Ruleless Digital Twins

Ivan Spajić[0009−0002−2888−4656] and Volker Stolz[0000−0002−1031−6936]

Western Norway University of Applied Sciences, Bergen, Norway
{ispa,vsto}@hvl.no

Abstract. We introduce a transparent digital twin (DT) decision-making approach without the use of explicit decision rules or rule-based models. Our approach utilizes ontological inference and simulation models to explore possible decisions before applying them to the twinning target. As proof of concept, we provide an implementation of the proposed framework purely based on widely-known technologies and standards, and subsequently demonstrate the feasibility of our approach. We discuss benefits and drawbacks, and recognize that a *ruleless* approach to DT decision-making ultimately rests on an effective method of choosing from a multitude of possibilities. Lastly, we consider potential future work and exploration in the context of more effective automation and simulation model usage.

Keywords: digital twins · IoT · ontologies · inference · simulation · FMUs · MAPE-K

1 Introduction

Digital twins (DTs) are a domain overlapping with various other ICT fields, such as the Internet of Things (IoT), modeling, simulation, runtime verification, and many others [16]. At their core, DTs consist of a digital system traditionally acting as a model for a physical one, often regarded as the physical twin (PT), particularly in contexts of cyber-physical systems (CPSs). This notion has, however, become more generalized whereby DTs are no longer used exclusively for modeling PTs. Instead, they have developed to reflect what are now more broadly referred to as twinning targets (TTs) [18].

According to Larsen *et al.* [16], DTs add value to their PTs[1] without unduly compromising their operations. To meet this goal, a DT includes a level of connectivity that allows data to be received from and transmited to its TT [15]. In CPSs, this is typically manifested in the context of a sensor-actuator network (SAN). In such scenarios, data is gathered by sensors on the PT side and sent to a smart node capable of processing it on the DT side. Thereafter, based on some logic or rules, certain actions are taken and commands are transmitted back to the actuators on the PT side. The same process generalizes beyond CPSs and SANs, although instead of data only being transmitted between a DT and its

[1] Although they use the term "PT", it should be noted that Larsen *et al.* do not limit themselves to DTs modeling physical objects.

physical devices, such as sensors and actuators, it may also be transmitted in a pure cyber context. In such cases, DTs may also be used for optimizing or reconfiguring processes [16], networking, or software [9] in general.

To facilitate their data processing and decision-making, various kinds of models are essential for DTs [16,18]. Apart from architectural [13] and data models [5], they often also depend on simulations [3] used for creating hypothetical scenarios whose results are subsequently utilized in making actuation or reconfiguration decisions for the TT. Moreover, DTs may use ontological knowledge graphs for conceptual modeling, which allows for semantic data validation and additional reasoning techniques [4,19]. Furthermore, for increased accuracy, numerous IoT and DT decision-making processes utilize rule-based models [20,23,31], many of which typically follow an `IF-THEN` pattern.

Perhaps somewhat obviously, a DT will always require models containing information about its TT's structure and behavior. In addition, given numerous examples of DTs making use of ontologies, simulations, and behavioral rule models, as well as an abundance of model-driven techniques in the literature [18], there is a naturally-arising question concerning the possibility of using ontological reasoning with simulations to power decision-making processes without behavioral rule models. Thus, our paper explores this question as follows: in section 2, we present a more specific context from related works in the literature as well as our own contributions. In section 3, we outline our idea on a conceptual level, and in 4, we describe its implementation and proof of concept. In section 5, we discuss drawbacks and other potential approaches, and in 6, we introduce ideas for future work. Lastly, we conclude in section 7.

2 Background

As outlined in Larsen *et al.* [16], models are vital to DTs. Furthermore, as highlighted by Lehner *et al.* [18], model-driven engineering (MDE) techniques additionally yield various benefits within the general field of DTs. For instance, models can be used in model-to-model transformations, where source and target models adhere to different hosting platforms. Moreover, one may also apply model-to-model transformations to more generative purposes, such as executable code generation. Regardless of the specific goal, it is evident that MDE techniques carry a shared aspect of increased automation. For this reason, we study the extent of this automation in the context of ontological inference, simulations, and rule-based models throughout the exploration of our own problem statement and related work.

2.1 Related Work

Gabor *et al.* [7] partition the different levels of control exerted by a DT into several tiers. Our work directly addresses their second tier for "immediate reaction", where actions are triggered by system state observations, and enables the third tier of "planned reaction", where reward functions steer planning.

Talasila *et al.* [25] introduce the "DT-as-a-Service" with valuable insights about composing new DTs. While they identify simulation ("`WHAT-IF`" analysis) as a key requirement, they do not explore decision planning in detail.

Kamburjan *et al.* [12] reflect their *program state* into a knowledge base to use the expressivity of ontologies. We take the opposite approach and initially realize several of the necessary components entirely in the knowledge base.

Compagnucci *et al.* [4] utilize the Semantic Sensor Network (SSN) ontology [29] for semantic data validation and class mapping to input models used for DT code generation. They make use of finite-state machines as rule-based models for defining the internal logic of their generated DTs. Although they acknowledge the potential of using ontological inference from SSN, Compagnucci *et al.* ultimately do not use this type of reasoning for any kind of automated rule or rule-based model generation.

Korth *et al.* [14] demonstrate DTs in the context of decision support systems (DSSs) based on digital models and simulations comprising an event-driven architecture. Crucially, our approach shares their list of five criteria for good decisions and planning: (*i*) current state knowledge, (*ii*) future situation (hypothetical) knowledge, (*iii*) functions identifying disruptions, (*iv*) functions generating (feasible) decisions, and (*v*) functions evaluating these decisions. Although their DTs demonstrate a level of decision support by investigating hypothetical scenarios through simulations, they leave full decision-making to the users.

Michael *et al.* [20] present a model-driven software architecture for DT generation. For a DT to accurately plan and make its decisions, they propose the use of case-based reasoning models in the form of `{IF pattern THEN actions EXPECT result}`. They provide examples of parameterized commands containing conditions for the DT. With this, the DT attempts to find a matching system condition case from an available repository of case-based models supplied by domain experts. If the case is applicable to the observed system conditions, its solution is applied. Otherwise, the DT's reasoner chooses a similar case and adjusts it. When the adjusted case mitigates the situation, it is added to the repository of cases. Additionally, Michael *et al.* focus on the aspect of explainability whereby a system should transparently explain its decision-making processes. To achieve this, they use a feedback loop framework similar to MAPE-K [24]. Although they rely on explicit rule models, they leave the rule adjustment details of their example cases outside of the scope of their work.

