CEUR-WS.org/Vol-3291/paper4.pdf

A RESTful Interaction Model for Semantic Digital Twins

Daniel Schraudner[a], Andreas Harth[a,b]

[a] *Chair of Technical Information Systems, Friedrich-Alexander-University Erlangen-Nürnberg, Nuremberg, Germany*
[b] *Fraunhofer IIS, Fraunhofer Institute for Integrated Circuits IIS, Nuremberg, Germany*

Abstract

We provide a formal model based on extended state machines (EFSMs) with the addition of so-called admissibility functions to describe the state changes of an asset for interactions between HTTP agents and the asset by using the abstraction of properties, actions, and events. Furthermore, we describe the RESTful interface for assets that can be derived from EFSMs to offer assets' interactions in a clearly defined way and based on established standards. We use the continuous example of a robot arm to explain our results and provide the complete interaction model for the robot arm.

Keywords

Read-Write Linked Data, REST, Digital Twin, Web of Things, Extended Finite State Machines, Linked Data Platform,

1. Introduction

In the past semantic digital twins have been built by enriching machine data with semantics (using RDF uplifting) and making that data available [1]. For this purpose only an interface for reading the Linked Data is necessary. A real digital twin that is not just a digital shadow, however, must also offer the capability to interact with it by sending data to the digital twin that influences it in some way and thus also the asset that is modeled by the twin. An example of this we will also look at in greater detail, later on, would be a robot arm. We could use a semantic digital twin to, on the one hand, get semantic information about the robot arm, but on the other hand, also control the robot by sending RDF to the digital twin.

The question arises, what the RDF that must be sent to the digital twin in order to control it, should look like. For this purpose it is very helpful to have a clearly defined interface with clear semantics for the interactions between agents and digital twins, i.e. we need an interaction model that can be formally described. This will also easy automatic composition or orchestration of several semantic digital twins.

For the interface, it makes sense to build on technologies that support semantics and are already well established for companies. Thus we want to build the interface on existing Web technologies like REST, HTTP, and RDF. Utilizing these technologies makes a semantic digital

Third International Workshop On Semantic Digital Twins (SeDiT 2022), co-located with the 19th European Semantic Web Conference (ESWC 2022), Hersonissos, Greece - 29 May 2022
daniel.schraudner@fau.de (D. Schraudner); andreas.harth@fau.de (A. Harth)
0000-0002-2660-676X (D. Schraudner); 0000-0002-0702-510X (A. Harth)

© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

twin even more useful because it can be easily integrated into the existing IT infrastructure of companies, the number of semantic digital twins can be scaled up massively and data can be integrated among all those twins in a straightforward way.

In view of the above considerations, we will treat a digital twin as a read-write interface for assets, additional services (e.g. a predictive maintenance model) that are often seen as an integral part of a digital twin can be implemented in a decoupled way as microservices on top of that read-write interface of the digital twin.

In this paper, we provide a formal model based on extended finite state machines for the semantics of interactions between HTTP agents and digital twins. This model can be used to clearly define what effects a certain interaction with an asset has on its current state.

In the next section, we describe the related work. In Section 3 we briefly introduce the two concepts we are building our work upon, namely REST and extended finite state machines. In Section 4 we describe our formal interaction model and in Section 5 the related RESTful interface. In Section 6 we give a short discussion of the results and conclude our work in Section 7.

2. Related Work

The Web of Things (WoT) is an initiative to solve the increasing fragmentation of Internet of Things devices (i.e. the fact that one needs a new app for every device) by proposing a uniform application layer: the web. One central concept of the WoT are the so-called Thing Descriptions which are a semantic self-description of a device and its capabilities. Charpenay et al. [2] introduce the ontology for Thing Descriptions and they also specifically talk about interactions that things should expose. Even though they define interactions as web resources and state that there are three patterns for interaction (properties, actions, and events similar to our approach) they stay relatively vague about what exactly the difference between them is and do not give a clear definition. It is also not clear for the Web of Things which effects a certain interaction exactly has.

