a model of these assets. We assume that a physical twin consists of a number of assets, and sensors and actuators associated to these assets.

In contrast, *architectural self-adaptation* adapts the structure of the system to react to unforeseen circumstances and aims to maintain architectural coherence, i.e., it is concerned with the consistency of the components of DTs. Architectural self-adaptation and behavioural self-adaptation interact if the components of behavioural self-adaptation must be considered in the consistency notion that is used by architectural self-adaptation. Architectural self-adaptation is needed to, e.g., handle different lifecycle stages in the underlying physical system: Each lifecycle stage may have its own requirements on the behavioural self-adaptation components.

In the remainder of this section, we position our work in the context of related research on self-adaptive systems, with a particular focus on self-adaptive DT architectures, which have inspired both our approach and the motivating example.

3.1.2 State-of-the-Art: Structured Self-Adaptation

Behavioural and architectural self-adaptation can be understood as two different, but interlinked feedback loops. In the case of our two-layered architecture, cf. Fig. 1, the two loops are stacked on top as layers and target lifecycle stages, but other approaches for linking two self-adaptation mechanisms have been proposed. In particular in robotics, two-layered self-adaptive frameworks

have been explored to enable autonomous control systems to resolve uncertainties from the environment [26].

Metacontrol [27], used in applications such as autonomous underwater robots [28], builds on the TOMASys [29] ontology for autonomous systems [30]. Metacontrol introduces a second self-adaptive feedback loop that manipulates and combines controllers to meet different combinations of requirements and adapt to unforeseen changes in the underlying system. However, the different controllers are not related to each other; their selection by the second self-adaptation layer is based purely on the state of the environment. Thus, instead of addressing lifecycle management, Metacontrol focuses on exchanging controllers based on application-specific KPIs rather than arbitrary components of the inner self-adaptive system.

The Reconfiguration Framework for distributed Embedded systems for Software and Hardware (ReFrESH) [31] is a layered framework that aims to increase fault tolerance in robots. It allows reconfiguring, i.e. exchange software components, if a fault is detected, and the component is needed for the current task. These software components can also be controllers, i.e., a lower layer of self-adaptation. ReFrESH is restricted to robotic software and does not consider relations between possible components, thus not providing support for modelling.

Architecture-based self-adaptation is a well-established approach that has been explored across various application domains. It focuses on adapting the *software architecture* of a system, using an architectural model [26]. This is closely related to our notion of structure. However, structure for us includes not only the software architecture, but also the additional modelling of lifecycles, domain knowledge and asset information. In particular, while we reconfigure the software of the managed system, we also consider the non-software parts of the overall system, i.e., the physical twin.

Our modelling of the relation between possible software components addresses a major shortcoming in the current architecture-based self-adaptation: The mechanisms are rarely reusable, and focus only on functionality (i.e., configuring to fulfil a task) [32]. Our two-layered architecture furthermore targets a self-adaptive system as its adaptation target, not a specific robotic software architecture. The MORPH reference architecture [6] is more fine-grained to achieve reuse, distinguishing between these architectural adaptations because of degradation of the system and those because of environmental changes.

Architectures that build on MAPE-K can structure or distribute MAPE-K loops in different ways, depending on how the different MAPE components are organised. Interestingly the local and global knowledge in these patterns is orthogonal to the decentralisation, since knowledge can be (partially) shared between the components, following its own hierarchical or distributed pattern. In particular, hierarchical MAPE-K patterns have a long history for self-adaptive software systems [33, 34], although managing and designing the hierarchy can be a challenge [35, 36]. Our approach fits within the general concept of hierarchical MAPE-K loops, and focuses on replacing components in the lower layer according to its detected lifecycle stage. In our work, declarative modelling is introduced to address the design challenge.

Recently, in the specific area of autonomous robotics, Alberts *et al.* [26] presented a systematic mapping of robotics software architecture-based self-adaptive systems, highlighting the robotics community's interest in addressing both behavioural and architectural self-adaptation using external feedback loops. Similarly, in IoT systems, characterised by software assets in the managed system, architectural self-adaptation has also gained significant attention. For example, Weyns *et al.* [37] proposed the MARTAS framework, which combines formal models with statistical techniques at runtime to optimise configurations of the managed system, while Alfonso *et al.* [38] introduced a domain-specific language for modelling the architectures of IoT systems.

An orthogonal perspective on two-layered adaptation can be found in business process modelling (BPM), with the following notions of adaptation [39–41]: Process-level and instance-level adaptation. Instance-level adaptation happens on a process instance, while process-level adaptation is concerned with adapting the abstract process model. While neither is self-adaptation in the sense of fully automatic, closed loops that react to unexpected changes, they are semi-automatic closed loops for a similar purpose. However, the implementation and modelling mechanics are different, and humans have to interact with the

adaptation, especially at the process-level [41]. Most work in DTs, including ours, focuses on instance-level self-adaptation.

3.1.3 State-of-the-Art: Self-Adaptation for DTs and Lifecycles

Architectural self-adaptation has been considered for DTs [42–44] and targets the relationships between assets using a simple notion of consistency, but does not consider lifecycles or behavioural self-adaptation at a lower layer. This line of work resembles the architectural approaches to robotics software engineering described above. Unlike earlier approaches, we model asset information by directly representing the DT's architectural configuration in the knowledge base, together with lifecycle information.

MAPE-K loops have previously been used in a DT context and examined for models@runtime, an earlier architecture that captures many aspects of modern DTs [45–47]. This runtime model supports dynamic runtime analysis and provides enhanced capabilities for self-adaptation and decision-making. Bibow *et al.* [48] describes a self-adaptive DT that handles changes in the structure of a cyber-physical production system by reacting to events by pre-defined actions that are used in planning. Splettstößer *et al.* [17] extend this system to also consider quality attributes with regards to the sensor data, thus realising self-adaptation with respect to both architecture and behaviour. However, lifecycles of neither the digital nor the physical components are modelled, even with the detection of quality attributes.

For purely behavioural self-adaptation, Feng *et al.* [49] integrated a MAPE-K loop in a DT for to provide self-adaptive decisions in a CPS case study. Similarly, Flammini *et al.* [15], used a MAPE-K loop to perform DT functionalities such as behaviour modelling 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 DT architectures exist [16, 18, 19], but do not tackle the problem of architectural self-adaptation of the DTs for lifecycle management addressed in our work.

Self-adaptation can be seen as the autonomous reestablishment of consistency for a DT. However, investigations into the notion of consistency for DTs so far focus on the relation between models within a DT [50–52], not on its reestablishment.

A substantial body of research examines DT architectures and platforms (see the survey by Lehner *et al.* [53]. Many of these studies focus on components and composition, although the terminology varies. For example, Josifovska *et al.* [54] describe the links between components as interrelations. Lehner *et al.* [55] Lehner *et al.* [55] compose service-oriented DTs using declarative templates. However, none of these works employs declarative modelling for lifecycle management or relates it to self-adaptation.

In the context of digital engineering, or model-based system engineering, DTs either focus on one part of the overall lifecycle, or on data exchange between different stages. While Tong *et al.* [56] propose a framework for co-evolution of architecture and models, their notion of a lifecycle is concerned with the whole cyber-physical system, i.e., design, operations, maintenance etc., and not the lifecycle of components or subsystems during operations. The former is a common view in model-driven engineering [57]. Mertens *et al.* [58] investigate the lifecycle evolution of DTs and how their architecture has to change. However, they do so manually and without a self-adaptation loop.

Esterle *et al.* [59] point out that selecting the correct model, and model parameters, is a challenge when based only on data from the physical system. This is for technological reasons, such as model swapping for simulations, but also to detect whether a model for a certain situation is already available. Modelling the lifecycle of a physical or digital components, as proposed here, addresses this challenge.

The seminal paper by Goldsby *et al.* [60] organises self-adaptive systems as collections of steady-state systems, where only one steady-state system can be active at any time. Adaptations are treated as dynamic transitions from one steady-state system to another. In this approach, steady-state systems bear similarities to our work on the stages (cf. Sect. 4). Additionally, the levels of requirements engineering (RE) identified in [60], including requirements for runtime monitoring infrastructure and decision-making mechanisms, complement our work on lifecycle stages.

Research Gap

To the best of our knowledge, no previous work (i) models a two-layer self-adaptive framework by explicitly analysing the interactions between two hierarchical MAPE-K loops that link behavioural and architectural self-adaptation, i.e., an upper layer that interacts with behavioural self-adaptation, or (ii) couples such a framework with a lifecycle-based consistency criterion.

3.2 Semantic Technologies

3.2.1 Preliminaries

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) [61] and its Manchester syntax [62], which we use for modelling 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 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 DTs applications [63]. 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.[3] 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 (·)? denotes optional elements.

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 D with a domain C and a range T can be expressed by

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

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

A *class* is a named set of individuals. We here describe classes C using axioms of the following form, where C', C'_i are other class names:

Class: C **SubClassOf**: C'
(**EquivalentTo**: Rst)?
(**DisjointUnionOf**: $C'_1,\ldots,C'_n$)?

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 Rst is a conjunction (using the symbol **and**) of class names with restrictions. Let D be a data property, P, P' object properties, n a number literal, I, I_1, ..., I_n individuals, and T a numerical data type. Restrictions take one of the following forms:

- D **some** T[> n]: every individual has at least one triple with predicate D that relates it to an object that is strictly bigger than n.
- D **some** T[$\leq$ n]: every individual has at least one triple with predicate D that relates it to an object that is smaller or equal to n.
- P **some** C: every individual has at least one triple with predicate P that relates it to an object of class C.
- P **min** n P': every individual has at least n triples with predicate P'.
- P **exactly** n P': every individual has exactly n triples with predicate P'.

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

- P **only** $\{\mathsf{I}_1, \ldots, \mathsf{I}_n\}$: all triples with predicate P are related to objects in the set $\{\mathsf{I}_1, \ldots, \mathsf{I}_n\}$.

Additionally, two special forms of axioms are considered. The first expresses that a set of classes is pairwise disjoint. The second expresses the above restrictions for single individuals.

(**DisjointClasses**: $\mathsf{C}'_1, \ldots, \mathsf{C}'_n$)?
{I} **only** Rst .

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($\mathsf{C}_1, \ldots, \mathsf{C}_n$) decides whether the classes $\mathsf{C}_1, \ldots, \mathsf{C}_n$ share individuals.

3.2.2 State-of-the-Art

Semantic technologies, including ontologies and knowledge graphs, have been recognised as crucial for information and data integration, as well as for federation of DTs [43, 63–65], but asset modelling and lifecycle management are so far decoupled and ontological representations are yet to be exploited for architectural self-adaptation. As demonstrated in Sect. 5, knowledge graphs and ontologies meet the technological demands of our two-layer architecture. A knowledge graph serves as an efficient store that supports fast queries to determine an asset's lifecycle stage, while the accompanying ontology enables (a) reasoning about the architectural consistency of the model, (b) precise stage-membership inference, and (c) seamless interoperability with existing industrial information models.

Industrial information models such as the industrial data ontology (ISO 23726) and the integrated asset planning lifecycle ontology (ISO 15926-13), and wrappers to existing standards, such as the asset administration shell [66], complement our work on declarative lifecycles by providing standardised data and concepts that could be integrated in our work as a starting point for modelling lifecycles for physical assets.

Sahlab *et al.* [67] use knowledge graphs to configure DTs at design time, Li *et al.* [68] use ontologies to detect errors in simulator configurations, and Kiritsis *et al.* [69] and Ren *et al.* [70] use ontologies for data exchange and component configuration in DT 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 DTs of Abburu *et al,* [71] and Ghanadbashi *et al.* [72] use knowledge graphs to adapt to unforeseen situations on the level of behavioural self-adaptation, i.e., the structure remains unchanged and is not relative to lifecycles.

Abbasi *et al.* [73] discuss four layers of semantic drift, where different parts of a DT can drift from their representation in the system: data, models, meta-models and ontologies. Our work maps to the two lowest layers: drift in the data and models. Their work does not discuss modelling or reacting to drift, but gives a conceptual framework to structure it.

Feldmann *et al.* propose a classification of different approaches to inter-model inconsistencies [74], where our approach falls under the rule-based category. Some case studies on inter-model consistency in model-based systems engineering are also based on knowledge graphs [75] and there are similar approaches to phrase inconsistencies between models using graphs [76, 77]. The closest work is the original framework from [78], which defines consistency in terms of links between models. However, it lacks the distinction between physical asset and twin component; i.e., it does not distinguish between membership in a stage, and consistency of the layer-1 components with the assets. Thus, no relation to self-adaptation framework like MAPE-K is given.

Addressing both drift and inconsistencies, David *et al.* [79, 80] discuss possible causes for inconsistencies in evolution of DTs. Their taxonomy groups inconsistency causes into model, data, or system evolution and operation causes. As their work is concerned with single models and focuses on behaviour, i.e., layer-1 or behavioural self-adaptation in our terminology, declarative stages (cf. Sect. 4) do not directly map into their taxonomy. Their work also does not discuss how to model or repair inconsistencies.

Our approach is intentionally domain-agnostic; we therefore avoid binding it to any single ontology. Domains differ markedly, e.g., BRICK or BIM-OIM for buildings [81, 82], SAREF

for IoT [83], and a range of industrial asset-information models. Qureshi *et al.* [84] similarly identify core concepts for requirements of self-adaptation and encode them in an ontology. However, hard-coding one ontology would merely shift the alignment burden to practitioners who deploy the framework in a different domain.

Research Gap

Semantic technologies already underpin asset descriptions in many DT solutions, but have so far not been applied to architectural self-adaptation nor to reasoning about relationships between lifecycle stages. Our work addresses both gaps.

4 Declarative Stages

A *declarative stage* provides a description of how a lifecycle stage of a system's assets can be recognised and how the SAS needs to adapt when the system's assets have transitioned into this lifecycle stage, but not why the transition into the stage occurs. For our work presented in this paper, it is key that this design realises a clear separation of concerns between describing lifecycle stages and describing temporal aspects such as the triggering conditions for changes between these stages.