While they do not strictly focus on DTs, some of the more related work comes from Päßler *et al.* [23], who present a formal model of an architecture for controlling an IoT-based smart home solution. They define decision-making models and, through simulation experiments, show improved performance of the system's ability to satisfy its requirements with respect to regulating air temperature and quality. They utilize all MAPE-K phases to perform different tasks, such as the *Analyze* phase for checking current system configuration satisfiability and the *Plan* phase for potentially choosing system reconfigurations. An interesting point to note is that for the *Knowledge* portion of the MAPE-K, Päßler *et al.* rely on a formal, ontological meta-model that describes elements

such as *objectives* and *functions*. In sub-section 4.1, we will demonstrate how our ontological contributions (interestingly) represent similar concepts.

2.2 Contributions

Having highlighted some of the relevant literature with respect to the use of rules in DT and IoT decision-making processes, we will now outline the primary and secondary contributions of this paper.

Primary contribution: We demonstrate the feasibility of *ruleless* DTs through ontological reasoning and simulations for powering autonomous decision-making processes. By removing the need for manually crafting and maintaining hand-written rule models, we also show increased automation and reduced complexity, albeit at the expense of greater computational cost.

Secondary contributions: We present lightweight additions to the SSN ontology to facilitate the primary contribution and further demonstrate how this ontology already contains most of the required concepts. Additionally, we present this idea in the context of only using widely-known and well-accepted technologies and standards. Lastly, as part of an available artifact[2], we provide a generic control loop solution that conforms to our proposed ontology and is complete with examples and interfaces intended for additional user-provided implementations. This solution demonstrates autonomous DT decision-making by evaluating system condition constraints, finding available mitigations, simulating hypothetical scenarios, and, as a result, picking optimal actions.

3 Approach

Fig. 1 visualizes an architectural perspective of all of the components of our proposed system. Our approach begins with using an ontology to infer additional model data from what was originally provided by the stakeholder. Since a DT requires models that describe the structure and behavior of its TT anyway, the idea is to let stakeholders create these in the form of knowledge graphs that conform to the ontology as instance models. The ontology thus requires concepts that allow stakeholders to describe sensors and actuators, in case of physical TT components, and soft sensors [11] and configurable software parameters, in case of cyber ones. In both cases, stakeholders would also need to describe their properties of interest, which can be observed and/or actuated on. For this, we ultimately envision stakeholders using a form of automated model export in conjunction with tools such as Protégé [21].

The ontology should allow the instance models to contain desirable conditions for the DT to maintain, e.g., `{18 <= roomTemperature < 25}`. In addition, to

[2] The ontology, inference rules, example instance models, simulation model, and the feedback loop solution are all available at https://doi.org/10.5281/zenodo.18741472.

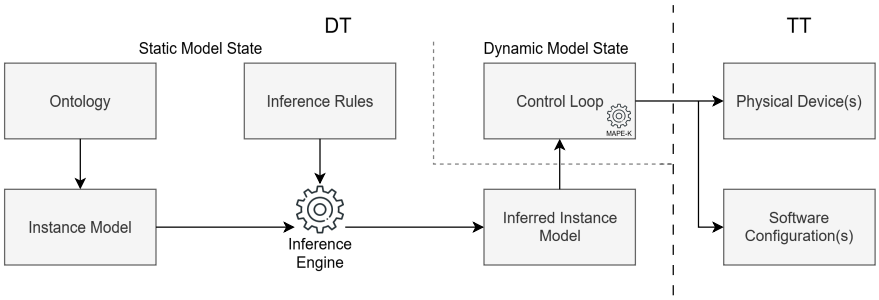

Fig. 1. An overview of the proposed system's components. Here, the inference rules and the control loop rely on the ontology.

express the power of DTs adding value to their TTs [16], they should also specify optimizations (e.g., minimizing energy consumption), since, for example, given a desired condition as `{roomTemperature > 18}`, the system could simply keep the heater on the warmest setting for as long as possible to satisfy it.

Most importantly, as part of the semantic descriptions of the actuators and reconfigurable parameters, it is also necessary for the ontology to provide different types of effects these elements can have on their respective properties. By knowing *how* an action changes a property, we can subsequently infer all relevant actions the DT can take to potentially restore its desirable conditions. For example, if a stakeholder is interested in observing a room temperature property and defines a heater as an actuator that (generally) increases its value, then, should the temperature drop below desired levels, it becomes inferrable that actuating the heater might mitigate the problem.

Further following fig. 1, the instance model is run through an inference engine to produce an inferred (expanded) one. In addition to the initial TT component descriptions, their respective properties, and the desired conditions, the inferred model will now also contain elements representing all possible optimization actions as well as actions for mitigating any unsatisfied conditions.

Next, this initial inferred model is supplied into the control loop component during its initialization. The control loop is responsible for controlling physical devices and software configurations on the TT side, and effectively acts as the DT's processing center. Since DTs need to handle both data input and output in a feedback fashion, we propose for the control loop to follow the MAPE-K [24] architecture. This allows us to organize different DT processing phases into the corresponding phases of the MAPE-K. We envision the loop to function as detailed in fig. 2: first, the *Monitor* phase handles value observations for the specified properties. With the observed property values, the *Analyze* phase evaluates which, if any, desired conditions are unsatisfied and selects a set of all possible mitigation and optimization actions from the knowledge base (initial inferred model). From this set of actions, the *Plan* phase constructs and