Ewert et al. [3] implemented asset administration shells which can be seen as a precursor for digital twins. The administration shells are based on so-called submodels that each describe one technical aspect of an asset. Submodels can specify properties, operations (similar to actions), and events for interaction with the asset. While Ewert et al. describe a clear interface for the interactions using MQTT, they also do not give semantics to the interactions, i.e. it is not clear what happens to the asset when a certain operation is executed or a property is changed.

Harth and Käfer [4] outline a formalism for Read-Write Linked Data based on state transition systems. They use a regular finite state machine formalization for their definition of a Linked System. Harth and Käfer do not differentiate between properties and actions as we do. Thus every HTTP request is an action and consequently leads to a transition in their model. In contrast for our model a PUT request just changes properties and does not lead to a change of the (symbolic) state. Hence our model can use much fewer states and thus prevent a possible state space explosion (to model a property of type `xsd:integer`, a finite state machine would already need infinitely many states, an extended finite state machine can cope with just one).

Gómez-Berbís and de Amescua-Seco [5] use Knowledge Graphs to give semantics to sensor

data of a digital twin, however, they only regard parameters (properties) to measure, analyze and monitor assets but they make no claims about how parameters to be changed in order to control the asset.

Boje et al. [6] propose the use of a semantic digital twin for the construction sector by using building information models. Although the authors acknowledge the need for a digital twin to be able to control the asset through its actuators their analysis of the research landscape does talk about sensing and monitoring, actuation, controlling, or interaction with the asset is not even a category for their comparison of different studies (the reason might be that this aspect of digital twins has not got much attention so far).

3. Preliminaries

3.1. Representational State Transfer (REST)

Representational State Transfer (REST) is an architecture style for distributed software systems introduced by Fielding [7]. REST comprises a set of constraints that should be fulfilled in order to create software systems that scale massively, like e.g. the Web. The constraints defined by Fielding are the following:

- Client-server principle
- Stateless communication
- Cachability
- Uniform interface (Unique resource identifiers, Resource manipulation through representations, Self-descriptive messages, Hypermedia as the engine of application state)
- Layered System
- Code-on-demand

For our work, the most important point in this enumeration is resource manipulation through representations. This means that in RESTful architectures, the state of resources cannot be changed by some explicit operation but only by submitting its new state to the resource. Suppose, e.g. we have a resource ”counter” that has the state ”3”. When we want to increase the counter by one, we cannot just call an operation ”increaseByOne” that does the increment but we explicitly have to tell the ”counter” that its new state should be ”4” (in case of HTTP often done using a PUT request).

REST constraints are quite general and could be applied to many technologies, most of the time, however, the REST principles are used for building scalable HTTP APIs. Because of the popularity of HTTP-based REST APIs on the Web, we also want our interaction model for semantic Digital twins to comply with the REST constraints. This makes it easy to interface with existing Web-based infrastructure as well as to scale up the number of semantic digital twins in use, as needed when using them in large-scale factories.

3.2. Extended Finite State Machines (EFSM)

Extended finite state machines (EFSMs) [8] are a generalization of the well-known finite state machine (FSM) model[1] that is being used in process of verification of soft- and hardware.

Informally, EFSMs extend FSMs by adding stateful variables to the machine, i.e. the current state of the machine is defined by an assignment of those variables together with the current *symbolic state* of the machine (which is comparable to the *state* in traditional FSMs). The current assignment of the variables is also used to determine which transitions from the current symbolic state to another one are allowed (*enabled*) and which are not.

The EFSM model assumes that variables are only changed when a transition happens and the way they are changed is defined in the update function for the transition. However, as we want our system to conform to the REST constraints, we need to take into account the fact, that the state of an asset can be manipulated directly and arbitrarily through the use of representations. This means that we cannot assume that the variable assignment will always stay the same while the EFSM is in the same symbolic state (as it could be changed using e.g. an HTTP PUT request). Nevertheless, we need to define which variable assignments are valid for a certain symbolic state[2] and thus slightly extend the EFSM model by an admissibility function.