4.1 Self-Adaptation

During the life of the system, different assets may change between different stages in their respective lifecycles, and the SAS will need to adapt *its architectural configuration* accordingly. This adaptation may involve removing components of the managing system that are no longer valid and starting up new components that may now be needed. For simplicity, we will focus the discussion on requirement analysers and controllers in the context of DTs.

A SAS that uses declarative stages must make them operational; i.e., the declarative stages 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 managing system collects streams of observations concerning the system's assets and updates the knowledge model of these assets.
- **Analyser:** For a given declarative stage, all assets at that stage in the managed system are Analysed for architectural coherence; for example, the DT checks that a plant in a sick stage has the correct requirement analyser and controller for that stage.
- **Planner:** For each inconsistent asset, the managing system identifies adaptation actions for the detected inconsistencies; for example, a sick plant might have an associated requirement analyser $m \leq 7$ for soil moisture when it should have a requirement analyser $m \leq 5$ instead. Other components in the managing system are handled analogously (e.g., controllers).
- **Executor:** The managing system executes the required changes, including the initialisation and removal of components, e.g., requirement analysers and controllers.

Already at this point, we can elicit some requirements.

1. The knowledge base needs to not only include information about the asset, but also about the components of the managing system. Additionally, the managing system needs to provide reasoning and query capabilities to deduce the declarative stage to which an asset belongs.
2. We 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 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., $\mathsf{plant1} \in \mathcal{A}_{\mathsf{Plant}}$.

4.2 Declarative Stages

A declarative stage in a lifecycle describes the conditions that determine that an asset is in the stage,

which components the managing system needs for the asset in this stage, e.g., the DT components in the DT that correspond to a sick plant, and how to generate these components when the asset enters this stage in the lifecycle. The central notion we use is a *consistent managing system*: a managing system is consistent if the set of 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 component in the managing system for our purposes:

Definition 2 (Component class)**.** *A* component class $\mathcal{C}$ *is a set of components in the managing system. We assume that components in different component classes are disjoint, i.e.,* $\mathcal{C}_1 \cap \mathcal{C}_2 = \emptyset$.

In our examples, we focus on two kinds of components in the managing system: requirement analysers and controllers. Let us denote by $\mathcal{I}$ the *observational inputs* to the managing system, 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 analysers and controllers)**.** *A* requirement analyser $ranl : \mathcal{I} \to \mathbb{B}$ *is a function from input streams to Booleans, modelling the verdict. A* controller $ctrl : \mathcal{I} \to \mathbb{R}$ *is a function from input streams to a real number that is modelling the control decision.*

The *type* of a declarative stage is defined in terms of a class of assets and the possible component classes of the declarative stage.

Definition 3 (Stage Type)**.** *Let* $\mathcal{A}$ *be an asset class and* $\overline{\mathcal{C}} = \{\mathcal{C}_1, \ldots, \mathcal{C}_n\}$ *a set of component classes. A* stage type $\mathcal{T}$ *is a pair* $\mathcal{T} = \langle \mathcal{A}, \overline{\mathcal{C}} \rangle$.

Example 3 (A stage type for the Greenhouse)**.** *Let* $\mathcal{A}_{\mathsf{Plant}}$ *be an asset class and* Req *the set of all requirement analysers (see Example 2). A stage type for declarative stages for assets in* $\mathcal{A}_{\mathsf{Plant}}$ *is* $\langle \mathcal{A}_{\mathsf{Plant}}, \{\mathsf{Req}\} \rangle$.

We define declarative stages formally as follows. A declarative stage always has a type.

Definition 4 (Declarative stages)**.** *Let* $\mathcal{T} = \langle \mathcal{A}, (\mathcal{C}_i)_{i \in I} \rangle$ *be a stage type and* $\mathcal{C} = \bigcup_i \mathcal{C}_i$ *the union of the contained component classes. A* declarative stage D *of type* $\mathcal{T}$ *is a pair of two sets*

$$\mathsf{D} : \mathcal{T} = \langle member_{\mathsf{D}}, consistent_{\mathsf{D}} \rangle,$$

where

- $member_{\mathsf{D}} \subseteq \mathcal{A}$ *is a set of assets;*
- *the relation* $consistent_{\mathsf{D}} \subseteq member_{\mathsf{D}} \times 2^{\mathcal{C}}$ *describes, for a given asset, all component sets in the managing system with which it is consistent.*

Let us first examine two properties that declarative stages can have. The first expresses that every asset in a group must always have assigned one component in the managing system of a certain component class. This property is used, for example, for actuator assets that always need to have an assigned control component.

We let X range over sets of components in the managing system and say that a declarative stage $\mathsf{D} : \langle \mathcal{A}, \overline{\mathcal{C}} \rangle$ is $\mathcal{C}_i$*-requiring* for some $\mathcal{C}_i \in \overline{\mathcal{C}}$ if

$$\forall a \in \mathcal{A}, X \in 2^{\mathcal{C}}. \\ (a, X) \in consistent_{\mathsf{D}} \to \exists c \in \mathcal{C}_i.\ c \in X\ .$$

This means that a member of $\mathcal{C}_i$ is required for consistency.

The second property expresses that some components are irrelevant for the consistency of an asset. This property models that assigning a class of components to an asset is optional, and changes do not require to recheck consistency. Formally, for a $\mathcal{C}_i$*-invariant* declarative stage, consistency does not depend on members of $\mathcal{C}_i$ and we say that $\mathsf{D} : \langle \mathcal{A}, \overline{\mathcal{C}} \rangle$ is $\mathcal{C}_i$*-invariant* for some $\mathcal{C}_i \in \overline{\mathcal{C}}$ if

$$\forall a \in \mathcal{A}, X \in 2^{\mathcal{C}}. \\ (a, X) \in consistent_{\mathsf{D}} \to \forall c \in \mathcal{C}_i.\ c \in X\ .$$

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

Example 4 (Declarative stages for plants)**.** *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. Their type is* $\langle \mathcal{A}_{\mathsf{Plant}}, \{\mathsf{Req}\} \rangle$ *(see Example 3). The NVDI measurement is used to determine whether a plant a is sick or healthy. The two stages use different moisture settings in the corresponding requirement analysers. Here, we*

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

Figure 3: Declarative stages for Example 4.

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 analyser for $i_{\mathsf{m}}(a) \leq x$. As before, X ranges over sets of DT components.

We can easily see that the two stages of Example 4 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 5 (Lifecycle)**.** *Let $\mathcal{T}$ be a stage type, $\mathcal{A}$ an asset class and I an index set. A* lifecycle $\mathsf{L}_{\mathcal{A}}$ *for $\mathcal{A}$ consists of a set of declarative stages $\left(\mathsf{D}_i\right)_{i\in I}$ such that the following conditions hold:*

- $\forall i.\ \mathsf{D}_i : \mathcal{T}$,
- $\mathcal{A} = \bigcup_{i\in I} member_{\mathsf{D}_i}$ *and*
- $\forall i, j \in I.\ i \neq j \Rightarrow member_{\mathsf{D}_i} \cap member_{\mathsf{D}_j} = \emptyset$.

Example 5 (Declarative stages and lifecycle for pumps)**.** *We consider water pumps for the plants above. Their status is read using i_{status}. This status is either* ok*, if the pump is operating normally, or* mtn*, if its internal logic has determined that it needs to be maintained. A maintained pump operates under stricter requirements; in particular, its energy consumption must stay below a certain level. The $\mathsf{ranl}_{\mathsf{energy}}^{\leq w}$ requirement analyser monitors that a pump uses less than w watts.*

$$
\begin{aligned}
\mathsf{D}_{\mathsf{ok}} &= \left(member_{\mathsf{ok}}, consistent_{\mathsf{ok}}\right)\\
member_{\mathsf{ok}} &= \{a \mid a \in \mathsf{Pump}, i_{\mathsf{status}}(a) = \mathsf{ok}\}\\
consistent_{\mathsf{ok}} &= \left\{(a, Y) \mid \mathsf{ranl}_{\mathsf{energy}}^{\leq 200}(a) \in Y\right\}\\
\mathsf{D}_{\mathsf{mtn}} &= \left(member_{\mathsf{mtn}}, consistent_{\mathsf{mtn}}\right)\\
member_{\mathsf{mtn}} &= \{a \mid a \in \mathsf{Pump}, i_{\mathsf{status}}(a) = \mathsf{mtn}\}\\
consistent_{\mathsf{mtn}} &= \left\{(a, Y) \mid \mathsf{ranl}_{\mathsf{energy}}^{\leq 100}(a) \in Y\right\}
\end{aligned}
$$

Together, these two stages form a lifecycle $\mathsf{L}_{\mathsf{Pump}} = \{\mathsf{D}_{\mathsf{ok}}, \mathsf{D}_{\mathsf{mtn}}\}$.

A lifecycle is *$\mathcal{C}_i$-ensuring* if all its stages are $\mathcal{C}_i$-requiring. A $\mathcal{C}_i$-ensuring lifecycle guarantees that an asset that is consistent with any of its stages, always has a $\mathcal{C}_i$-component assigned to it.

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 DT 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 managing system must exist.

Definition 6 (Compatible stages and lifecycles)**.** *Let $\mathcal{T}_1 = \langle \mathcal{A}, \overline{\mathcal{C}}_1\rangle, \mathcal{T}_2 = \langle \mathcal{A}, \overline{\mathcal{C}}_2\rangle$ be stage types with $\overline{\mathcal{C}}_1 \cap \overline{\mathcal{C}}_2 \neq \emptyset$, and let $\mathsf{D}_1 : \mathcal{T}_1$ and $\mathsf{D}_2 : \mathcal{T}_2$ be declarative 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 $X \subseteq \overline{\mathcal{C}}_1 \cap \overline{\mathcal{C}}_2$ 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.*

Example 6 (Stage compatibility)**.** *Note that stages are trivially compatible if their membership predicates do not overlap. Let us consider a stage for a pump that states that for any pump, there must be a planning component implementing a PDDL interface [85] (here modelled by $\mathsf{PDDL}(a)$):*

$$
\begin{aligned}
\mathsf{D}_{\mathsf{PDDL}} &= \{member_{\mathsf{PDDL}}, consistent_{\mathsf{PDDL}}\}\\
member_{\mathsf{PDDL}} &= \mathsf{Pump}\\
member_{\mathsf{PDDL}} &= \left\{a, Y) \mid a \in \mathsf{Pump}, Y \supseteq \{\mathsf{PDDL}(a)\}\right\}
\end{aligned}
$$

This stage is compatible with both D_{ok} and $\mathsf{D}_{\mathsf{mtn}}$, however the following variant would not be compatible because it prohibits the addition of the requirement analyser DT component needed for the stages from $\mathsf{L}_{\mathsf{Pump}}$:

$member_{\mathsf{PDDL}} = \{(a, Y) \mid a \in \mathsf{Pump}, Y = \{\mathsf{PDDL}(a)\}\}$.

$\langle \mathsf{Plant}, \mathsf{Pump} \rangle = \langle \mathsf{Plant} \rangle \cdot \langle \mathsf{Pump} \rangle$.

5 Multi-Asset Declarative Stages

The previous section introduced *single-asset* declarative stages. These are non-compositional for the following reasons. First, they cannot consider aggregates of assets. For example, they cannot express consistency over a pair of a plant and its pump. Second, two single-asset declarative stages cannot be composed into a new single-asset declarative stage, which is desirable to support reuse of consistency-predicates in aggregates.

In this section, we generalise single-asset declarative stages into multi-asset declarative stages and consider how these can be composed. To this aim, we first generalise asset classes and then stage types, as defined in Def. 3. Let $\langle e_1, \dots e_n \rangle$ denote a sequence with elements $e_1, \dots e_n$ and $s_1 \cdot s_2$ the concatenation of sequences s_1 and s_2. The type of a multi-asset declarative stage is not a single asset class, but a sequence of asset classes, which we denote *asset scheme*:

Definition 7 (Asset Schemes)**.** *An* asset scheme $\mathcal{AS}$ *is a sequence of asset classes. We say that a set of assets* $\{a_1, \dots, a_n\}$ *adheres to* $\mathcal{AS}$*, written* $\{a_1, \dots, a_n\} : \mathcal{AS}$*, if the following holds:*

$$\begin{aligned} \{a_1, \dots, a_n\} : \langle \mathcal{A}_1, \dots, \mathcal{A}_n \rangle \iff \\ \forall i \leq n. \exists j \leq n.\ a_i \in \mathcal{A}_j \\ \wedge \forall i \leq n. \exists j \leq n.\ a_j \in \mathcal{A}_i\ . \end{aligned}$$

The semantics of an asset scheme, denoted $[\![\mathcal{AS}]\!]$, is the set of all asset sequences that adhere to the asset scheme: $[\![\mathcal{AS}]\!] = \{\mathcal{AC} \mid \mathcal{AC} : \mathcal{AS}\}$.

We can illustrate asset schemes by the composition of the asset classes of plants and pumps.

Example 7 (Composition of asset classes)**.** *Let* Plant *be the asset class of all plants and* Pump *the asset class of all pumps, and let* $a \in \mathsf{Plant}, b \in \mathsf{Pump}$ *be assets. The sequence* $\mathcal{AS}_p = \langle \mathsf{Plant}, \mathsf{Pump} \rangle$ *is an asset scheme for pairs of plants and pumps. The sequence* $\langle a, b \rangle$ *is an asset aggregate such that* $\langle a, b \rangle : \mathcal{AS}_p$*. The asset scheme can be decomposed as follows, using concatenation:*

Multi-asset stage types and declarative stages are defined formally by generalising Definitions 3 and 4. The difference to single-asset stage types and declarative stages is that multi-asset stage types and multi-asset declarative classes are defined with respect to asset schemes and not to asset classes.