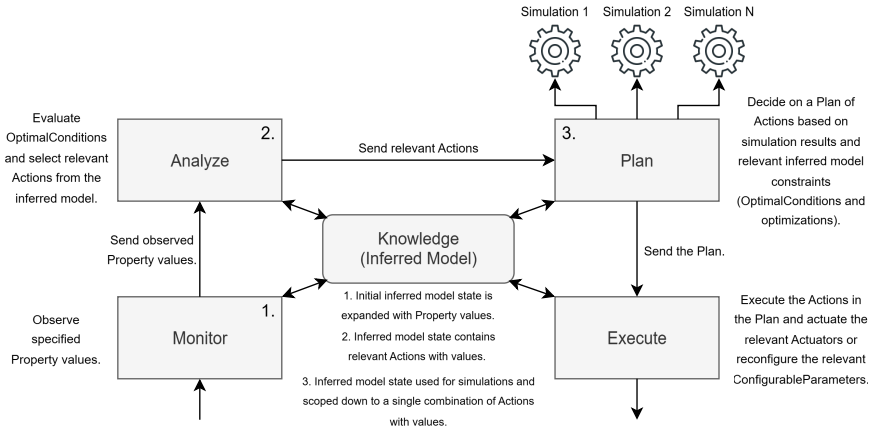

Fig. 2. A visualization of the parts comprising the control loop component based on the MAPE-K [24] architecture.

executes all possible simulation configurations, which may be split into several intervals. This approach allows a simulation configuration to change component states throughout various stages of its execution. A fitness function identifies the optimal simulation's combination of actions which the *Execute* phase enacts.

Throughout the different stages in this process, the framework uses the stakeholder-provided instance model to expand or filter its data. Initially, the static instance model is dynamically expanded with the set of inferred actions and then reduced by evaluating conditions and selecting possible mitigations. The model is dynamically expanded again with all possible simulation configurations before being reduced to a single combination of concrete actions.

In summary, our overall approach of *ruleless* DT decision-making can be represented analogously with a tree of possibilities, as visualized in fig. 3. In this tree, each path to a leaf node represents a sequence of possible combinations of actions the DT can choose to attempt to restore a condition and/or to optimize for certain properties, whereby the DT's ultimate goal is to prune the tree of possibilities down to a single option. Initially, with actions inferred from the instance model, the number of possible decisions grows. By analyzing which, if any, desired conditions are unsatisfied, the DT selects the subset of actions that positively affect the model. From these, the DT constructs simulation combinations, and the number of possible decisions grows again. Finally, a simulation fitness function prunes the tree, ultimately yielding the DT's final decision.

3.1 Technologies and Standards

Firstly, we require an ontology that can semantically represent IoT-relevant concepts such as hosting systems, physical devices, software or soft sensors, and ob-

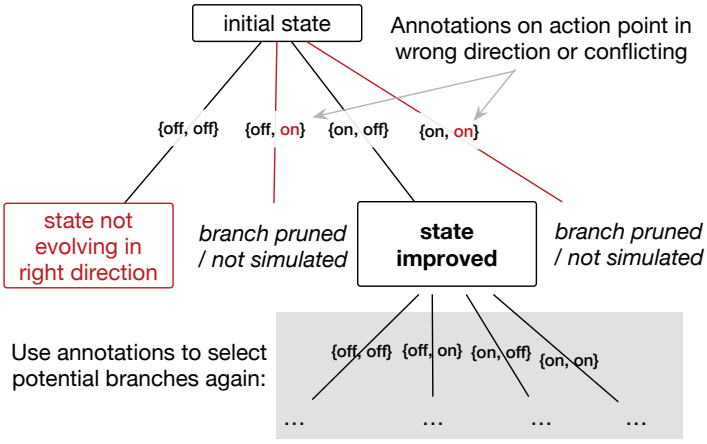

Fig. 3. The tree of possible decisions the DT can take based on the number of inferred actions and possible simulation configurations.

servable properties. For this, we recognize the Semantic Sensor Network (SSN) and Smart Applications REFerence (SAREF) [6] ontologies as some of the more established and widely-used candidates [4,8,22]. Of the two, we evaluate SSN to be more representative of what is needed for our approach, based on the lower number of missing required concepts. Although SAREF Core provides numerous extension ontologies that could provide some of those concepts, it already contains 33 classes, 64 object properties, and 8 data properties. Conversely, while SSN is itself an extension of the Sensor, Observation, Sample, and Actuator (SOSA) [29] ontology, it only contains 23 classes, 36 object properties, and 2 data properties, making it the more light-weight option.

Among inference engines, we opted to use Apache Jena [26] as it is one of the most widely-known open-source technologies in the field with a long history and wide-ranging standard feature support. Apart from allowing for different levels of inference, Jena's API also provides model-checking features for additional verification against inferred models breaking ontological constraints.

We selected the Functional Mock-up Interface (FMI) [27] standard to power our simulations due to its wide-spread industry support and increasing usage in the field of DTs [1,10]. Our approach relies on the use of Functional Mock-up Units (FMUs) as self-contained, executable simulation models for creating relevant hypothetical scenarios. Moreover, apart from wide-spread adoption, FMUs are fairly lightweight components and thus exhibit a high degree of execution performance.

Lastly, to interact with RDF/OWL knowledge graphs and execute FMUs, we utilize the dotNetRDF [30] and Femyou [2] .NET libraries, respectively.

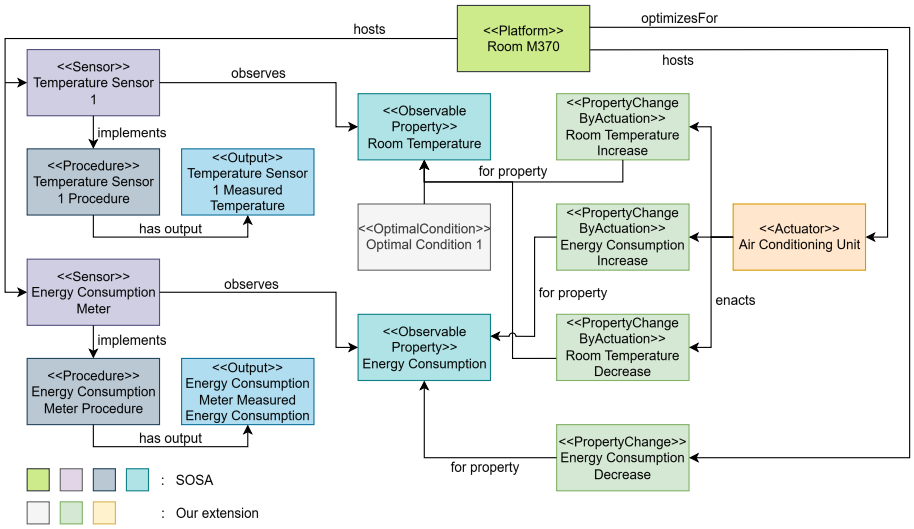

Fig. 4. Some of example 1's components annotated with their classes. Note that *sosa:ObservableProperty* is a subclass of *ssn:Property*.