We thus define an EFSM as a 10-tuple $E = (S, I, O, D, F, U, T, A)$, where

- S is the set of symbolic states,
- I is a set of input symbols,
- O is a set of output symbols,
- D is an n-dimensional space $D_1 \times \cdots \times D_n$ (i.e. the value space of all variables),
- F is a set of enabling functions f_i such that $f_i : D \to \{true, false\}$,
- U is a set of variable update transformations u_i such that $u_i : D \to D$,
- T is a transition relation such that $T : S \times F \times I \to S \times U \times O$ and
- A is the admissibility function such that $A : S \times D \to \{true, false\}$.

We use $(x_1, \dots, x_n) = x \in D$ for the current variable assignment and we write $(s, f, i) \to (s', u, o)$ to denote that if E is in the symbolic state s, it holds that $f(x) = true$ and input symbol i is received, then a transition to the new symbolic state s' happens where the variables assignment is updated $x \leftarrow u(x)$ and the output symbol o is sent.

For the admissibility and enabling functions we use abbreviations such as $x_1 < 10 \land x_2 = true$ to denote the function

$$f(x_1, x_2) = \begin{cases} true & \text{if } x_1 < 10 \land x_2 = true \\ false & \text{else.} \end{cases} \tag{1}$$

as well as just $true$ and $false$ for the respective constant functions.

An EFSM can be represented using a graph where the nodes represent the symbolic states and the edges represent the transitions between them. We annotate the admissibility function

[1] More precisely of finite state transducers

[2] We need this such that nobody can send a representation with a variable assignment that is (physically) not possible or nonsensical for the asset.

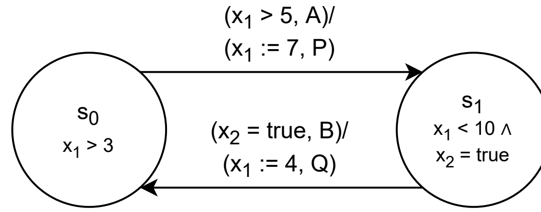

Figure 1: An example of an EFSM. Note that the formulas in the nodes do not represent a variable assignment but the admissibility functions of the symbolic state in the abbreviated notation.

for each state at the node and the enabling function and input symbol as well as the update function and output symbol in the form $(f, i)/(u, o)$ at each transition.

Figure 1 shows an example of an EFSM with the symbolic states $S = \{s_0, s_1\}$, the input symbols $I = \{A, B\}$, the output symbols $O = \{P, Q\}$ the variable value space $D = \mathbb{N} \times \{true, false\}$, two transitions $(s_0, x_1 > 5, A) \rightarrow (s_1, x_1 := 7, P)$ and $(s_1, x_2 = true, B) \rightarrow (s_0, x_1 := 4, Q)$, and the admissibility function $A(s_0) = x_1 > 3; A(s_1) = x_1 < 10 \wedge x_2 = true$.

4. Interaction Model

We want to use a robot arm as a running example for the rest of the paper. We assume the robot arm can have three different positions, it can sense whether an item is currently in the clamp and it can open and close its clamp. A few of the different possible states of the robot arm can be seen in Figure 2. Note that the two conveyor belts are other assets that can interact with the robot arm physically (by moving items) but cannot be controlled directly using the digital twin.

During the rest of this section, we will provide an interaction model for this robot arm by constructing a suitable EFSM. For this purpose will use the established abstractions of properties, actions, and events and explain how these can be represented using our formal model.

We use EFSMs with admissibility functions – in contrast to normal FSMs – for our interaction model to be able to describe the behavior of an asset in a very concise way with as few states as possible and thus prevent a state space explosion which would most of the time lead to having infinitely many states.

4.1. Properties

In our model properties are represented using the variables of an EFSM where every variable models a distinct property. An EFSM with the variable value space $D = \{1, 2, 3\} \times \{true, false\} \times \{true, false\}$ thus could be used to describe the properties of the robot arm. We will use the following variable names: $(pos, item, closed) \in D$ in the following.

As all variables of an EFSM always have a clearly defined value assignment, properties can always be read (by using HTTP GET requests) and return a value.