Definition 8 (Multi-Asset Stage Types)**.** *Let* $\mathcal{AS}$ *be an asset scheme and* $\mathcal{C}_1, \dots, \mathcal{C}_m$ *component classes. A* multi-asset stage type $\mathcal{T}$ *has the form* $\langle \mathcal{AS}, \{\mathcal{C}_1, \dots, \mathcal{C}_m\} \rangle$.

Definition 9 (Multi-Asset Declarative Stages)**.** *Let* $\mathcal{T} = \langle \mathcal{AS}, \{\mathcal{C}_1, \dots, \mathcal{C}_m\} \rangle$ *be a multi-asset stage type and* $\mathcal{C} = \bigcup_i \mathcal{C}_i$ *the union of the component classes in* $\mathcal{T}$*. A* multi-asset declarative stage $\mathsf{D} : \mathcal{T}$ *is a pair of two sets* $(member_{\mathsf{D}}, consistent_{\mathsf{D}})$ *where*

- $member_{\mathsf{D}} \subseteq [\![\mathcal{AS}]\!]$ *is a set of sequences of assets;*
- *the relation* $consistent_{\mathsf{D}} \subseteq member_{\mathsf{D}} \times 2^{\mathcal{C}}$ *describes, for a given sequence of asset, all sets of components in the managing system with which this sequence is consistent.*

We identify the multi-asset stage type of a single asset with its single-asset counterpart; i.e., $\langle \langle \mathcal{A} \rangle, \overline{\mathcal{C}} \rangle = \langle \mathcal{A}, \overline{\mathcal{C}} \rangle$. Analogously, if $|\mathcal{AS}| = 1$ then we identify the single-asset stage type with its multi-asset counterpart.

To illustrate multi-asset declarative stages, consider the composition of the single-asset declarative stages of Examples 4 and 5. We consider the asset aggregate consisting of a plant and a pump that waters it. As before, the plant can be healthy or sick and the pump can be fully operational or under maintenance, in which case it operates under additional requirements. The overall lifecycle will then require four stages. A multi-asset declarative stage describing a system composed of a sick plant and a fully operational pump is as follows.

Example 8 (Multi-asset declarative stages)**.** *The multi-asset declarative stage* $\mathsf{D}_{\mathsf{sick,ok}}$ *models the situation where the plant is sick, and the pump is fully operational. Its asset scheme* $\mathcal{T}_p = \langle \langle \mathsf{Plant}, \mathsf{Pump} \rangle, \{\mathsf{ReqMoisture}, \mathsf{ReqOutput}\} \rangle$ *describes pairs of pumps and plants, which are assigned different requirement monitors.*

$$\mathsf{D}_{\mathsf{sick,ok}} = \big(\mathit{member}_{\mathsf{sick,ok}}, \mathit{consistent}_{\mathsf{sick,ok}}\big) \ ,$$
$$\mathit{member}_{\mathsf{sick,ok}} = \big\{\{a,b\} \mid i_{\mathsf{nvdi}}(a) \leq 0.5, i_{\mathsf{status}}(b) = \mathtt{ok}\big\} \ ,$$
$$\mathit{consistent}_{\mathsf{sick,ok}} = \big\{(\{a,b\}, Y) \mid \{\mathsf{ranl}_m^{\leq 5}(a), \mathsf{ranl}_{\mathsf{energy}}^{\leq 200}(b)\} \subseteq Y\big\} \ .$$

In Example 8, the information from Example 4 has been duplicated. It is also easy to see that the duplication will occur in an eventual stage $\mathsf{D}_{\mathsf{sick,ok}}$ for sick plants and maintained pumps. To avoid duplication, we thus turn our attention to the composition of multi-asset stages.

5.1 Composing Multi-Asset Stages

We consider two notions of composition for multi-asset stages. The first of these is based on directly concatenating two multi-asset stage types, and preserves the declarative stages defined over them.

Definition 10 (Direct composition of multi-asset stage types)**.** *Let $\langle \mathcal{AS}_1, \overline{\mathcal{C}}_1\rangle$ and $\langle \mathcal{AS}_2, \overline{\mathcal{C}}_2\rangle$ be two multi-asset declarative stage types. The direct composition of multi-asset stage types concatenates the asset schemes and considers both sets of component classes:*

$$\langle \mathcal{AS}_1, \overline{\mathcal{C}}_1\rangle \cdot \langle \mathcal{AS}_2, \overline{\mathcal{C}}_2\rangle = \big\langle \mathcal{AS}_1 \cdot \mathcal{AS}_2, \overline{\mathcal{C}}_1 \cup \overline{\mathcal{C}}_2\big\rangle \ .$$

The intuition behind Def. 10 is that the first assets selected with the membership predicate of the multi-asset stage type, must be consistent with the first stage type and correspondingly the assets of the second stage type with its consistency predicate.

Definition 11 (Direct composition of multi--asset declarative stages)**.** *Let $\langle \mathcal{AS}_1, \overline{\mathcal{C}}_1\rangle$ and $\langle \mathcal{AS}_2, \overline{\mathcal{C}}_2\rangle$ be multi-asset declarative stage types, and $\mathsf{D}_1 \colon \langle \mathcal{AS}_1, \overline{\mathcal{C}}_1\rangle$ and $\mathsf{D}_2 \colon \langle \mathcal{AS}_2, \overline{\mathcal{C}}_2\rangle$ multi-asset declarative stages. The composition $\mathsf{D}_1 \cdot \mathsf{D}_2$ is defined by*

$$\begin{aligned}&\big(\mathsf{D}_1 \colon \langle \mathcal{AS}_1, \overline{\mathcal{C}}_1\rangle\big) \cdot \big(\mathsf{D}_2 \colon \langle \mathcal{AS}_2, \overline{\mathcal{C}}_2\rangle\big)\\ &\quad = \big(\mathsf{D}_1 \cdot \mathsf{D}_2\big) \colon \big(\langle \mathcal{AS}_1, \overline{\mathcal{C}}_1\rangle \cdot \langle \mathcal{AS}_2, \overline{\mathcal{C}}_2\rangle\big)\\ &\quad = \big(\mathit{member}_{\mathsf{D}_1\mathsf{D}_2}, \mathit{consistent}_{\mathsf{D}_1\mathsf{D}_2}\big) \ ,\end{aligned}$$

where the two predicates are defined in Fig. 4a.

Example 9. *We can use direct composition to compose the stage introduced in Example 8 with $\mathsf{D}_{\mathsf{sick,ok}} = \mathsf{D}_{\mathsf{sick}} \cdot \mathsf{D}_{\mathsf{ok}}$.*

Observe that direct composition does not allow the consistency in the composed stage to be constrained based on interactions between two assets. For example, consider the case where we want to refine the declarative stage for a sick plants with an operational pump into two substages: If the plant is very sick, a more aggressive controller is required, while if it is barely sick, then a less aggressive controller will suffice. To model such situations, we introduce *restricting compositions*:

Definition 12 (Restricting composition of multi-asset declarative stages)**.** *Let $\langle \mathcal{AS}_1, \overline{\mathcal{C}}_1\rangle$ and $\langle \mathcal{AS}_2, \overline{\mathcal{C}}_2\rangle$ be multi-asset stage types, where $\mathcal{AS}_2$ is a subsequence of $\mathcal{AS}_1$ and $\overline{\mathcal{C}}_2 \subseteq \overline{\mathcal{C}}_1$, and let $\mathsf{D}_1 \colon \langle \mathcal{AS}_1, \overline{\mathcal{C}}_1\rangle$ and $\mathsf{D}_2 \colon \langle \mathcal{AS}_2, \overline{\mathcal{C}}_2\rangle$ be multi-asset declarative stages. The* restricting composition $\mathsf{D}_1 \downarrow \mathsf{D}_2$ *is defined by*

$$\begin{aligned}&\big(\mathsf{D}_1 \colon \langle \mathcal{AS}_1, \overline{\mathcal{C}}_1\rangle\big) \downarrow \big(\mathsf{D}_2 \colon \langle \mathcal{AS}_2, \overline{\mathcal{C}}_2\rangle\big)\\ &\quad = \big(\mathsf{D}_1 \downarrow \mathsf{D}_2\big) \colon \langle \mathcal{AS}_1, \overline{\mathcal{C}}_1\rangle\\ &\quad = \big(\mathit{member}_{\mathsf{D}_1\downarrow\mathsf{D}_2}, \mathit{consistent}_{\mathsf{D}_1\downarrow\mathsf{D}_2}\big) \ ,\end{aligned}$$

where the two predicates are defined in Fig. 4b.

Example 10. *Let us refine $\mathsf{D}_{\mathsf{sick,ok}}$ into two substages: In one sub-stage, the plant is very sick. In the other, the plant is barely sick. In the first stage, we require an additional, more involved controller for the pump. The encoding of the first restriction is shown in Fig. 4c.*

The stage resulting from the restricting composition (sick plant, working pump, aggressive controller) is

$$\mathsf{D}_{\mathsf{sick,ok,aggr}} = \big(\mathsf{D}_{\mathsf{sick}} \cdot \mathsf{D}_{\mathsf{ok}}\big) \downarrow \mathsf{D}_{\mathsf{aggr}} \ .$$

Note that $\mathsf{D}_{\mathsf{aggr}}$ can also be used for bigger aggregates, e.g., when a pump is connected to several plants and the pump preserves the condition of the stage to which it is applied.

5.2 Properties of Multi-Asset Stages

We now investigate the properties introduced for single-asset declarative stages: lifecycles and compatibility. They both carry over directly.

$$member_{\mathsf{D}_1 \cdot \mathsf{D}_2} = \{Y_1 \cup Y_2 \mid Y_1 \in member_{\mathsf{D}_1}, Y_2 \in member_{\mathsf{D}_2}\}$$
$$consistent_{\mathsf{D}_1 \cdot \mathsf{D}_2} = \{(Y, X) \mid Y = Y_1 \cup Y_2, X = X_1 \cup X_2, Y_1 \in member_{\mathsf{D}_1}, Y_2 \in member_{\mathsf{D}_2}, (Y_1, X_1) \in consistent_{\mathsf{D}_1}, (Y_2, X_2) \in consistent_{\mathsf{D}_2}\}$$

(a) Direct composition

$$member_{\mathsf{D}_1 \downarrow \mathsf{D}_2} = \{Y_1 \mid Y_1 \in member_{\mathsf{D}_1}, \exists Y_2 \subseteq Y_1.\ Y_2 \in member_{\mathsf{D}_2}\}$$
$$consistent_{\mathsf{D}_1 \downarrow \mathsf{D}_2} = \{(Y_1, X_1) \mid Y_1 \in member_{\mathsf{D}_1}, (Y_1, X_1) \in consistent_{\mathsf{D}_1}, \exists X_2 \subseteq X_1, Y_2 \subseteq Y_1.\ Y_2 \in member_{\mathsf{D}_2} \wedge (Y_2, X_2) \in consistent_{\mathsf{D}_2}\}$$

(b) Restricting composition

$$\mathsf{D}_{\mathsf{aggr}} = \{member_{\mathsf{aggr}}, consistent_{\mathsf{aggr}}\}$$
$$member_{\mathsf{aggr}} = \{a \in \mathsf{Plant} \mid i(a) < 0\}$$
$$member_{\mathsf{aggr}} = \{(\{a, b\}, Y) \mid a \in \mathsf{Pump}, Y \supseteq \{\mathsf{ranl}^{\leq 0.5}(a), \mathsf{ctrl}^{\leq 0.5}(a)\}\}$$

(c) Example for a cross-asset restriction.

Figure 4: Direct and restricting composition

Definition 13 (Multi-asset lifecycles)**.** *Let $\mathcal{T}$ be a multi-asset stage type with asset scheme $\mathcal{AS}$, and I an index set. A* lifecycle $\mathsf{L}_{\mathcal{AS}}$ *for $\mathcal{AS}$ consists of a set of declarative stages $(\mathsf{D}_i)_{i \in I}$ such that the following conditions hold:*

- $\forall i.\ \mathsf{D}_i : \mathcal{T}$,
- $\mathcal{AS} = \bigcup_{i \in I} member_{\mathsf{D}_i}$ *and*
- $\forall i, j \in I.\ i \neq j \Rightarrow member_{\mathsf{D}_i} \cap member_{\mathsf{D}_j} = \emptyset$.

Definition 14 ($\mathcal{A}$-compatibility)**.** *Let $\langle \mathcal{AS}_1, \overline{\mathcal{C}}_1 \rangle$ and $\langle \mathcal{AS}_2, \overline{\mathcal{C}}_2 \rangle$ be multi-asset declarative stage types, $\mathcal{A}$ an asset class that is part of both $\mathcal{AS}_1$ and $\mathcal{AS}_2$, assume that $\overline{\mathcal{C}}_1 \cap \overline{\mathcal{C}}_2 \neq \emptyset$, and let $\mathsf{D}_1 : \langle \mathcal{AS}_1, \overline{\mathcal{C}}_1 \rangle$ and $\mathsf{D}_2 : \langle \mathcal{AS}_2, \overline{\mathcal{C}}_2 \rangle$ be multi-asset declarative stages. We say that D_1 and D_2 are* $\mathcal{A}$-compatible *if the following holds:*

$$\forall a \in \mathcal{A}, A_1 \in member_{\mathsf{D}_1}, A_2 \in member_{\mathsf{D}_2}.\ a \in A_1 \wedge a \in A_2 \rightarrow \exists C_1, C_2.\ (A_1, C_1) \in consistent_{\mathsf{D}_1} \wedge (A_2, C_2) \in consistent_{\mathsf{D}_2}\ .$$

We say that D_1 and D_2 are compatible if they are $\mathcal{A}$-compatible for every $\mathcal{A}$ that occurs in both $\mathcal{AS}_1$ and $\mathcal{AS}_2$. We say that two lifecycles are ($\mathcal{A}$-)compatible if all their stages are pairwise ($\mathcal{A}$-)compatible.