4 Implementation

We designed two examples for our proof of concept: example 1 (shown partially in fig. 4) describes a room containing sensors measuring temperature, humidity, and energy consumption, as well as actuators for affecting each property. As such, it is representative of a TT containing physical components as part of a CPS. Example 2 (in the digital artifact) describes a soft sensor using an online lossy compression algorithm with configurable parameters for adjusting the input data size and the degree of lossiness, and is therefore representative of a TT containing cyber elements. Throughout this paper, we mostly use example 1 as a case for demonstrating the different stages of our implementation.

4.1 Ontology

As outlined in section 3, although we make thorough use of SSN, this ontology still lacks concepts crucial to our approach. Hence, we introduce a lightweight extension by adding classes required to complete stakeholder-provided instance models as well as the resulting inferred ones.

```
// OptimalCondition
'for property' exactly 1 Property
hasValueConstraint exactly 1 rdfs:Literal
reachedInMaximumSeconds max 1 xsd:int
```

Listing 4.1.1. *:OptimalCondition* class definition triples in Manchester syntax.

An important missing concept is that of desired conditions on targeted property values. Here, we introduce the *:OptimalCondition* class whose definition

triples are shown in listing 4.1.1. An *:OptimalCondition* requires three things: (*i*) an *ssn:Property*, (*ii*) constraints on its value, and (*iii*) an upper limit on the time taken to restore it, should the value of the respective *ssn:Property* fall out of bounds.

Moreover, to facilitate cyber TTs, we add a *:ConfigurableParameter* class as a subclass of *ssn:Property* and *ssn:Input*. *:ConfigurableParameters* represent adjustable soft sensor inputs used to affect their algorithms and subsequent *ssn:Outputs*. In terms of example 2, both the size of input data and the degree of lossiness *ssn:Properties* would be *:ConfigurableParameters*.

```
// PropertyChangeByReconfiguration is a subclass of PropertyChange
'for property' exactly 1 Property
affectsPropertyWith exactly 1 Effect
alteredBy exactly 1 Effect
enactedBy min 1 ConfigurableParameter
```

Listing 4.1.2. *:PropertyChangeByReconfiguration* definition triples in Manchester syntax.

Two more classes we add are those of *:Effect* and *:PropertyChange*. The *:Effect* class in our prototype contains just two individuals, *:ValueIncrease* and *:ValueDecrease*, representing the type of change done to an *ssn:Property*. The purpose of the *:PropertyChange* class is to act as a wrapper facilitating the mapping between *ssn:Properties* and the *:Effects* applied to them. Furthermore, to support both physical and cyber TT elements, the *:PropertyChange* class has two subclasses, namely *:PropertyChangeByActuation* and *:PropertyChangeByReconfiguration*, the latter of which is shown in listing 4.1.2. These classes contain references to *sosa:Actuators* and *:ConfigurableParameters*, respectively, and thus facilitate the construction of inferred models later in the process. As shown in fig. 4, example 1 contains four *:PropertyChangeByActuation* individuals (without *Effect* references) relating to temperature and energy consumption changes.

```
:optimizesFor rdf:type owl:ObjectProperty ;
              rdfs:domain <http://www.w3.org/ns/sosa/Platform> ;
              rdfs:range :PropertyChange ;
```

Listing 4.1.3. *:optimizesFor* definition triples in OWL.

Somewhat related to the *:OptimalCondition* class, but without the hard value constraints, is the concept of targeted system optimizations. As shown in listing 4.1.3, this was introduced as the *:optimizesFor* object property, with *ssn:Platform* as its domain and *:PropertyChange* as its range. In example 1, a system optimization for lower energy consumption is defined through the relation *:RoomM370 :optimizesFor :EnergyConsumptionDecrease*. Overall, this concept enables stakeholders to specify their optimizations while also facilitating simulation result filtering, as discussed further in subsection 4.3.

```
// ActuationAction is a subclass of Action
hasActuator exactly 1 Actuator
setsActuatorToState exactly 1 rdfs:Literal
```

Listing 4.1.4. *:ActuationAction* definition triples in Manchester syntax.

Moreover, we added the *:Action* class, which defines an action the DT can take as part of its eventual decision. Individuals belonging to this class are not directly supplied by the stakeholder but are instead inferred from the definitions of *:OptimalConditions*, optimizations, and *:PropertyChanges*. Much like *:PropertyChange*, the *:Action* class has two subclasses, namely *:ActuationAction* and *:ReconfigurationAction*. Whereas the former reconfigures *physical* components, the latter is for *cyber* components. As shown in listing 4.1.4, each *:ActuationAction* contains a reference to a *sosa:Actuator* and a literal representing its state. Similarly, a *:ReconfigurationAction* contains a reference to a *:ConfigurableParameter* and a literal representing the value to configure it to.

Lastly, it should be noted that, as an adjustable input variable, a *:ConfigurableParameter* (as e.g., a float, double, etc.) can have a virtually infinite number of values. Furthermore, there may be specific limitations on its value ranges or distributions. Similarly, a *sosa:Actuator*, such as *:AirConditioningUnit* in example 1, may have a wide range of programmable states that would be unsuitable to represent as explicit instance model components. Thus, we include the *:ActionValueGenerator* class as a subclass of *ssn:Procedure* to represent a method (e.g., FMU execution, REST API call, or other) for deriving possible *sosa:Actuator* states and *:ConfigurableParameter* values, respectively.

4.2 Ontological Inference

There are two main objectives in using ontological inference rules in our system. First, inference rules are used to verify the structural invariants of the model, an example of which is shown in listing 4.2.1. Second, they are used for deriving additional information from what is initially provided by the stakeholder. To simplify both verification and additional triple creation, some inference rules leave intermediate marker comments for subsequent rules to match on.

```
[rule4e:    (?restriction1 rdf:comment ?property)
            (?restriction1 rdf:comment "verification mark")
            (?restriction1 xsd:minExclusive ?value1)
            (?restriction2 rdf:comment ?property)
            (?restriction2 rdf:comment "verification mark")
            (?restriction2 xsd:maxExclusive ?value2)
            notEqual(?property, "verification mark")
            ge(?value1, ?value2)
            -> ... print("Conflicting conditions detected!") ... ]
```

Listing 4.2.1. An example verification rule checking for conflicting *OptimalCondition* constraints based on markings applied by a previous rule.