Writing properties on the other side is subject to some restrictions, as the admissibility function for the current symbolic state needs to allow the new variable assignment that is to be written. This way properties can be made read-only (and thus not be changed from the outside directly) by fixing the value through the admissibility function. If e.g. our EFSM currently is

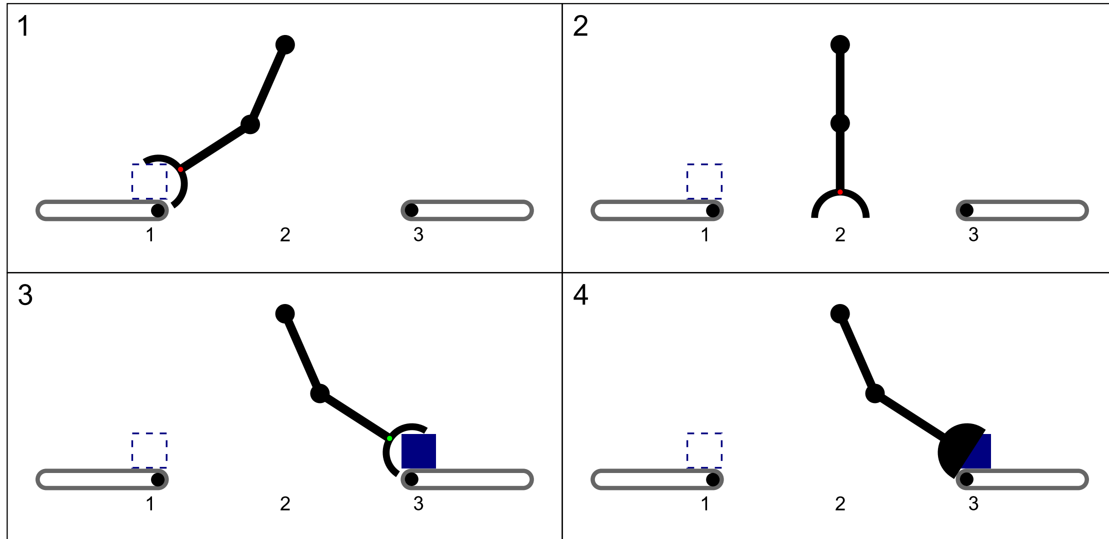

Figure 2: A robot arms in different states: 1. The arm is at position 1, opened and senses no item - 2. The arm is at position 2, opened and senses no item - 3. The arm is at position 3, opened and senses an item - 4. The arm is at position 3, closed and senses an item

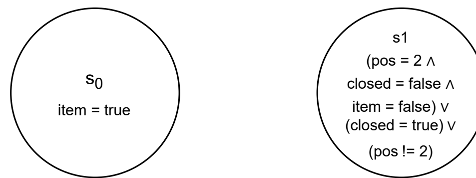

Figure 3: EFSM for the robot arm with only two symbolic states, the first having a variable that models a read-only property, the second modeling an interdependency between two properties.

in the symbolic state s_0 and $A(s_0) = (item = true)$, then *item* can not be changed to any other value than *true*, i.e. it is read-only (in the current symbolic state). This would reflect the fact that in the physical world there currently is an item under the sensor, and sensors are read-only devices (that cannot make an item disappear). The admissibility function can thus be seen as an invariant over the variables for the current symbolic state.

Also complex interdependencies between different variables can be modeled, e.g. by assuming $A(s_1) = (pos = 2 \wedge closed = false \wedge item = false) \vee (closed = true) \vee (pos! = 2)$. Here the possible value for *item* depends on the current value of *pos* and *closed* which means that when *pos* or *closed* is changed, *item* also might need to be changed[3]. We need a property interdependency in this case because a state of the robot arm where the clamp is open in position 2 and an item is sensed at the same time is physically impossible (the item would fall down) and thus must be prohibited. The two states we gave as an example can be seen in Figure 3.

[3]Note that we do not need transactions or anything similar – the REST principle of resource manipulation through representations makes this possible as we always set the new value for all of our properties.

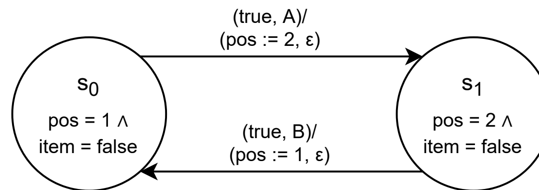

Figure 4: EFSM for the robot arm with only two symbolic states representing different positions and a transition between them.