Clearly, the lifting of single-asset stages to multi-asset stages preserves compatibility. We can also state a stronger statement: if two multi-asset declarative stages are compatible to a third stage, then so is their direct composition. Observe that restricting composition does not preserve compatibility: if the set of consistent components is restricted, there is no guarantee that those required for compatibility will not be removed. Consequently, compatibility has to be reproven for restricting composition. However, as one would expect for hierarchical groups, restricting compositions have the form $(\mathsf{D}_1 \cdot, \ldots, \cdot \mathsf{D}^n) \downarrow \mathsf{D}$, so compatibility proofs will consist of proving compatibility of single-asset stages and that restriction does not remove components that are critical for the consistency predicate.

Proposition 1 (Direct composition preserves compatibility)**.** *Let $\mathsf{D}^1_{\mathcal{AS}_1, \mathcal{C}}, \mathsf{D}^2_{\mathcal{AS}_2, \mathcal{C}}$ and $\mathsf{D}^3_{\langle \mathcal{A} \rangle, \mathcal{C}}$ be declarative stages. If D^1 and D^2 are $\mathcal{A}$-compatible to D^3, then $\mathsf{D}^1 \cdot \mathsf{D}^2$ is $\mathcal{A}$-compatible to D^3.*

5.3 Discussion

On Granularity

Declarative stages can be employed on both system and component levels, and multi-asset stages provide support for composition within these stages across (sub-)systems. For example, $\mathsf{D}_{\mathsf{sick}}$ is a single-component stage for plants, while $\mathsf{D}_{\mathsf{sick}} \cdot \mathsf{D}_{\mathsf{ok}}$ covers a system composed of a plant and a pump.

On Declarative Modelling

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

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

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

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

In summary, we require a unified representation that covers both the asset information and the architectural configuration of the managing system. The following sections demonstrate how semantic technologies can provide exactly this cohesion.

Although the above requirements arise from our declarative modelling of lifecycles and stages, they can be met by reusing established artefacts. In addition to domain ontologies (cf. Sect. 3), an individual asset can be captured with existing information models—such as the Asset Administration Shell (AAS)[4]—and subsequently linked to semantic data. While these asset models do not encode lifecycles themselves, they offer a structured foundation that improves scalability when lifecycle specifications are layered on top.

6 A Semantic Representation of Declarative Stages

The formalisation 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 realised in terms of *knowledge graphs* [9], using *ontologies* and associated *semantic technologies*, such as reasoners and databases.

For simplicity, we first consider the case of single-asset stages, and then build on this foundation for multi-asset stages. As a first step, let us describe the knowledge graph and ontology for semantics stages.

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, components of the

[4] https://industrialdigitaltwin.org/en/content-hub/aasspecifications

managing system, and their relations, in the form of an OWL ontology [5] with two layers. First, the *stage core ontology* describes the core concepts for all systems. Given a concrete set of asset classes and components of the managing system, it is then extended to an *extended stage ontology*. We illustrate the concepts concrete in terms of our greenhouse example, for which we consider an ontology with classes for requirement analysers and controllers.

Definition 15 (Stage Core Ontology)**.** *The* stage core ontology *expresses the existence of assets, aggregates and components, their disjointness and relation:*

DisjointClasses: Asset, Component, Aggregate
ObjectProperty: assignedTo
Domain: Asset **Range**: Component
ObjectProperty: hasPart
Domain: Aggregate **Range**: Asset .

For a concrete system, we extend the stage core ontology with axioms for components and assets. This extended stage ontology can be further extended with further classes, properties and axioms, as long as it remains consistent, and no class is trivially empty.

Definition 16 (Extended Stage Ontology)**.** *Let $\mathcal{A}_1, \ldots, \mathcal{A}_n$ be asset classes, $\mathcal{C}_1, \ldots, \mathcal{C}_m$ component classes, $I_1, \ldots, I_l$ functions that have some asset or component class as their domain, and an XSD data type as their range. An ontology is an* extended stage ontology *if it is consistent and contains (1) the axioms from Def. 15, (2) the axioms below, and (3) any other axioms provided that the classes contained in the axioms of (1) and (2) are non-empty.*

Class: Asset **DisjointUnionOf**: $\mathcal{A}_1, \ldots, \mathcal{A}_n$
Class: Component **DisjointUnionOf**: $\mathcal{C}_1, \ldots, \mathcal{C}_m$
DataProperty: I_1
Domain: **dom** I_1 **Range**: **rng** I_1
Characteristics: **functional**
...
DataProperty: I_l
Domain: **dom** I_l **Range**: **rng** I_l
Characteristics: **functional** .

[5] We do not align with any specific top-level ontology that deals with mereotopology [86], since no specific properties are needed in our encoding. In general, the choice would depend on the ontologies used in the overall application.

In the sequel, every considered ontology is assumed to be an extended stage ontology according to Def. 16.

Example 11 (An extended core ontology for the Greenhouse)**.** *For the running example of a greenhouse, we need information about plants, NVDI values, and the considered requirement analysers. Figure 5 show the axioms that are added to the core ontology. The only asset class is* Basil*, which is described by the first axiom. There are two component classes,* Ranalyser *and* Controller*, and one function that models the NVDI values of the plants (*nvdi*). The additional axioms refine* assignedTo *and* Ranalyser*. The requirement analyser classes are modelled as disjoint; using semantic technologies, we can also express subtype relations on requirement analyser classes using subclass axioms.*

A *semantic stage* is a representation in the knowledge graph of a declarative stage $\mathsf{D}_{\mathcal{A},\overline{\mathcal{C}}} = \langle member, consistent \rangle$ (see Def. 4), where all conditions and properties of membership and consistency are expressed in terms of OWL axioms.

Definition 17 (Semantic Stages)**.** *Let $\mathcal{T} = \langle \mathcal{A}, \overline{\mathcal{C}} \rangle$ be a stage type. A* semantic stage $\mathsf{S}{:}\mathcal{T} = \langle \mathsf{S}_{\mathsf{member}}, \mathsf{S}_{\mathsf{cons}} \rangle$ *is a pair of two OWL class names $\mathsf{S}_{\mathsf{member}}$ and $\mathsf{S}_{\mathsf{cons}}$ such that the following axioms must hold for $\mathsf{S}_{\mathsf{member}}$ and $\mathsf{S}_{\mathsf{cons}}$:*

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

where the symbol SC *is a concept defined as*

SC **EquivalentTo**: P_1 **some** C_1, ..., P_n **some** C_n .

Here, the axiom

ObjectProperty: P_i **SubPropertyOf**: assignedTo

has to hold for each P_i, and the axiom

Class: C_i **SubClassOf**: $\mathcal{C}$'

has to hold for each C_i and for some $\mathcal{C}' \in \overline{\mathcal{C}}$.

Additionally, $\mathsf{S}_{\mathsf{member}}$ must be independent of Component*; i.e, neither* Component *nor any of its subclasses occur in any axiom for $\mathsf{S}_{\mathsf{member}}$.* [6]

For a semantic stage S, $\mathsf{S}_{\mathsf{member}}$ is the *membership class* and $\mathsf{S}_{\mathsf{cons}}$ the *consistency class.*

[6] Ontology modules [87] can make this condition more precise: None of these classes should be in the module of $\mathsf{S}_{\mathsf{member}}$.

Class: Basil **SubClassOf**: Asset **and** nvdi **some** int
Class: Component **DisjointUnionOf**: Ranalyser, Controller
DataProperty: nvdi **Domain**: Basil **Range**: xsd::int **Characteristics**: **functional**
Class: RanalyserMoistUnder5 **SubClassOf**: Ranalyser
Class: RanalyserMoistUnder10 **SubClassOf**: Ranalyser
ObjectProperty: analysedBy **SubPropertyOf**: assignedTo **Domain**: Asset **Range**: Ranalyser
ObjectProperty: controlledBy **SubPropertyOf**: assignedTo **Domain**: Asset **Range**: Controller

Figure 5: Example extended stage ontology for the Greenhouse.

Example 12. *Consider semantic stages corresponding to the declarative stages from Fig. 3. First the healthy stage*

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

can be represented as follows:

Class: Healthy **SubClassOf**: Basil
and nvdi **some** int [> 5]
Class: HealthyCons **EquivalentTo**: Healthy
and analysedBy **some** RanalyserMoistUnder10 .

Next, the sick stage $\mathsf{S}_{\mathsf{Sick}} = \langle \mathsf{Sick}, \mathsf{SickCons} \rangle$ *can be represented by*

Class: Sick **SubClassOf**: Basil **and** nvdi **some** int[≤5]
Class: SickCons **EquivalentTo**: Sick
and analysedBy **some** RanalyserMoistUnder5 .

A *lifecycle* (see Def. 5) can be represented by semantic stages with disjoint members. To define semantic lifecycles, we first formalise compatibility and disjointness of semantic stages as follows:

Definition 18 (Compatibility and disjointness)**.** *Let* $\mathsf{S}_1 = \langle \mathsf{S}^1_{\mathsf{member}}, \mathsf{S}^1_{\mathsf{cons}} \rangle$ *and* $\mathsf{S}_2 = \langle \mathsf{S}^2_{\mathsf{member}}, \mathsf{S}^2_{\mathsf{cons}} \rangle$ *be semantic stages. Then*

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

Definition 19 (Semantic Lifecycle)**.** *A* semantic lifecycle *for an asset class* $\mathcal{A}$ *is a set of stages* $\mathsf{S}_i = \langle \mathsf{S}^i_{\mathsf{member}}, \mathsf{S}^i_{\mathsf{cons}} \rangle$ *for* $\mathcal{A}$ *such all membership classes are disjoint:*

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

6.1 Multi-Asset Semantic Stages

Multi-asset declarative stages (see Def. 9) are represented using subclasses of Aggregate for their membership class, instead of the asset classes in Def. 17.

Definition 20 (Multi-Asset Semantic Stages)**.** *Let* $\langle \mathcal{AS}, \overline{\mathcal{C}} \rangle$ *be a multi-asset stage type with* $\mathcal{AS} = \langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$ *and* $\overline{\mathcal{C}} = \{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$*, and let* $\#\mathcal{A}$ *be the number of occurrences of* $\mathcal{A}$ *in* $\langle \mathcal{AS}, \overline{\mathcal{C}} \rangle$*. A* multi-asset semantic stage $\mathsf{S}{:}\mathcal{T} = \langle \mathsf{S}_{\mathsf{member}}, \mathsf{S}_{\mathsf{cons}} \rangle$ *is a pair of two OWL class names* $\mathsf{S}_{\mathsf{member}}$ *and* $\mathsf{S}_{\mathsf{cons}}$*. The following axioms must hold for* $\mathsf{S}_{\mathsf{member}}$ *and* $\mathsf{S}_{\mathsf{cons}}$*, for some* $m \in \mathbb{N}$*:*

Class: $\mathsf{S}_{\mathsf{member}}$ **SubClassOf**: Aggregate **and**
partOf **exactly** $\#\mathcal{A}_1$ $\mathcal{A}_1$ **and** ...
and partOf **exactly** $\#\mathcal{A}_n$ $\mathcal{A}_n$
Class: $\mathsf{S}_{\mathsf{cons}}$ **EquivalentTo**: $\mathsf{S}_{\mathsf{member}}$ **and**
hasPart SC_1 ... **and** hasPart SC_m .

Here, all SC_i *are constrained as in Def. 17. Additionally, we enforce that* $\mathsf{S}_{\mathsf{cons}}$ *must be defined using a single axiom of the single form.*

In contrast to declarative stages, we cannot identify single-asset semantic stages with multi-asset semantic stages over single assets, because the membership classes are defined explicitly over disjoint classes. However, given a single-asset semantic stage, we can define a *lifting* into an equivalent multi-asset semantic stage.

Definition 21 (Lifting)**.** *Let* $\mathcal{T}$ *be a single-asset stage type, and* S *a single-asset semantic stage such that* $\mathsf{S} : \mathcal{T}$*. The* lifting *of* S*, denoted* $\lceil \mathsf{S} \rceil$*, with multi-asset stage type* $\langle\langle \mathcal{A} \rangle, \overline{\mathcal{C}} \rangle$*, is defined as follows:*

Class: $\lceil \mathsf{S} \rceil_{\mathsf{member}}$ **EquivalentTo**: Aggregate **and**
hasPart **exactly** 1 $\mathsf{S}_{\mathsf{member}}$
Class: $\lceil \mathsf{S} \rceil_{\mathsf{cons}}$ **EquivalentTo**: $\lceil \mathsf{S} \rceil_{\mathsf{member}}$ **and**
hasPart **exactly** 1 $\mathsf{S}_{\mathsf{cons}}$.

We can express both kinds of composition for semantic stages, by adding additional axioms for the class names of the composed stages. First, let us define direct composition (see Def. 10).

Definition 22 (Direct Composition). *Let* S^1 *and* S^2 *be multi-asset semantic stages with types* $\langle \mathcal{AS}_1, \overline{\mathcal{C}}_1 \rangle$ *and* $\langle \mathcal{AS}_2, \overline{\mathcal{C}}_2 \rangle$. *Let* $\mathcal{A}_1, \ldots, \mathcal{A}_n$ *be the asset classes in* $\mathcal{AS}_1 \cdot \mathcal{AS}_2$. *The* direct composition $\mathsf{S}^1 \cdot \mathsf{S}^2$ *has type*

$$\langle \mathcal{AS}_1, \overline{\mathcal{C}}_1 \rangle \cdot \langle \mathcal{AS}_2, \overline{\mathcal{C}}_2 \rangle = \langle \mathcal{AS}_1 \cdot \mathcal{AS}_2, \overline{\mathcal{C}}_1 \cup \overline{\mathcal{C}}_2 \rangle$$

and the following definition:

Class: $\mathsf{S}^1{\cdot}\mathsf{S}^2_{\mathsf{member}}$ **SubClassOf**: Aggregate **and** partOf **exactly** $\#\mathcal{A}_1$ $\mathcal{A}_1$ **and** ... **and** partOf **exactly** $\#\mathcal{A}_n$ $\mathcal{A}_n$
Class: $\mathsf{S}^1{\cdot}\mathsf{S}^2_{\mathsf{cons}}$ **EquivalentTo**: $\mathsf{S}^1_{\mathsf{cons}}$ **and** $\mathsf{S}^2_{\mathsf{cons}}$.