Since the DT ultimately has to pick a set of *Actions* that restore *OptimalConditions* and/or optimize for a *Property*, it is important that these *Actions* not be contradicting. Most potential user-related constraint problems would typically be prevented by using their tools' built-in reasoners. For a single individual, they would immediately highlight problematic cases, such as simultaneously defining a constraint value to be <18 and >25. However, if stakeholders define multiple *OptimalCondition* individuals pertaining to the same *Property*, they will also be able to freely define them with mutually contradicting constraints such that

the reasoner cannot highlight the issue. Without constraint verification rules, the DT could encounter scenarios where it was forced to simultaneously simulate *Actions* with *Actuators* that apply conflicting *Effects* on the same *Property*, ultimately preventing it from arriving to a meaningful decision.

```
[rule3a:    (?thing rdf:comment ?propertyChange)
            (?propertyChange meta:enactedBy ?actuator)
            (?actuator rdf:type sosa:Actuator)
            uriConcat(?actuator, meta:ActuationAction, ?actionName)
            ->  (?actionName rdf:type meta:ActuationAction)
                (?actionName meta:hasActuator ?actuator) ]
```

Listing 4.2.2. An example rule for inferring *ActuationActions*.

To derive additional triples, the inference rules rely on instance models containing crucial information, such as the *PropertyChanges* caused by various TT components. As shown in listing 4.2.2, the inference rules match on this information and expand an instance model with the *Actions* the DT can take. Each *Action* will therefore either contain an *Actuator* or a *ConfigurableParameter* capable of causing the kind of *PropertyChange* required for restoring a specific *OptimalCondition* constraint or optimizing for a *Property*.

The rules do not, however, simply infer *Actions* as a result of any *PropertyChanges* present. In terms of the tree of possible DT decisions (see fig. 3), branches are potentially immediately pruned away during the creation of an inferred model. This is done by initially focusing on the specified *OptimalConditions'* value constraints. Given a particular type of constraint, such as an upper or lower bound, only the corresponding, mitigating *PropertyChanges* are selected and thus only a subset of all possible *Actions* is inferred.

Since example 1 contains *OptimalConditions* with upper and lower bounds for *RoomTemperature*, both the *Actuators* causing *PropertyChanges* with *ValueIncrease* and those causing *PropertyChanges* with *ValueDecrease* become relevant. Because the *AirConditioningUnit* component is responsible for both types of relevant *PropertyChanges*, an *ActuationAction* is inferred with it as its *Actuator*. Moreover, due to a specified optimization on *RoomHumidity* in the full instance model, an additional *ActuationAction* with the *Dehumidifier* is inferred. Note that, due to reasons discussed in 4.1, these inferred *ActuationActions* have no concrete *Actuator* states specified and thus serve as placeholders until the execution of their respective *ActionValueGenerators* (discussed further in 4.3).

4.3 Control Loop

In line with the MAPE-K [24] architecture, the control loop component features four main phases assisted by the inferred model as its knowledge base. Each phase interacts with the inferred model's knowledge graph through the use of SPARQL [28] queries. The control loop features interfaces for stakeholder-provided *Sensor* and *Actuator* implementations used for *Property* observations and *Action* executions, respectively. These implementations handle the DT's data input and output during the *Monitor* and *Execute* phases of the loop. The following sub-subsections outline the phases in greater detail.

Monitor Phase The *Monitor* phase invokes stakeholder-provided *Sensor* implementations and observes all relevant *Property* values to store them in a dedicated *Property* cache, thereby effectively augmenting the inferred model.

Analyze Phase As discussed in section 3 and shown in fig. 2, the overall aim of the *Analyze* phase is to further reduce the set of *Actions* used for simulating hypothetical scenarios, thereby further pruning the tree of possible decisions. To achieve this, the phase begins by querying the knowledge base (inferred model) for all *OptimalConditions* and evaluating their constraints against the respective *Property* values observed earlier.

```
SELECT ?optimalCondition ?property ?reachedInMaximumSeconds WHERE {
  ?optimalCondition rdf:type meta:OptimalCondition .
  ?optimalCondition ssn:forProperty ?property .
  ?optimalCondition meta:reachedInMaximumSeconds ?reachedInMaximumSeconds . }
```

Listing 4.3.1. The SPARQL query used for selecting all *OptimalConditions* and most of their elements. The constraints are selected with more complex, subsequent queries.

After selecting all *OptimalConditions* (listing 4.3.1), each one's constraints are evaluated against the values of their respective *Properties*. This evaluation determines whether an *OptimalCondition* is unsatisfied and thus whether an *Action* should be taken to restore it. During evaluation, each constraint is broken down into its constituent atomic constraints (e.g., upper and lower bounds) whose operators are used to determine the *Effect* and thus the *PropertyChange* required from potential mitigation *Actions*. With this, a subset of *Actions* is selected from those in the knowledge base, as shown in listing 4.3.2.

```
SELECT ?actuationAction ?actuatorState ?actuator ?property WHERE {
    ?actuationAction rdf:type meta:ActuationAction.
    ?actuationAction meta:hasActuatorState ?actuatorState .
    ?actuatorState meta:enacts ?propertyChange .
    ?actuator meta:hasActuatorState ?actuatorState .
    ?actuator rdf:type sosa:Actuator .
    ?propertyChange ssn:forProperty ?property .
    ?property owl:sameAs @property .
    ?propertyChange meta:affectsPropertyWith @filter . }
```

Listing 4.3.2. The SPARQL query used for selecting all mitigating *ActuationActions* with a `filter` parameter representing the *Effect*.

As outlined in subsection 4.2, the knowledge base initially only contains placeholder *Actions* without specified *Actuator* states or *ConfigurableParameter* values. To obtain concrete *Actions* with specific states and values, stakeholder-provided *ActionValueGenerators* are executed for every *Action's Actuator* or *ConfigurableParameter*. Assuming example 1's *OptimalConditions* are unsatisfied, the relevant mitigation *Action* is the one containing *AirConditioningUnit*. Executing *AirConditioningUnit's ActionValueGenerator* thus produces all possible states for the *Actuator* and subsequently yields a concrete *ActuationAction* for each one. This process is also used for optimization *Actions*.