An important characteristic of properties is that a change of the same must always happen instantaneously. If there is some process in the physical world behind the change of a property, then such a change is often not possible instantaneously. For e.g. the position of a robot arm can not change instantaneously because the arm has to move to the given position first which takes some time; the position of a robot, therefore, is not suitable to be modeled as a property and actions should be used for this purpose instead.

4.2. Actions

Because there are interactions with an asset (or more specifically with its digital twin) that cannot be modeled using simple properties – namely those where changes to the state do not happen instantaneously but only over a longer period of time, we need to introduce a possibility to execute those interactions. We will call this type of interaction a *action*.

In the case of the robot arm, such an action would be the movement of the arm between different positions. Although there is a property that signals the current position of the robot arm, we cannot use it for setting a new position as the arm would take some time (maybe several seconds) to move there[4]. We model an action in an EFSM with one or multiple transitions. When an action can be modeled by a single transition the action can be identified with an input symbol of the EFSM.

Let us look at the EFSM that can be seen in Figure 4: We have two symbolic states, s_0 that fixes the position to "1" and s_1 that fixes the position to "2" (while making *item* read-only and making no restrictions about the clamp, both). As already mentioned it would not be appropriate to change the position directly by manipulating the property, but we need a transition for this.

The transition going from s_0 to $s_!$ that can be seen in the figure has the enabling function *true* (i.e. it is always enabled), the input symbol A, the update function $pos := 2$ which sets the variable *pos* to "2" (which coincides with the allowed value in symbolic state s_1) and the empty output symbol ϵ (i.e. it has no output). We can now identify the action "moveFromPos1ToPos2" with the input symbol A, i.e. when an agent wants to execute this action, the input symbol A is sent to the EFSM and the corresponding transition is eventually executed (updating the variables among other things). It is important to note that we make no assumption about when the transition is executed after the input symbol is sent because we do not know how long

[4]Note that we assume a robot arm that moves slowly (moving cannot be modeled using a property) but can close and open its clamp very fast (we assume it happens instantaneously and use a property for it). In practice, it always will depend on the scenario whether "several seconds" can be treated as instantaneous change or change over a longer period of time.

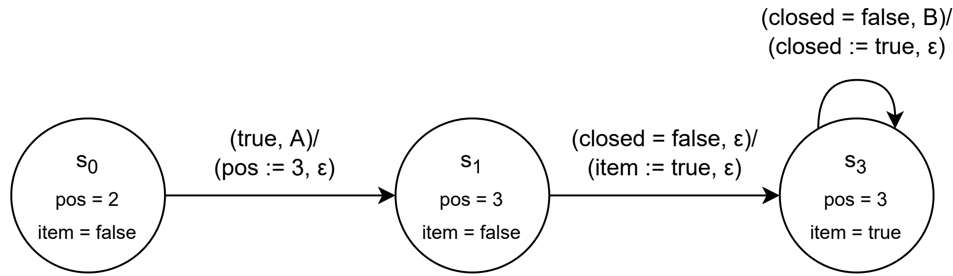

Figure 5: EFSM for the robot arm with only three symbolic states containing an epsilon transition. Note that state s_3 has a loop transition here that just changes the value of a variable but not the symbolic state. We could also introduce another symbolic state as a target for the transition, however, this is not necessary as both states would have the same admissibility function as well as the same ingoing and outgoing transitions.

the robot arm will take to move, however, we know the transition will eventually be executed before processing the next input symbol.

The transition from s_1 to s_0 on the other hand takes the input symbol B and sets the position back to "1", so the action behind B could be seen as the inverse action to the action behind A. However, we can also imagine that actions can be parametrized. Then we could have just one action "moveArm" and depending on the parameter (e.g. a boolean flag for the direction) it is chosen which input symbol is submitted to the EFSM (i.e. there needs to be a mapping from the parameter value space to the set of input symbols).