There is no schematic definition of the restricting composition (see Def. 12), as there is no subset selection in OWL. However, we can give an informal pattern for how to implement it.

Example 13 (Restricting composition). *Let us consider the semantic equivalent of Example 10 and assume given an extended ontology with the terms needed for pumps. We formalise operational pumps by* $\mathsf{S}_{\mathsf{ok}} = \langle \mathsf{PumpOk}, \mathsf{PumpOkCons} \rangle$ *with:*

Class: PumpOk **SubClassOf**: Pump **and** status **value** ok
Class: PumpOkCons **EquivalentTo**: Pump **and** assignedTo **some** RanalyserPowerOver10 .

The lifting of the classes S_{ok} *and* $\mathsf{S}_{\mathsf{Healthy}}$ *(i.e.,* $\lceil \mathsf{S}_{\mathsf{ok}} \rceil$ *and* $\lceil \mathsf{S}_{\mathsf{Healthy}} \rceil$*), is defined as follows:*

Class: PumpOkLift **SubClassOf**: Aggregate **and** hasPart **exactly** 1 PumpOk
Class: PumpOkLiftCons **EquivalentTo**: PumpOkLift **and** hasPart **exactly** 1 PumpOkCons
Class: HealthyLift **SubClassOf**: Aggregate **and** hasPart **exactly** 1 Healthy
Class: PumpOkLiftCons **EquivalentTo**: HealthyLift **and** hasPart **exactly** 1 HealthyCons .

Their direct composition, following Def. 22, becomes

Class: OkHealthy **SubClassOf**: Aggregate **and** hasPart **exactly** 1 PumpOk **and** hasPart **exactly** 1 Healthy
Class: PumpOkLiftCons **EquivalentTo**: OkHealthy **and** HealthyCons **and** PumpOkCons .

Now, to restrict the composed stage to aggressive control, we need to modify the consistency class definition by unrolling PumpOkCons *and adding a restriction:*

Class: OkHealthyAggr **SubClassOf**: Aggregate **and** hasPart **exactly** 1 PumpOk **and** hasPart **exactly** 1 Healthy
Class: PumpOkLiftCons **EquivalentTo**: OkHealthy **and** HealthyCons **and** hasPart **exactly** 1 (PumpOkCons **and** controlledBy **some** AggressiveCtrl) .

Definitions 18 and 19 carry over to multi-asset semantic stages.

6.2 Discussion

The patterns that capture relations between classes and the axioms required for these patterns, as used in Example 13, are not directly expressible in OWL. However, these patterns can be represented using another semantic technology, namely *ontology templates* [88, 89]. The templates for our running system are given in the auxiliary online material. 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 $\mathsf{S}_{\mathsf{member}}$. Otherwise, the reasoner could deduce the existence of requirement analysers, 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 the solution in this paper), one could add additional axioms that essentially enforce a closed-world semantics, or one could use a different formalisation for stages that does not require open-world reasoning, such as SHACL shapes [90].

Note that repairing of the managing system configuration corresponds to so-called ABox abduction [91] 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.

7 A Self-Adaptive Architecture for Lifecycle Management

The overall aim of the self-adaptive architecture is to maintain the architectural coherence of the managing system while assets transition between the declarative stages in their lifecycles; i.e., every asset must have the correct associated components

in the managing system, reflecting the current declarative stages of the asset. The architecture of the self-adaptive managing system is depicted in Fig. 6 for two generic components in the managing system, such as requirement analysers and controllers. It has the following architectural components; the first four components constitute the layer for *architectural reconfiguration*, while the components of the managing system, together with the assets of the managed systems constitute the system being managed by the architectural reconfiguration layer. 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 the current configuration, i.e., the components in the managing system and the assets, and the stages. The KB offers operations for the other components to manipulate and query this information. Note that the configuration is runtime information and can change, while the information about the stages is static.
- **Monitor:** The monitor component includes a semantic tagger that acts as the entry point for information about the assets of the managed system, translates this information into semantic data and adds it to the KB. Although the Monitor component is used for both the behavioural and architectural feedback loop of Fig. 1, we here focus on its use for architectural self-adaptation.
- **Stage Analyser:** The stage analyser identifies changes to the lifecycle stages of the different assets by querying the KB for inconsistencies. This component is the Layer 2 analyser of Fig. 1.
- **Stage Planner:** The stage planner determines how to adapt the components in the managing system (Layer 1), to resolve the inconsistencies in the KB. It identifies components in the managing system that need to be added and removed. This component is the Layer 2 Planner of Fig. 1.
- **Components:** The components in the managing system address behavioural self-adaptation, including requirement analysis, planning and control for the different assets of the managed system. 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 components of the managing system, 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. 6).

7.1 The Knowledge Base

The task of the KB is to manage knowledge about the assets, their aggregates, as well as the architectural configuration of the managing system; i.e., the KB tracks information about declarative stages, requirement analysers and controllers. This requires a knowledge store for the information, operations to add and remove information, and a query interface that utilises the managed knowledge to enable the stage analyser to detect lifecycle changes and the stage planner to plan changes to the managing system 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, Aggregate or subclasses of Component. These *closing axioms* take the following form for a class C and individuals $i_1, \ldots, i_n$:

$$\mathsf{C}\ \textbf{EquivalentTo}\ \{i_1, \ldots, i_n\}\ .$$

Furthermore, we must express that an aggregate has no additional parts. For an aggregate aggr with parts $\mathsf{ast}_1, \ldots, \mathsf{ast}_n$, this axiom takes the following form:

$$\{\mathsf{aggr}\}\ \textbf{SubClassOf}\text{: hasPart only } \{\mathsf{ast}_1, \ldots, \mathsf{ast}_n\}\ .$$

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.

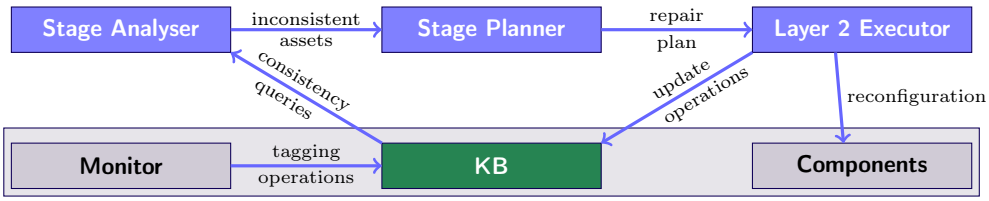

Figure 6: A self-adaptive architecture for stage-based lifecycle management.

Example 14. *Consider the following knowledge graph $\mathcal{K}$ that models the semantic stages from the previous section, as well as one asset with one requirement analyser on the second level of the MAPE-K loop.*

The set $\mathcal{K}_{\text{store}}$ of stored triples becomes:

(ast1, *a*, Basil), (ast1, nvdi, 4)
(ast1, analysedBy, rq1)
(rq1, *a*, RanalyserMoistUnder5) .

Note that the water level is not *stored in the KB. It is handled by the requirement analyser and is not relevant for stage membership or consistency. In addition to the axioms of Def. 15 and Examples 11 and 12, the ontology $\mathcal{K}_{\text{ontology}}$ contains the following axioms:*

Asset **EquivalentTo** { ast1 } ,
Ranalyser **EquivalentTo** { rq1 } .

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 components in the managing system. The set of known declarative stages, however, remains static.

Definition 23 (Operations)**.** *Let $\sqcup$ denote disjoint union,* A *an OWL asset class,* C *an OWL controller class, and* RA *an OWL requirement analyser class. Given a knowledge graph $\mathcal{K}$ and an asset node* ast*, we let* neighbour$_{\mathcal{K}}$(ast) *denote the set of triples that has* ast *as either subject or object in $\mathcal{K}$. The set of operations on knowledge graphs, and their effects, is defined in Fig. 7.*

In our running example, let ranalysers$_{\mathcal{K}}$(ast) denote the set of requirements analysers *ra* with triples of the form (ast, analysedBy, ra) and control$_{\mathcal{K}}$(ast) be the set

{(ast, controlledBy, ctrl), (ctrl, *a*, C)} .

if such a ctrl exists, and $\emptyset$ otherwise.

The 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. The operation CLEAN(ast, C) removes all components in the managing system of class C assigned to an asset ast. The operation REMOVE(ast) removes an asset, as well as all its associated components, in the managing system. The operation UPDATE(ast, p, v) updates the information modelled by property p for asset ast to value v. The operation ADD(ast, C, M) adds a component to the KB, again updating the relevant closing axioms as well as adding a triple that connects the nodes and classes. In the definition of ADD(ast, C, M), the additional parameter P can be used to provide a subproperty of assignedTo to connect the component in the managing system to the asset. It defaults to assignedTo. The operation AGGREGATE(ast$_1$,...,ast$_n$) creates a new aggregate out of pre-existing assets. The definition of these operations in terms of SPARQL queries is given in Fig. 8. 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 components in the managing system are used only to reflect the changes during reconfiguration, while the operations on the asset itself are updated based on incoming data.

Example 15. *We continue with Example 14. We update the NVDI value of the asset by the operation* UPDATE(ast1, nvdi, 4)*, add a new asset by the* ADD(ast2, FPGA) *operation, add a controller component to the asset by means of the operation* ADD(ast2,ctrl1,BasilController,controlledBy) *and aggregate the two assets using the operation* AGGREGATE(ast1,ast2)*. After these operations, the updated knowledge graph $\mathcal{K}^1_{\text{store}}$ contains the following triples:*

$[\![\mathcal{K} \sqcup \{\mathsf{A}\ \textbf{EquivalentTo}\colon \{\mathsf{a}_1, \ldots, \mathsf{a}_n\}\}, \mathsf{ADD}(\mathsf{ast}, \mathsf{A})]\!]$
$= \mathcal{K} \cup \{\mathsf{A}\ \textbf{EquivalentTo}\colon \{\mathsf{a}_1, \ldots, \mathsf{a}_n, \mathsf{ast}\}\} \cup \{(\mathsf{ast}, a, \mathsf{A})\}$

$[\![\mathcal{K} \sqcup \{\mathsf{C}\ \textbf{EquivalentTo}\colon \{\mathsf{c}_1, \ldots, \mathsf{c}_n\}\}, \mathsf{CLEAN}(\mathsf{ast},\mathsf{C})]\!]$
$= (\mathcal{K} \backslash \mathsf{neighbour}_{\mathcal{K}}(\mathsf{ast})) \cup \{\mathsf{C}\ \textbf{EquivalentTo}\colon (\{\mathsf{c}_1, \ldots, \mathsf{c}_n\} \backslash \mathsf{C}_{\mathcal{K}}(\mathsf{ast}))$

$[\![\mathcal{K} \sqcup \{(\mathsf{ast}, a, \mathsf{A}), \mathsf{A}\ \textbf{EquivalentTo}\colon \{\mathsf{a}_1, \ldots, \mathsf{a}_n, \mathsf{ast}\}\}, \mathsf{REMOVE}(\mathsf{ast})]\!]$
$= \bigcap_{\text{Component class } \mathsf{C}} [\![\mathcal{K}, \mathsf{CLEAN}(\mathsf{ast},\mathsf{C})]\!] \cup \{\mathsf{A}\ \textbf{EquivalentTo}\colon \{\mathsf{a}_1, \ldots, \mathsf{a}_n\}\}$

$[\![\mathcal{K}, \mathsf{UPDATE}(\mathsf{ast}, \mathsf{p}, \mathsf{v})]\!]$
$= (\mathcal{K} \backslash \{(\mathsf{ast}, \mathsf{p}, \mathsf{X}) \mid \mathsf{X} \in \mathbf{dom}(\mathsf{p})\}) \cup \{(\mathsf{ast}, \mathsf{p}, \mathsf{v})\}$

$[\![\mathcal{K} \sqcup \{\mathsf{C}\ \textbf{EquivalentTo}\colon \{\mathsf{c}_1, \ldots, \mathsf{c}_n\}\}, \mathsf{ADD}(\mathsf{ast}, \mathsf{c}, \mathsf{C}, \mathsf{P})]\!]$
$= \mathcal{K} \cup \{\mathsf{C}\ \textbf{EquivalentTo}\colon \{\mathsf{c}, \mathsf{c}_1, \ldots, \mathsf{c}_n\}\} \cup \{(\mathsf{ast}, \mathsf{P}, \mathsf{c}), (\mathsf{c}, a, \mathsf{C})\}$

$[\![\mathcal{K}, \mathsf{AGGREGATE}(\mathsf{ast}_1, \ldots, \mathsf{ast}_n)]\!]$ (for a fresh aggr)
$= \mathcal{K} \cup \{(\mathsf{aggr}, a, \mathsf{Aggregate}), \{\mathsf{aggr}\}\ \textbf{SubClassOf}\colon \mathsf{hasPart}\ \textbf{only}\ \{\mathsf{ast}_1, \ldots, \mathsf{ast}_n\}\}$
$\cup \{(\mathsf{aggr}, \mathsf{hasPart}, \mathsf{ast}_1), \ldots, (\mathsf{aggr}, \mathsf{hasPart}, \mathsf{ast}_n)\}$

Figure 7: Semantics of knowledge store operations.

(ast1, a, Basil)
(ast1, nvdi, 4)
(ast2, a, FPGA)
(ast1, analysedBy, rq1)
(ast2, controlledBy, ctrl1)
(aggr, a, Aggregate)
(aggr, hasPart, ast1)
(aggr, hasPart, ast2)
(rq1, a, RanalyserMoistUnder5)
(ctrl1, a, BasilController) .