Plan Phase The overall purpose of the *Plan* phase is to explore via simulation, within the scope of the provided mitigation and optimization *Actions*, all hypothetical scenarios possible and subsequently pick the optimal one to execute.

This process can be divided into several steps: (*i*) generate all allowed combinations of provided *Actions*, (*ii*) generate all possible configurations of simulation intervals with *Action* combinations, (*iii*) execute all simulation configurations and obtain their results, and (*iv*) select the optimal configuration.

Upon receiving potential mitigation and optimization *Actions*, all of their allowed combinations are generated such that no *Actions* in a given set share *Actuators* or *ConfigurableParameters*. Next, we generate simulation configurations, each of which represents a unique simulation scenario setup. Because different *Actuator* states might affect *Property* values differently, we divide our simulations into timed intervals to join with different *ActuationAction* combinations. This allows *Actuators* to potentially restore *OptimalConditions* through a variety of states in a given simulation. We deliberately choose not to include *ReconfigurationActions* as part of the timed simulation intervals since changes in cyber TTs depending on *ConfigurableParameters* do not necessarily occur as time-dependent processes. Updated *ConfigurableParameter* values may produce results instantly, or they may take longer than the *OptimalConditions* allow, due to, e.g., cyber implementation details. Thus, combinations of *ReconfigurationActions* are appended to simulation configurations as post-simulation *Actions*. With this, a unique configuration is generated for every combination of *ActuationActions* in every interval and with every combination of post-simulation *ReconfigurationActions*. In example 1, for 4 *AirConditioningUnit* states, 2 *Dehumidifier* states, and a simulation granularity (depth of tree) of 4, a total of 4096 unique simulation configurations are generated.

We retrieve the simulation model (FMU) URI from the knowledge base. Although our example uses a stakeholder-provided URI, it could alternatively be derived through a query. This model represents the hosting *Platform* and thus needs to comprise its *Actuators* as input parameters, as well as all of its system *Properties* as both input and output parameters. With these FMUs, all simulation configurations are executed based on values in the *Property* cache (current state of the model), whereby for each simulation interval, *Actuator* states and *Properties* are supplied, the simulation time is advanced to the duration of the interval, and the resulting *Property* values are recorded for the next step.[3]

The Fitness Function Finding the optimum effectively relies on a fitness function that successively filters simulation configurations out based on their resulting *Property* values. Our current implementation of this function contains several steps. Firstly, the function prioritizes simulation configurations that result in the most *OptimalConditions* satisfied. In case of more than one configuration sharing the highest number, the function checks their values of the *Properties* to optimize for. In this step, each *Property* the system wishes to optimize for has its value compared with the same *Property* from every other remaining simulation configuration. In case that *Property* is optimized for with *ValueIncrease*, simulation configurations with the highest value for that *Property* have their scores

[3] The simulation of *ReconfigurationActions* is not currently implemented but may easily be performed through, e.g., API calls.

incremented. Similarly, if a *Property* is optimized for with *ValueDecrease*, configurations with the lowest value for that *Property* have their scores incremented instead. Scores are compared between all remaining simulation configurations, and those with the highest total are chosen. Finally, if there are still multiple configurations remaining, the function arbitrarily chooses from the collection. Note that our fitness function currently does not address totally avoiding prohibited TT states or the exclusion of *Actions* that might lead to them, both of which are intended as future work.

Execute Phase Having identified the optimal simulation configuration, the *Execute* phase performs all *ActuationActions* by invoking the corresponding *Actuator* implementations, e.g. through API calls, and *ReconfigurationActions* by updating the *ConfigurableParameters* in the *Property* cache. These are then used for the next *Monitor* phase's soft sensor value observations.

5 Discussion

One of the biggest strengths of this approach is its flexibility to adapt to frequently-changing TT components and conditions. Because the system repeatedly queries its knowledge base, it is also resilient against changing stakeholder requirements. Unlike with a rule-based system, our approach allows stakeholders to change their *OptimalCondition* specifications and have the resulting DT decisions automatically adapt in response. Moreover, in a rule-based system, the stakeholder must provide a whole new rule (`IF-THEN`) for each new condition (the `IF`) and desired decision (the `THEN`), whereas *OptimalConditions* (the `IF`) allow inference and simulations in our approach to yield the decision (the `THEN`) automatically. Furthermore, because our approach utilizes simulations, it also effectively looks ahead into any potential future before making its decision.

Conversely, because it relies on FMUs for simulations, our approach requires that they be provided by the stakeholder beforehand. Adopting our method thus effectively creates a trade-off between the development time spent on rule-based models versus FMUs, although this is arguably mitigated due to the aforementioned simulation benefits. Moreover, because FMUs may contain default or starting values for their input parameters, it is possible to utilize them without exact parameter matching, thus adding to their flexibility at the cost of simulation accuracy. In addition, because we explore all potential scenarios within the scope of the previously inferred *Actions* and all of their possible states and values, our approach makes use of heuristics to combat combinatorial blowup, which we anticipate to be especially problematic for low-powered or embedded IoT processing devices. This blowup can be reduced further through more information about the types of possible *Effects* in the ontology. Our changes to the SSN ontology only introduced two (qualitative) individuals, namely *ValueIncrease* and *ValueDecrease*, for simplicity. Specifying more detailed *Effect* types would increase instance model complexity, but would simultaneously help in pruning down the tree of possibilities during the DT's *Plan* phase.

In comparison to Päßler *et al.* [23], our method exhibits some key differences between our respective simulations and decision-making functions. Our method utilizes FMUs while Päßler *et al.* encode their simulation models entirely in Maude. Furthermore, they apply domain knowledge to assign quantitative *quality attribute* values with respect to objective fulfilment in their decision-making process, which subsequently allows them to select the maximum out of an available pool of scores and thus to choose the most appropriate model of premade decision-making rules. Since our approach is intended for more general applications, our own default decision-pruning function aims to be domain-independent, and, as a result, does not consider quantitatively scoring with respect to *OptimalCondition* restoration, although this does leave plenty of opportunity for user-provided logic with scoring to be used instead. Regardless, one of the most prominent differences between our works is in our solution's *rulelessness*. Whereas Päßler *et al.* rely on the aforementioned premade rule models, our approach ultimately enables DTs to make decisions without them.

In general, however, if a ruleless DT is to successfully decide on a combination of *Actions* to execute, it must still explore its tree of possibilities. Without streamlining, the DT is forced to consider all possible combinations of all *Actions*, both necessary and unnecessary, occurring within every possible simulation interval, all of which realistically equates to an expensively simulated trial-and-error approach. For this reason, to ensure the drawbacks of combinatorial explosions do not outweigh the benefits of flexibility, removing rules from DTs ultimately comes down to employing a reasonably effective decision-pruning process.

6 Future Work

Currently, our framework is purely reactive such that actions are triggered only by unmet *OptimalConditions*. It is straightforward to use the same simulation process for all possible *Actions* or a subset thereof, to preemptively avoid violating *OptimalConditions* in the *simulated state*, i.e., after the next simulation interval. As an example, a *Property* whose value is within its *OptimalCondition* requires no *Actions*. However, given an FMU showing that taking no *Actions* results in the *Property*'s value falling outside *OptimalConditions*, the same process would demonstrate that *Actions* should instead be taken proactively.

Our focus is on harnessing the power of simulation, and we plan to evaluate against real TTs in the future. We will thus also investigate the feasibility of implementing our planning solution on Talasila *et al.*'s [25] "DT-as-a-Service" platform that comprehensively addresses model management, including FMUs.

In addition to providing the instance model (which is easily automated for a given TT) with manual annotations for *OptimalConditions* and *PropertyChanges*, the quality of the simulations still hinges on the availability and fidelity of the respective FMUs. We plan to explore lightweight methods for implementing stakeholder preferences in the fitness function without extensive programming effort; e.g., SPARQL queries over simulated configurations could provide

the required flexibility in writing custom functions, though they might still be far from user-friendly.

During construction of the initial DT, instead of annotating *PropertyChanges* in the knowledge base, we could leverage FMUs to derive them. The FMUs must already contain information essential for describing *how Properties* are affected, so *PropertyChanges* might be automatically inferable from simple FMU test executions. This may also allow for more accurate *PropertyChanges*, not only describing *Effects*, but also *Property* value ranges they hold for, subsequently representing that, e.g., running an *Actuator* in different states causes different kinds of *Effects*. Additionally, historical data about specific *Action* combinations taken for specific TT conditions could comprise cases to power a framework akin to that of Michael *et al.* [20]. This could thus alleviate combinatorial blowups by matching on previously encountered TT conditions that already reference the optimal set of *Actions* to take.

Other potential future work includes exploring the feasibility of matching DTs with FMUs, based on ontological *Property* descriptions. The semantic data in the instance model, such as minimum/maximum values and SI units, could be used in conjunction with FMI model descriptions to facilitate FMU and/or parameter matching. With a given *matching threshold*, it may be possible to automatically offer ontologically compatible FMUs and alleviate stakeholder burden. Furthermore, with the introduction of solutions such as UniFMU [17], it may prove feasible to utilize model-driven code generation techniques to produce the required FMUs from additional information in the knowledge base. For example, identifying or deriving an FMU from the body of engineering knowledge for a room of a given size with components of given specifications sounds attainable. Moreover, we envision vendors potentially providing FMUs for their systems.

7 Conclusion

In this paper, we explore and demonstrate the feasibility of *ruleless* DTs. By utilizing widely-accepted technologies and standards (the SSN ontology, SPARQL queries, and FMUs), we present an implementation comprising ontologies, inference rules, simulations, and the MAPE-K architecture that enables the removal of explicit rule-based models from DT decision-making processes. We provide two instance models (available in the artifact) as examples and use one of them to demonstrate the different stages of the approach. The resulting combinatorial explosion of combinations needs heuristics that select promising simulations to remain manageable. Furthermore, we discuss the pros and cons of both the approach and the technologies used. Lastly, we discuss future work, among which are implementation improvements as well as explorations into the semantic matching of simulation models.

Acknowledgments. This work was supported by Bergens Rederiforening and the Norwegian Research Council project CroFlow: Enabling Highly Automated Cross-Organisational Workflow Planning (grant no. 326249).

References

1. Abrazeh, S., Mohseni, S.R., Zeitouni, M.J., Parvaresh, A., Fathollahi, A., Gheisarnejad, M., Khooban, M.H.: Virtual Hardware-in-the-Loop FMU Co-Simulation Based Digital Twins for Heating, Ventilation, and Air-Conditioning (HVAC) Systems. IEEE Trans. on Emerging Topics in Computational Intelligence (2023). https://doi.org/10.1109/TETCI.2022.3168507
2. Azeau, O.: Femyou (2020), https://github.com/Oaz/Femyou
3. Cimino, C., Leva, A., Ferretti, G.: Ensuring consistency in scalable-detail models for DT-based control. In: 17th IFAC Symp. on Information Control Problems in Manufacturing (INCOM 2021) (2021). https://doi.org/10.1016/j.ifacol.2021.08.158
4. Compagnucci, I., Snoeck, M., Asensio, E.S.: Supporting Digital Twins Systems Integrating the MERODE Approach. In: 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C) (2023). https://doi.org/10.1109/MODELS-C59198.2023.00079
5. Dalibor, M., Heithoff, M., Michael, J., Netz, L., Pfeiffer, J., Rumpe, B., Varga, S., Wortmann, A.: Generating customized low-code development platforms for digital twins. J. Comput. Lang. (2022). https://doi.org/10.1016/j.cola.2022.101117
6. European Telecommunications Standards Institute: SAREF: the Smart Applications REFerence ontology (2024), https://saref.etsi.org/core/v4.1.1/
7. Gabor, T., Belzner, L., Kiermeier, M., Beck, M.T., Neitz, A.: A simulation-based architecture for smart cyber-physical systems. In: 2016 IEEE Intl. Conf. on Autonomic Computing (ICAC). pp. 374–379 (2016). https://doi.org/10.1109/ICAC.2016.29
8. Guittoum, A., Aïssaoui, F., Bolle, S., Boyer, F., Palma, N.D.: Inferring Threatening IoT Dependencies using Semantic Digital Twins Toward Collaborative IoT Device Management. In: SAC '23: Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing (2023). https://doi.org/10.1145/3555776.3578573
9. Huang, X., Hu, S., Yang, H., Wang, X., Pei, Y., Shen, X.: Digital Twin-Based Network Management for Better QoE in Multicast Short Video Streaming. IEEE Trans. on Wireless Communications (2024). https://doi.org/10.1109/TWC.2024.3438159
10. Hugues, J., Hristosov, A., Hudak, J.J., Yankel, J.: TwinOps – DevOps meets Model-Based Engineering and Digital Twins for the engineering of CPS. In: MODELS '20: Proc. 23rd ACM/IEEE Intl. Conf. on Model Driven Engineering Languages and Systems: Companion Proceedings (2020). https://doi.org/10.1145/3417990.3421446
11. Jiang, Y., Yin, S., Dong, J., Kaynak, O.: A Review on Soft Sensors for Monitoring, Control, and Optimization of Industrial Processes. IEEE Sensors Journal (2021). https://doi.org/10.1109/JSEN.2020.3033153
12. Kamburjan, E., Klungre, V.N., Schlatte, R., Johnsen, E.B., Giese, M.: Programming and debugging with semantically lifted states. In: Verborgh, R., Hose, K., Paulheim, H., Champin, P., Maleshkova, M., Corcho, Ó., Ristoski, P., Alam, M. (eds.) The Semantic Web - 18th Intl. Conf., ESWC 2021. Lecture Notes in Computer Science, vol. 12731, pp. 126–142. Springer (2021). https://doi.org/10.1007/978-3-030-77385-4_8