Its ontology $\mathcal{K}^1_{\text{ontology}}$ *contains, in addition to the axioms of Def. 15 and Examples 11 and 12, the following axioms:*

Asset **EquivalentTo** {ast1, ast2}
Ranalyser **EquivalentTo** rq1}
Controller **EquivalentTo** {ctrl1}
Aggregate **EquivalentTo** {aggr}
{aggr} **EquivalentTo** {ast1,ast2} .

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

Definition 24 (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 components of class* C *that are assigned to* a *in knowledge graph* $\mathcal{K}$*.*

Example 16. *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 analyser (recall that the update performed in Example 14 moved* ast1 *from its sick to its healthy stage), and second,* ast2 *has no requirement analyser:*

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

Recall that aggregates can overlap and that an asset can be part of several aggregates. Additionally, an asset may be part of one or more single-asset stage, and multi-assets stages. In case the later situation is unwanted, then the query above be modified to explicitly select only those assets that are not part of aggregates:

Healthy **and not** HealthyCond
and not inverse(hasPart) **some** Aggregate .

In the sequel, we do not exclude this situation.

7.2 The 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;

- ADD(ast,A) maps to

```
INSERT DATA { uri(ast) rdf:type uri(A) }
```

- ADD(ast,c,C,P) maps to

```
INSERT DATA { uri(c) rdf:type uri(C).
              uri(ast) P 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}
```

- AGGREGATE($\mathsf{ast}_1, \ldots, \mathsf{ast}_n$) maps to the following, where aggr is a fresh URI.

```
INSERT DATA { aggr rdf:type Aggregate.
              aggr hasPart uri(ast_1).
              ...
              aggr hasPart uri(ast_n).}
```

Figure 8: Mapping operations to SPARQL queries. Operation uri(·) retrieves the URI of an asset, components, asset kind or other parameter.

- 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 analysers and its eventual controller from the KB and system, if the asset observation stream notifies of its removal from the managed 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.

```
1  while(true)
2    toGenerate := ∅, toAdd := ∅, toRemove := ∅
3    foreach stage S = ⟨S_member, S_cons⟩
4      // analyser: 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 9: A self-adaptation algorithm combining stage analyser and planner.

7.3 The Stage Analyser

The stage analyser is responsible for detecting inconsistencies between the current stage of an asset and its associated components in the managing system. To this aim, the stage analyser queries the KB for the assets that satisfy stage membership of different stages and checks whether the components in the managing system for these assets comply with the consistency relation of the stage. The stage analyser is realised by the first part of the algorithm in Fig. 9, which implements the MAPE-K loop for architectural self-adaptation. The algorithm iterates through all stages and finds the inconsistent assets (Sect. 7.2). It then collects all such assets in a set (Sect. 7.2) that records the stage with which the assets need to be consistent (Sect. 7.2).

7.4 The Stage Planner

The stage planner is responsible for repairing the managing system once an inconsistency is detected between the components in the managing system, and the stage associated with an asset. The stage planner requires a set of compatible lifecycles, and is realised by the second part of the algorithm in Fig. 9.

Repair is based on abducing an explanation for the required consistency. This explanation may add or remove components in the managing system. Thus, abduction is performed on all inconsistencies at once. We formalise this procedure as follows. First, copy the KB (Sect. 7.2), then remove all components of the managing system (Sect. 7.2). The abduction then derives the new components that are needed to explain the consistency of this KB (Sect. 7.2). Finally, abduced components that are not already present,[7] are added to the managing system and to the original KB (Sect. 7.2). Components that are no longer needed, are similarly removed (Sect. 7.2).

7.5 The Executor

The executor receives the plan, i.e., the components to be removed or created, and performs it. Additionally, it uses the operations defined in Fig. 8 to update the knowledge base.

8 Evaluation

We evaluate declarative stages and their realisation as semantic stages to answer the following research questions (introduced in Sect. 1):

RQ1: *Can declarative stages be used to model an existing SAS?*

RQ2: *Can multi-asset stages be used to model mutually dependent assets and their lifecycles in a SAS?*

RQ3: *How does using semantic stages over non-semantic declarative stages affect performance?*

RQ1 and **RQ2** are qualitative and indicate the general applicability of declarative stages, while **RQ3** 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 three experiments, available in the auxiliary online material.

8.1 Experimental Design and Setup

Experiment **EX1** addresses **RQ1**. The experiment uses the example from Sect. 2, based on GreenhouseDT [8], a DT exemplar specifically designed for structural self-adaptation. We have implemented both semantic stages and non-semantic declarative stages. The latter were implemented using suitable data structures for the knowledge base, while semantic stages were implemented with an RDF knowledge graph based on Apache Jena[8] and its built-in reasoner. Thus, adding a new stage requires defining it in the ontology and to implement the corresponding class in the architecture, respectively. The architecture is implemented directly in the object-oriented language Kotlin. At its core, it implements Fig. 6, and supports switching between a KB based on Apache Jena for semantic stages and a KB that implements declarative stages as Kotlin data structures. The interface, i.e., the operations of Sect. 7, are methods, where the implementation of the interface maps to SPARQL queries for semantic stages, or operations on the data structures for non-semantic stages.

Experiment **EX2** addresses **RQ2**. The experiment extends GreenhouseDT by a new, explicit lifecycle model for pumps and uses direct composition to manage pairs of pumps and plants. This experiment is implemented for both semantic stages and non-semantic declarative stages, analogously to **EX1**.

EX3 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 [89]. For each configuration (n, m), we measure the time needed for the non-semantic and semantic stages to adapt from a random starting stage, respectively. This experiment addresses **RQ3**. All experiments were performed on a Ubuntu 22.04 laptop with a i7-1355U CPU and 16 GB RAM.

[7] For simplicity, the task of matching new with existing components has been omitted from the algorithm.

[8] https://jena.apache.org/

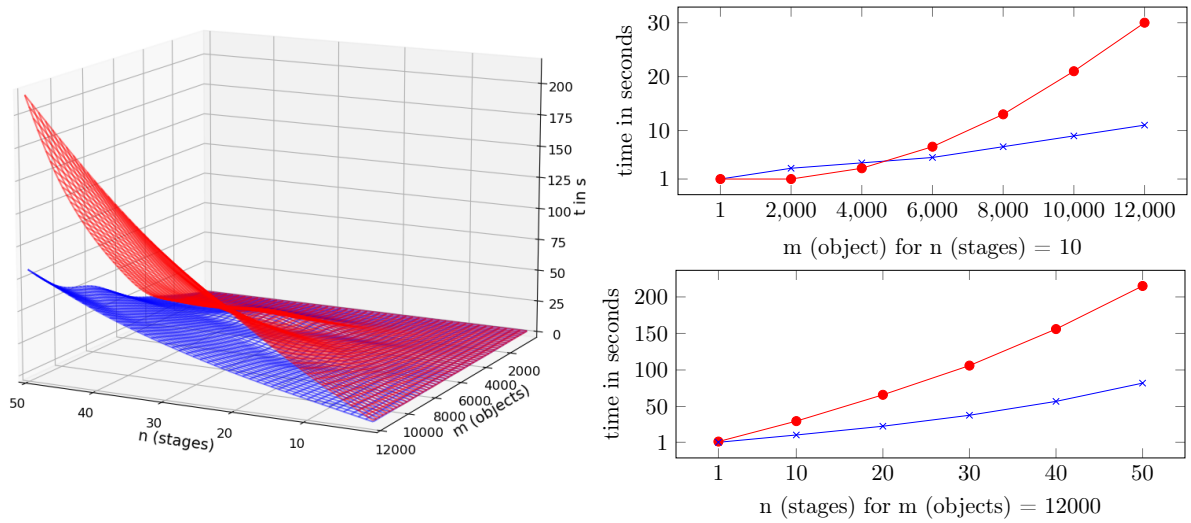

Figure 10: Evaluation Results for n stages and m assets, with time t measured in seconds s. Red (and round) data points denote measurements using non-semantic stages, and blue (and crossed) data points denote measurements using semantic stages.

8.2 Results

We can answer **RQ1** and **RQ2** positively: we are able to model the stages and lifecycles needed for GreenhouseDT, and use lifecycle stages to adapt the managing system of this DT 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 lifecycles are sufficient.

Regarding, modelling efforts, the GreenhouseDT demonstrator was modelled and implemented by the first author, who is experienced with both semantic technologies and conventional (non-semantic) DT solutions. No appreciable difference was observed in the effort required to define semantic versus non-semantic stages. The only substantive divergence emerged during repair: for non-semantic stages the repair logic had to be coded manually, whereas for semantic stages the existing generic abduction algorithm could be reused unchanged.

Figure 10 shows results for **EX3**. The left figure shows the time (in seconds) needed for semantic stages (blue) and non-semantic stages (red) for one self-adaptation cycle, when considering stages $n \in \{1, 10, 20, 30, 40, 50\}$ and assets $m \in \{1, 2000, 4000, 6000, 8000, 10000, 12000\}$ (intermediate values have been 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 behaviour 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 ($m = 12000$), semantic stages scale strictly better. To answer **RQ3**, the overhead of using semantic stages improves performance as the number of stages and assets rises.

8.3 Threats to Validity

A threat to *internal validity* is that our implementation of non-semantic stages is a direct implementation of the architecture in Kotlin, while the implementation of semantic stages reuses a mature graph database. Nevertheless, we consider the comparison to be realistic, as both implementations are general enough to accommodate new stages and lifecycles. A highly specialised system for non-semantic stages 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** and **EX2** are performed on a single case study

and **EX3** 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.

9 Discussion

Implementing our method consists of two steps. First, one must must model the lifecycles. Second, one must implement the two-layered architecture.

On semantic technologies

Our experiments indicate that the technological overhead of adopting the proposed approach is minimal. We have two arguments for this.

1. Knowledge graphs, including property graphs such as Neo4J, and ontologies are a technology with a rich ecosystem for both software and methodology; numerous tutorials and stable open source tools suits are available. Following the method as described above does not require training in technologies that are otherwise irrelevant for self-adaptation in cyber-physical systems. Similarly, the rest of the solution follows a conventional MAPE-K loop.
2. Ontologies are a common technology for asset modelling and DTs, meaning that following our method uses standard technologies in their context. Semantic stages merely require an ontology that captures asset information. Fortunately, such asset information models are already defined by several interoperability standards (e.g., IEC CDD, the Asset Administration Shell, SAREF), enabling practitioners to apply the approach without crafting custom vocabularies from scratch.

From a more technical perspective, a standard open-source knowledge-graph engine and rule reasoner are sufficient, where highly optimised systems are available that add no extra runtime constraints and little overhead.

On modelling in self-adaptation

Our main conceptual contributions are (1) hierarchical self-adaptation where structural self-adaptation is "above" behavioural self-adaptation and (2) a declarative notion of a lifecycle stage. Both have implications to modelling in self-adaptive systems.

An immediate consequence is that lifecycles are no longer thought off as possible sequences of transitions. Indeed, the notion of transition is only relevant in modelling when it comes to checking compatibility. Instead, lifecycles are sets of consistency conditions. This is natural in the context of MAPE-K, where the planner only has possible actions as its inputs and produces a sequence of actions as the plan. Using a declarative approach, the focus shifts towards modelling, with less emphasis on the transitions, as the operationalisation is generic. As previously argued, declarative modelling of lifecycle stages allows to scale lifecycles in our context, as the model scales with the number of stages, not the, potentially quadratic, number of transitions between them.

A more research-directed consequence is that the hierarchy between adaptations is not based on a system break down (where the hierarchy of MAPE-K loops follows the hierarchy of the system), but on different concepts: structure and behaviour.

Implementing declarative lifecycle management

The above observations suggest that implementing our method is feasible for an experienced software engineer for both modelling and implementation. If the ontology to describe the assets and their states is not given, an expert on semantic modelling should be considered.

For modelling, our notions of compatibility and composition can be used to guide the modelling efforts, providing both a modular structure and a way to check logical consistency of the model. In case of semantic stages, compatibility reduces to a standard reasoning task and ontology engineering methodologies, e.g., to align or match with existing ontologies [92].

10 Conclusions and Future Work

Self-adaptive systems can be realised by a feedback loop between a managed and a managing system. The managing system monitors the managed system, updates its internal model, and adjusts the managed system by means of controllers

to maintain given requirements. Unexpected shifts in the behaviour of the managed system can make the managing system inconsistent with the managed system, triggering a need for adaptation of the managing system as well, such as changes in the system requirements and in the associated analysers and controllers.

This paper proposes an automated method for self-adaptation of a managing system, to address shifts between such lifecycle stages in the managed system. The method is based on the novel notion of *declarative lifecycle stages* that specify such shifts in the behaviour of the managed system and a corresponding adaptation logic for the managing system. The specification of a lifecycle stage is declarative in the sense that it does not describe how an asset transitions to a stage, but rather provides a logical characterisation of what it means for the asset to be in the new stage and what it means for the managing system to be consistent with the asset at this new stage in its lifecycle. The paper considers both single-asset and multi-asset specifications of declarative lifecycle stages, and how these specifications compose.

Further, we introduce a two-layer self-adaptive architecture that realises declarative lifecycle management. When the managed system transitions between lifecycle stages, the upper (managing) layer consults formal stage specifications and adapts accordingly. Our prototype instantiates this architecture with semantic technologies: a knowledge graph stores the stage definitions, and a rule-based reasoner drives adaptation decisions. Finally, we empirically assess how effectively semantic reasoning supports declarative lifecycle management.

Future Work

We can identify at least three interesting directions for future work. First, the declarative lifecycles in this paper are fixed; i.e., the definition of the stages does not change. However, the definition of the lifecycles may also change, e.g., due to a changing organisational context or evolution of the design or domain. This would require to handle different versions of the lifecycle for single assets within the lifecycles of the digital twin as a whole, and we think that eventual inconsistencies between different versions of a stage can be handled using ideas from BPM [41], DevOps for Digital Twins [93] or declarative programming, where similar challenges have been solved through controlled updates or restrictions on the composition [94]. Second, we plan to introduce *hierarchical asset representations*, which will increase the framework's expressiveness and give engineers finer control over adaptation scopes. Finally, we aim to *align our ontology*, currently the interface for semantic stages and the repository of declarative stage definitions—with domain-specific ontologies. This alignment should boost interoperability and facilitate richer contextual integration.

Acknowledgements

The work was partly supported by the EU project SM4RTENANCE (grant no. 101123423), the UArctic project DART: Digital Arctic Twins and the Durham Strategic Research Funding JusTN0W. We thank Leif Harald Karlsen for his help with the OTTR templates.

References

[1] Bencomo, N., Cabot, J., Chechik, M., Cheng, B.H., Combemale, B., Wąsowski, A., Zschaler, S.: Abstraction engineering. arXiv preprint arXiv:2408.14074 (2024) https://doi.org/10.48550/ARXIV.2408.14074

[2] Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003) https://doi.org/10.1109/MC.2003.1160055

[3] Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. **4**(2), 14–11442 (2009) https://doi.org/10.1145/1516533.1516538

[4] Weyns, D.: An Introduction to Self-adaptive Systems: A Contemporary Software Engineering Perspective. John Wiley & Sons, New Jersey (2020). https://doi.org/10.1002/9781119574910

[5] Weyns, D., Gerostathopoulos, I., Abbas, N., Andersson, J., Biffl, S., Brada, P., Bures, T., Salle, A.D., Galster, M., Lago, P., Lewis, G.A., Litoiu, M., Musil, A., Musil, J., Patros,

P., Pelliccione, P.: Self-adaptation in industry: A survey. ACM Trans. Auton. Adapt. Syst. **18**(2), 5–1544 (2023) https://doi.org/10.1145/3589227

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

[7] Brun, Y., Serugendo, G.D.M., Gacek, C., Giese, H., Kienle, H.M., Litoiu, M., Müller, H.A., Pezzè, M., Shaw, M.: Engineering self-adaptive systems through feedback loops. In: Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 48–70. Springer, Heidelberg, Germany (2009). https://doi.org/10.1007/978-3-642-02161-9_3

[8] Kamburjan, E., Sieve, R., Baramashetru, C.P., Amato, M., Barmina, G., Occhipinti, E., Johnsen, E.B.: GreenhouseDT: An exemplar for digital twins. In: Proc. 19th Intl. Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2024), pp. 175–181. ACM, New York, NY, USA (2024). https://doi.org/10.1145/3643915.3644108

[9] Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., Melo, G., Gutierrez, C., Kirrane, S., Gayo, J.E.L., Navigli, R., Neumaier, S., Ngomo, A.N., Polleres, A., Rashid, S.M., Rula, A., Schmelzeisen, L., Sequeda, J.F., Staab, S., Zimmermann, A.: Knowledge graphs. ACM Comput. Surv. **54**(4), 71–17137 (2022) https://doi.org/10.1145/3447772

[10] Kamburjan, E., Bencomo, N., Tapia Tarifa, S.L., Johnsen, E.B.: Declarative lifecycle management in digital twins. In: Proceedings of the ACM/IEEE 27th Int. Conf. on Model Driven Engineering Languages and Systems (EDTconf), MODELS Companion 2024, pp. 353–363. ACM, New York, NY, USA (2024). https://doi.org/10.1145/3652620.3688248

[11] Kriegler, F.J., Malila, W.A., Nalepka, R.F., Richardson, W.: Preprocessing Transformations and Their Effects on Multispectral Recognition. Technical report, University of Michigan (January 1969)

[12] Abdelrahman, M., Macatulad, E., Lei, B., Quintana, M., Miller, C., Biljecki, F.: What is a digital twin anyway? deriving the definition for the built environment from over 15,000 scientific publications. Building and Environment **274**, 112748 (2025) https://doi.org/10.1016/j.buildenv.2025.112748

[13] Lehner, D., Zhang, J., Pfeiffer, J., Sint, S., Splettstößer, A., Wimmer, M., Wortmann, A.: Model-driven engineering for digital twins: a systematic mapping study. Softw. Syst. Model. **24**(5), 1339–1377 (2025) https://doi.org/10.1007/S10270-025-01264-7

[14] Kritzinger, W., Karner, M., Traar, G., Henjes, J., Sihn, W.: Digital twin in manufacturing: A categorical literature review and classification. IFAC-PapersOnLine **51**(11), 1016–1022 (2018) https://doi.org/10.1016/j.ifacol.2018.08.474 . 16th IFAC Symposium on Information Control Problems in Manufacturing INCOM 2018

[15] Flammini, F.: Digital twins as run-time predictive models for the resilience of cyber-physical systems: a conceptual framework. Philosophical Transactions of the Royal Society A **379**(2207), 20200369 (2021) https://doi.org/10.1098/rsta.2020.0369

[16] Kamburjan, E., Din, C.C., Schlatte, R., Tapia Tarifa, S.L., Johnsen, E.B.: Twinning-by-construction: ensuring correctness for self-adaptive digital twins. In: Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles - 11th International Symposium ISoLA, pp. 188–204 (2022). https://doi.org/10.1007/978-3-031-19849-6_12 . Springer

[17] Splettstößer, A.-K., Ellwein, C., Wortmann, A.: Self-adaptive digital twin reference architecture to improve process quality. Procedia CIRP **119**, 867–872 (2023) https://doi.org/10.1016/j.procir.2023.03.131 . The 33rd CIRP Design Conference

[18] Pileggi, P., Lazovik, E., Broekhuijsen, J., Borth, M., Verriet, J.: Lifecycle governance for effective digital twins: A joint systems engineering and IT perspective. In: 2020 IEEE International Systems Conference (SysCon), pp. 1–8 (2020). https://doi.org/10.1109/SYSCON47679.2020.9275662 . IEEE

[19] Edrisi, F., Perez-Palacin, D., Caporuscio, M., Giussani, S.: Adaptive controllers and digital twin for self-adaptive robotic manipulators. In: 18th Symposium on Software Engineering for Adaptive and Self-Managing Systems SEAMS, pp. 56–67 (2023). https://doi.org/10.1109/SEAMS59076.2023.00017 . IEEE