13. Kirchhof, J.C., Malcher, L., Rumpe, B.: Understanding and Improving Model-Driven IoT Systems through Accompanying Digital Twins. In: Proc. 20th ACM SIGPLAN Intl. Conf. on Generative Programming: Concepts and Experiences (2021). https://doi.org/10.1145/3486609.3487210
14. Korth, B., Schwede, C., Zajac, M.: Simulation-ready digital twin for realtime management of logistics systems. In: 2018 IEEE International Conference on Big Data (Big Data). pp. 4194–4201. IEEE, Seattle, WA, USA (Dec 2018). https://doi.org/10.1109/BigData.2018.8622160
15. Kritzinger, W., Karner, M., Traar, G., Henjes, J., Sihn, W.: Digital Twin in manufacturing: A categorical literature review and classification. IFAC-PapersOnLine (2018). https://doi.org/10.1016/j.ifacol.2018.08.474
16. Larsen, P.G., Fitzgerald, J., Gomes, C.: Engineering Digital Twins for Cyber-Physical Systems, chap. 1. Springer Nature (2024). https://doi.org/10.1007/978-3-031-66719-0_1
17. Legaard, C.M., Tola, D., Schranz, T., Macedo, H., Larsen, P.G.: A Universal Mechanism for Implementing Functional Mock-up Units. In: Proc. 11th Intl. Conf. on Simulation and Modeling Methodologies, Technologies and Applications - SIMULTECH. vol. 1, pp. 121–129 (2021). https://doi.org/10.5220/0010577601210129
18. Lehner, D., Zhang, J., Pfeiffer, J., Sint, S., Splettstößer, A.K., Wimmer, M., Wortmann, A.: Model-driven engineering for digital twins: a systematic mapping study. Software and Systems Modeling (2025). https://doi.org/10.1007/s10270-025-01264-7
19. Lin, Y.J., Tu, Y.M.: A Scalable IoT-driven Smart Agriculture System: Ontology-based Inference and Automation for Hydroponic Farming. Sensors and Materials (2025). https://doi.org/10.18494/SAM5410
20. Michael, J., Schwammberger, M., Wortmann, A.: Explaining Cyberphysical System Behavior With Digital Twins. IEEE Software (2023). https://doi.org/10.1109/MS.2023.3319580
21. Musen, M.A., the Protégé Team: The Protégé Project: A Look Back and a Look Forward. AI Matters (2015). https://doi.org/10.1145/2757001.2757003
22. Ouedraogo, E.B., Hawbani, A., Wang, X., Liu, Z., Zhao, L., Al-qaness, M.A.A., Alsamhi, S.H.: Digital Twin Data Management: A Comprehensive Review. IEEE Trans. on Big Data (2025). https://doi.org/10.1109/TBDATA.2025.3533891
23. Päßler, J., Aguado, E., Silva, G.R., Tarifa, S.L.T., Corbato, C.H., Johnsen, E.B.: A Formal Model of Metacontrol in Maude. In: Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles (2022). https://doi.org/10.1007/978-3-031-19849-6_32
24. Sinreich, D.: An architectural blueprint for autonomic computing. Tech. rep., IBM (2006), https://web.archive.org/web/20230714193134/https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=0e99837d9b1e70bb35d516e32ecfc345cd30e795, (captured with the Wayback Machine)
25. Talasila, P., Gomes, C., Vosteen, L.B., Iven, H., Leucker, M., Gil, S., Mikkelsen, P.H., Kamburjan, E., Larsen, P.G.: Composable digital twins on Digital Twin as a Service platform. Simul. **101**(3), 287–311 (2025). https://doi.org/10.1177/00375497241298653
26. The Apache Software Foundation: Apache Jena (2025), https://jena.apache.org/
27. The Modelica Association: Functional Mock-up Interface (2025), https://fmi-standard.org/

28. The World Wide Web Consortium: SPARQL 1.1 Query Language (2013), https://www.w3.org/TR/sparql11-query/
29. The World Wide Web Consortium, Open Geospatial Consortium: Semantic Sensor Network Ontology (2017), https://www.w3.org/TR/vocab-ssn/
30. Vesse, R., Zettlemoyer, R.M., Ahmed, K., Moore, G., Pluskiewicz, T., Lang, S.: dotNetRDF (2025), https://dotnetrdf.org/
31. Yang, X., Ran, Y., Zhang, G., Wang, H., Mu, Z., Zhi, S.: A digital twin-driven hybrid approach for the prediction of performance degradation in transmission unit of CNC machine tool. Robotics and Computer-Integrated Manufacturing (2022). https://doi.org/10.1016/j.rcim.2021.102230