[20] Pfeiffer, J., Lehner, D., Wortmann, A., Wimmer, M.: Towards a product line architecture for digital twins. In: 2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C), pp. 187–190 (2023). https://doi.org/10.1109/ICSA-C57050.2023.00049 . IEEE

[21] National Academies of Sciences, Engineering, and Medicine (NASEM): Foundational Research Gaps and Future Directions for Digital Twins. The National Academies Press (2024). https://doi.org/10.17226/26894

[22] Klinkenberg, R.: Learning drifting concepts: Example selection vs. example weighting. Intell. Data Anal. **8**(3), 281–300 (2004) https://doi.org/10.3233/IDA-2004-8305

[23] Widmer, G., Kubat, M.: Learning in the presence of concept drift and hidden contexts. Machine Learning **3**, 69–101 (1996) https://doi.org/10.1023/A:1018046501280

[24] Torres, R., Bencomo, N., Astudillo, H.: 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), pp. 28–34 (2013). https://doi.org/10.1109/RAISE.2013.6615201

[25] Eramo, R., Bordeleau, F., Combemale, B., Den Brand, M., Wimmer, M., Wortmann, A.: Conceptualizing digital twins. IEEE Software **39**(2), 39–46 (2021) https://doi.org/10.1109/MS.2021.3130755

[26] Alberts, E., Gerostathopoulos, I., Malavolta, I., Corbato, C.H., Lago, P.: Software architecture-based self-adaptation in robotics. J. Syst. Softw. **219**, 112258 (2025) https://doi.org/10.1016/J.JSS.2024.112258

[27] Silva, G.R., Garcia, N.H., Bozhinoski, D., Deshpande, H., Oviedo, M.G., Wasowski, A., Montero, M.R., Corbato, C.H.: MROS: A framework for robot self-adaptation. In: 45th IEEE/ACM International Conference on Software Engineering: ICSE 2023 Companion Proceedings, Melbourne, Australia, May 14-20, 2023, pp. 151–155 (2023). https://doi.org/10.1109/ICSE-COMPANION58688.2023.00044

[28] Silva, G.R., Päßler, J., Zwanepol, J., Alberts, E., Tapia Tarifa, S.L., Gerostathopoulos, I., Johnsen, E.B., Corbato, C.H.: SUAVE: an exemplar for self-adaptive underwater vehicles. In: 18th IEEE/ACM Symposium on Software Engineering for Adaptive and Self-Managing Systems SEAMS, pp. 181–187 (2023). https://doi.org/10.1109/SEAMS59076.2023.00031

[29] Corbato, C.H.: Model-based self-awareness patterns for autonomy. PhD thesis, Universidad Politécnica de Madrid (2013)

[30] Hernández, C., Bermejo-Alonso, J., Sanz, R.: A self-adaptation framework based on functional knowledge for augmented autonomy in robots. Integr. Comput. Aided Eng. **25**(2), 157–172 (2018) https://doi.org/10.3233/ICA-180565

[31] Cui, Y., Voyles, R.M., Lane, J.T., Mahoor, M.H.: ReFrESH: A self-adaptation framework to support fault tolerance in field mobile robots. In: 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2014, Chicago, IL, USA, September 14-18, 2014, pp. 1576–1582. IEEE, USA (2014). https://doi.org/10.1109/IROS.2014.6942765

[32] Alberts, E., Gerostathopoulos, I., Stoico, V.,

Lago, P.: ReBeT: Architecture-based self-adaptation of robotic systems through behavior trees. In: IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS 2024, Aarhus, Denmark, September 16-20, 2024, pp. 1–10. IEEE, USA (2024). https://doi.org/10.1109/ACSOS61780.2024.00018

[33] Litoiu, M., Woodside, C.M., Zheng, T.: Hierarchical model-based autonomic control of software systems. ACM SIGSOFT Softw. Eng. Notes **30**(4), 1–7 (2005) https://doi.org/10.1145/1082983.1083071

[34] Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: Briand, L.C., Wolf, A.L. (eds.) International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA, pp. 259–268. IEEE, USA (2007). https://doi.org/10.1109/FOSE.2007.19

[35] Weyns, D., Schmerl, B.R., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., Göschka, K.M.: On patterns for decentralized control in self-adaptive systems. In: Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II - International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers. Lecture Notes in Computer Science, vol. 7475, pp. 76–107. Springer, Cham (2010). https://doi.org/10.1007/978-3-642-35813-5_4

[36] Weyns, D., Malek, S., Andersson, J.: FORMS: unifying reference model for formal specification of distributed self-adaptive systems. ACM Trans. Auton. Adapt. Syst. **7**(1), 8–1861 (2012) https://doi.org/10.1145/2168260.2168268

[37] Weyns, D., Iftikhar, M.U., Hughes, D., Matthys, N.: Applying architecture-based adaptation to automate the management of Internet-of-Things. In: Proc. 12th European Conference on Software Architecture,. Lecture Notes in Computer Science, vol. 11048, pp. 49–67. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00761-4_4

[38] Alfonso, I., Garcés, K., Castro, H., Cabot, J.: Modeling self-adaptative IoT architectures. In: ACM/IEEE Int. Conference on Model-Driven Engineering Languages and Systems Companion, pp. 761–766. IEEE, New York (2021). https://doi.org/10.1109/MODELS-C53483.2021.00122

[39] Weber, B., Rinderle, S., Reichert, M.: Change patterns and change support features in process-aware information systems. In: CAiSE 2007. Lecture Notes in Computer Science, vol. 4495, pp. 574–588. Springer, Cham (2007). https://doi.org/10.1007/978-3-540-72988-4_40

[40] Aalst, W.M.P., Pesic, M., Schonenberg, H.: Declarative workflows: Balancing between flexibility and support. Comput. Sci. Res. Dev. **23**(2), 99–113 (2009) https://doi.org/10.1007/S00450-009-0057-9

[41] Kapuruge, M., Han, J., Colman, A.: Adaptation management. In: Kapuruge, M., Han, J., Colman, A. (eds.) Service Orchestration As Organization, pp. 135–155. Elsevier, Boston (2014). https://doi.org/10.1016/B978-0-12-800938-3.00006-0

[42] Kamburjan, E., Klungre, V.N., Schlatte, R., Tapia Tarifa, S.L., Cameron, D., Johnsen, E.B.: Digital twin reconfiguration using asset models. In: Leveraging Applications of Formal Methods, Verification and Validation. Practice - 11th International Symposium ISoLA. LNCS, vol. 13704, pp. 71–88. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-19762-8_6

[43] Gil, S., Kamburjan, E., Talasila, P., Larsen, P.G.: An architecture for coupled digital twins with semantic lifting. Software and System Modeling (2024) https://doi.org/10.1007/s10270-024-01221-w

[44] Kamburjan, E., Johnsen, E.B.: Knowledge structures over simulation units. In: ANNSIM 2022, pp. 78–89. IEEE, USA (2022). https://doi.org/10.23919/ANNSIM55834.

2022.9859490

[45] Bencomo, N., Garcia Paucar, L.H.: 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), pp. 216–226 (2019). https://doi.org/10.1109/MODELS.2019.00005

[46] Bencomo, N., Götz, S., Song, H.: Models@run.time: a guided tour of the state of the art and research challenges. Softw. Syst. Model. **18**(5), 3049–3082 (2019) https://doi.org/10.1007/s10270-018-00712-x

[47] Blair, G., Bencomo, N., France, R.B.: Models@ run. time. Computer **42**(10), 22–27 (2009) https://doi.org/10.1109/MC.2009.326

[48] Bibow, P., Dalibor, M., Hopmann, C., Mainz, B., Rumpe, B., Schmalzing, D., Schmitz, M., Wortmann, A.: Model-driven development of a digital twin for injection molding. In: CAiSE 2020. Lecture Notes in Computer Science, vol. 12127, pp. 85–100. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-49435-3_6

[49] Feng, H., Gomes, C., Gil, S., Mikkelsen, P.H., Tola, D., Larsen, P.G., Sandberg, M.: Integration of the MAPE-K loop in digital twins. In: ANNSIM, pp. 102–113 (2022). https://doi.org/10.23919/ANNSIM55834.2022.9859489 . IEEE

[50] Feichtinger, K., Kegel, K., Pascual, R., Aßmann, U., Beckert, B., Reussner, R.H.: Towards formalizing and relating different notions of consistency in cyber-physical systems engineering. In: Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems, MODELS Companion 2024, Linz, Austria, September 22-27, 2024, pp. 915–919. ACM, USA (2024). https://doi.org/10.1145/3652620.3688565

[51] Pascual, R., Beckert, B., Ulbrich, M., Kirsten, M., Pfeifer, W.: Formal foundations of consistency in model-driven development. In: ISoLA 2024. Lecture Notes in Computer Science, vol. 15221, pp. 178–200. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-75380-0_11

[52] Abbasi, F., Pruski, C., Sottet, J.-S.: Semantic drift evaluation in language and data-specific digital twin frameworks. Future Generation Computer Systems, 108240 (2025) https://doi.org/10.1016/j.future.2025.108240

[53] Lehner, D., Pfeiffer, J., Tinsel, E., Strljic, M.M., Sint, S., Vierhauser, M., Wortmann, A., Wimmer, M.: Digital twin platforms: Requirements, capabilities, and future prospects. IEEE Softw. **39**(2) (2022) https://doi.org/10.1109/MS.2021.3133795

[54] Josifovska, K., Yigitbas, E., Engels, G.: Reference framework for digital twins within cyber-physical systems. In: Bures, T., Schmerl, B.R., Fitzgerald, J.S., Weyns, D. (eds.) Software Engineering for Smart Cyber-Physical Systems, SEsCPS@ICSE 2019. IEEE / ACM, New York (2019). https://doi.org/10.1109/SESCPS.2019.00012

[55] Lehner, D., Pfeiffer, J., Klikovits, S., Wortmann, A., Wimmer, M.: A method for template-based architecture modeling and its application to digital twins. J. Object Technol. **23**(3) (2024) https://doi.org/10.5381/JOT.2024.23.3.A8

[56] Tong, X., Bao, J., Tao, F.: Co-evolutionary digital twins: A multidimensional dynamic approach to digital engineering. Adv. Eng. Informatics **61**, 102554 (2024) https://doi.org/10.1016/J.AEI.2024.102554

[57] France, R.B., Rumpe, B.: Model-based lifecycle management of software-intensive systems, applications, and services. Softw. Syst. Model. **12**(3), 439–440 (2013) https://doi.org/10.1007/S10270-013-0362-4

[58] Mertens, J., Klikovits, S., Bordeleau, F., Denil, J., Haugen, Ø.: Continuous evolution of digital twins using the DarTwin notation. Softw. Syst. Model. **24**(5), 1405–1426 (2025) https:

//doi.org/10.1007/S10270-024-01216-7

[59] Esterle, L., Gomes, C., Frasheri, M., Ejersbo, H., Tomforde, S., Larsen, P.G.: Digital twins for collaboration and self-integration. In: ACSOS 2021, pp. 172–177. IEEE, USA (2021). https://doi.org/10.1109/ACSOS-C52956.2021.00040

[60] Goldsby, H.J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Hughes, D.: 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), pp. 36–45 (2008). https://doi.org/10.1109/ECBS.2008.22

[61] Grau, B.C., Horrocks, I., Motik, B., Parsia, B., Patel-Schneider, P.F., Sattler, U.: OWL 2: The next step for OWL. J. Web Semant. **6**(4), 309–322 (2008) https://doi.org/10.1016/J.WEBSEM.2008.05.001

[62] Horridge, M., Drummond, N., Goodwin, J., Rector, A.L., Stevens, R., Wang, H.: The Manchester OWL syntax. In: OWLED. CEUR Workshop Proceedings, vol. 216. CEUR-WS.org, Aachen, Germany (2006)

[63] Karabulut, E., Pileggi, S.F., Groth, P., Degeler, V.: Ontologies in digital twins: A systematic literature review. Future Generation Computer Systems **153**, 442–456 (2024) https://doi.org/10.1016/j.future.2023.12.013

[64] Zheng, X., Lu, J., Kiritsis, D.: The emergence of cognitive digital twin: vision, challenges and opportunities. Int. J. Prod. Res. **60**(24), 7610–7632 (2022) https://doi.org/10.1080/00207543.2021.2014591

[65] Kamburjan, E., Pferscher, A., Schlatte, R., Sieve, R., Tapia Tarifa, S.L., Johnsen, E.B.: Semantic reflection and digital twins: A comprehensive overview. In: Hinchey, M., Steffen, B. (eds.) The Combined Power of Research, Education, and Dissemination - Essays Dedicated to Tiziana Margaria on the Occasion of Her 60th Birthday. Lecture Notes in Computer Science, vol. 15240, pp. 129–145. Springer, Cham (2025). https://doi.org/10.1007/978-3-031-73887-6_11

[66] Bader, S.R., Maleshkova, M.: The semantic asset administration shell. In: SEMANTiCS. LNCS, vol. 11702, pp. 159–174. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-33220-4_12

[67] Sahlab, N., Kamm, S., Müller, T., Jazdi, N., Weyrich, M.: Knowledge graphs as enhancers of intelligent digital twins. In: 4th IEEE International Conference on Industrial Cyber-Physical Systems ICPS, pp. 19–24. IEEE, USA (2021). https://doi.org/10.1109/ICPS49255.2021.9468219

[68] Li, Y., Chen, J., Hu, Z., Zhang, H., Lu, J., Kiritsis, D.: Co-simulation of complex engineered systems enabled by a cognitive twin architecture. International Journal of Production Research **60**(24), 7588–7609 (2022) https://doi.org/10.1080/00207543.2021.1971318

[69] Lu, J., Zheng, X., Gharaei, A., Kalaboukas, K., Kiritsis, D.: Cognitive twins for supporting decision-makings of internet of things systems, 105–115 (2020) https://doi.org/10.1007/978-3-030-46212-3_7

[70] Ren, Z., Shi, J., Imran, M.: Data evolution governance for ontology-based digital twin product lifecycle management. IEEE Trans. Ind. Informatics **19**(2), 1791–1802 (2023) https://doi.org/10.1109/TII.2022.3187715

[71] Abburu, S., Berre, A.J., Jacoby, M., Roman, D., Stojanovic, L., Stojanovic, N.: COGNITWIN - hybrid and cognitive digital twins for the process industry. In: 2020 IEEE International Conference on Engineering, Technology and Innovation ICE/ITMC, pp. 1–8 (2020). https://doi.org/10.1109/ICE/ITMC49519.2020.9198403

[72] Ghanadbashi, S., Safavifar, Z., Taebi, F., Golpayegani, F.: Handling uncertainty in self-adaptive systems: an ontology-based reinforcement learning model. J. Reliab. Intell. Environ. **10**(1), 19–44 (2024) https://doi.org/10.1007/S40860-022-00198-X

[73] Abbasi, F., Brimont, P., Pruski, C., Sottet, J.-S.: Understanding semantic drift in model driven digital twins. In: 1st Conference on Engineering of Digital Twins (EDTConf), pp. 419–430. Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3652620.3688256

[74] Feldmann, S., Herzig, S.J.I., Kernschmidt, K., Wolfenstetter, T., Kammerl, D., Qamar, A., Lindemann, U., Krcmar, H., Paredis, C.J.J., Vogel-Heuser, B.: A comparison of inconsistency management approaches using a mechatronic manufacturing system design case study. In: 2015 IEEE International Conference on Automation Science and Engineering (CASE), vol. 2015-October, pp. 158–165. IEEE, USA (2015). https://doi.org/10.1109/CoASE.2015.7294055

[75] Feldmann, S., Herzig, S.J.I., Kernschmidt, K., Wolfenstetter, T., Kammerl, D., Qamar, A., Lindemann, U., Krcmar, H., Paredis, C.J.J., Vogel-Heuser, B.: Towards effective management of inconsistencies in model-based engineering of automated production systems. IFAC-PapersOnLine **48**, 916–923 (2015) https://doi.org/10.1016/J.IFACOL.2015.06.200

[76] Herzig, S.J.I., Qamar, A., Paredis, C.J.J.: An approach to identifying inconsistencies in model-based systems engineering. Procedia Computer Science **28**, 354–362 (2014) https://doi.org/10.1016/J.PROCS.2014.03.044

[77] Muctadir, H.M., Cleophas, L., Brand, M.: Maintaining consistency of digital twin models: Exploring the potential of graph-based approaches. In: 50th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2024, Paris, France, August 28-30, 2024, pp. 152–159. IEEE, USA (2024). https://doi.org/10.1109/SEAA64295.2024.00031

[78] Feldmann, S., Wimmer, M., Kernschmidt, K., Vogel-Heuser, B.: A comprehensive approach for managing inter-model inconsistencies in automated production systems engineering. In: 2016 IEEE International Conference on Automation Science and Engineering (CASE), pp. 1120–1127 (2016). https://doi.org/10.1109/COASE.2016.7743530

[79] Michael, J., David, I., Bork, D.: Digital twin evolution for sustainable smart ecosystems. In: MoDELS (Companion), pp. 1061–1065. ACM, New York, NY, USA (2024). https://doi.org/10.1145/3652620.3688343

[80] David, I., Bork, D.: Towards a taxonomy of digital twin evolution for technical sustainability. In: ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2023 Companion, Västerås, Sweden, October 1-6, 2023, pp. 934–938. IEEE, USA (2023). https://doi.org/10.1109/MODELS-C59198.2023.00147

[81] Balaji, B., Bhattacharya, A.A., Fierro, G., Gao, J., Gluck, J., Hong, D., Johansen, A., Koh, J., Ploennigs, J., Agarwal, Y., Berges, M., Culler, D.E., Gupta, R.K., Kjærgaard, M.B., Srivastava, M.B., Whitehouse, K.: Brick: Towards a unified metadata schema for buildings. In: International Conference on Systems for Energy-Efficient Built Environments, BuildSys@SenSys 2016. ACM, New York (2016). https://doi.org/10.1145/2993422.2993577

[82] Abanda, F.H., Akintola, A., Tuhaise, V.V., Tah, J.H.M.: BIM ontology for information management (BIM-OIM). Journal of Building Engineering **107**, 112762 (2025) https://doi.org/10.1016/j.jobe.2025.112762

[83] Daniele, L., Hartog, F.T.H., Roes, J.: Created in close interaction with the industry: The smart appliances reference (SAREF) ontology. In: Cuel, R., Young, R. (eds.) Formal Ontologies Meet Industry - 7th International Workshop, FOMI 2015. Lecture Notes in Business Information Processing, vol. 225. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21545-7_9

[84] Qureshi, N.A., Jureta, I., Perini, A.: Requirements engineering for self-adaptive systems: Core ontology and problem statement. In: Advanced Information Systems Engineering - 23rd International Conference, CAiSE 2011, London, UK, June 20-24, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6741, pp. 33–47. Springer, Cham (2011). https://doi.org/10.1007/978-3-642-21640-4_5

[85] Gerevini, A., Long, D.: Plan constraints and preferences in PDDL3. Technical report, Technical Report 2005-08-07, Department of Electronics for Automation. (2005)

[86] Smith, B.: Mereotopology: A theory of parts and boundaries. Data and Knowledge Engineering **20**(3), 287–303 (1996) https://doi.org/10.1016/S0169-023X(96)00015-8

[87] Sattler, U., Schneider, T., Zakharyaschev, M.: Which kind of module should I extract? In: Description Logics. CEUR, vol. 477 (2009)

[88] Skjæveland, M.G., Lupp, D.P., Karlsen, L.H., Forssell, H.: Practical ontology pattern instantiation, discovery, and maintenance with reasonable ontology templates. In: Proc. 17th International Semantic Web Conference (ISWC 2018). LNCS, vol. 11136, pp. 477–494. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00671-6_28

[89] Skjæveland, M.G., Karlsen, L.H.: The reasonable ontology templates framework. TGDK **2**(2), 1–54 (2024) https://doi.org/10.4230/TGDK.2.2.5

[90] W3C, SHACL Working Group: Shapes Constraint Language. https://www.w3.org/TR/shacl/

[91] Koopmann, P.: Signature-based abduction with fresh individuals and complex concepts for description logics. In: Proc. Thirtieth International Joint Conference on Artificial Intelligence (IJCAI 2021), pp. 1929–1935. ijcai.org, California (2021). https://doi.org/10.24963/ijcai.2021/266

[92] Shvaiko, P., Euzenat, J.: Ontology matching: State of the art and future challenges. IEEE Trans. Knowl. Data Eng. **25**(1), 158–176 (2013) https://doi.org/10.1109/TKDE.2011.253

[93] Beaumont, G.G., Beugnard, A., Martínez, S., Urtado, C., Vauttier, S.: Towards automating the life cycle management of digital twins. In: Bork, D., Lukyanenko, R., Sadiq, S., Bellatreche, L., Pastor, O. (eds.) ER 2025. Lecture Notes in Computer Science, vol. 16189, pp. 412–430. Springer, Cham (2025). https://doi.org/10.1007/978-3-032-08623-5_22

[94] Sieve, R., Kamburjan, E., Damiani, F., Johnsen, E.B.: Declarative dynamic object reclassification. In: Aldrich, J., Silva, A. (eds.) ECOOP 2025. LIPIcs, vol. 333, pp. 29–12931. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl (2025). https://doi.org/10.4230/LIPICS.ECOOP.2025.29