An action can not always be mapped to just one single transition and consequently not to one single input symbol. This is always the case when there is some interaction with processes in the physical environment. For our robot arm, we could want to make an action "pickUp" available that picks up a new item when there currently is no item in the clamp. However, the robot arm cannot just spawn items but it needs to go to position "1", open its clamp and wait for the conveyor belt to bring an item (we assume that this will always happen if we wait long enough).

To solve this issue we need to introduce the possibility to have transitions that do not need an input symbol to be executed (we then write ϵ instead of an input symbol, similar to what we have seen for output symbols and call that transition epsilon or empty transition) but can just happen spontaneously as long as their enabling function is fulfilled. This type of transition can be used to model processes that happen in the environment and which the asset cannot influence directly.

In Figure 5 we can see an EFSM with three transitions: the first one with input symbol A moving the arm to position "3", the second one being an epsilon transition modeling the fact that an item has arrived at the item sensor and the third one having input symbol B closing the clamp. We can now model the action "pickUp" as the sequence of input symbols $[A, B]$. When an agent wants to execute this action, the symbols A and B are submitted to the input queue of the EFSM. First, the A transition is executed and the robot arm moves to position "3". Then it waits for the epsilon transition to happen (for the item to appear) and afterward the B transition is executed and the clamp is closed.

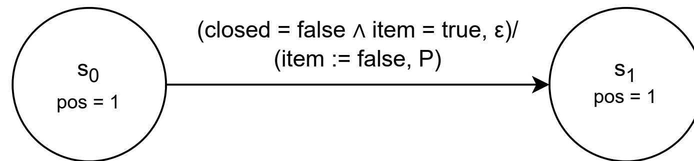

Figure 6: EFSM for the robot arm with only three symbolic states containing an epsilon transition that has an output.

4.3. Events

For the interaction with an asset not only the current state could be of interest but also what has happened in the past. For e.g. one could want to do something (increasing a counter, etc.) every time the robot arm delivers an item successfully to the conveyor belt at position "1". Every type of event can be identified with an output symbol in the EFSM. Thus every time a transition is executed, an event (of the event type belonging to the output symbol) is emitted (ϵ means there is no event to emit).

In Figure 6 we can see an EFSM that represents exactly this case. We have two symbolic states that fix the position to "1" and an epsilon transition between them that is active when there is an item in the clamp and it is open. As soon as the transition is executed it outputs the symbol P. This occurrence of an event can be made available (together with a timestamp) to agents over the HTTP interface.

5. Interface

In the previous section, we explained how properties, actions, and events of an asset can be modeled using an EFSM, i.e. what effects interaction can have, but we only touched on the topic of how the interactions between HTTP agents and the asset themselves work. We now want to describe those interactions more extensively in the following.

The HTTP interface that can be derived from the EFSM is based on the Linked Data Platform (LDP) [9] standard.

5.1. Properties

As explained before the current values of the properties can be read all the time. This can be done using an HTTP GET request to an endpoint representing the robot arm (i.e. its digital twin). An example of an RDF graph the endpoint could return can be seen in Figure 7 where we can see the current value of all the properties. The RDF predicates `:pos`, `:item`, and `:closed` map directly to the variables of the EFSM with the respective name and are used to display the property values as RDF.

If an agent now wants to change a property it must send an HTTP PUT request against the endpoint with the RDF representing the new desired state, e.g. could the "closed" property be changed to "false". The request can either be accepted by the digital twin, then it returns a 204 status code and changes the property, or it can be denied. The denial can have multiple reasons (e.g. could the server be overloaded and return a 503 status code) but the important reason for

```
@prefix : <http://example.org/robotArm#> .

</>  a        :RobotArm ;
    :pos      1 ;
    :item     false ;
    :closed   true ;
    :tasks    </tasks/> ;
    :events   </events/> .
```

Figure 7: Example RDF to describe the robot arm.

us is when the new property (and thus EFSM variable) assignment is not allowed for the current symbolic state by the admissibility function. In that case, the digital twin would return a ”409 Conflict” status code.

5.2. Actions

Agents could submit actions to the digital twin by sending a POST request with the needed parameter to an endpoint for the action (e.g. ”/pickUp”). However, this would not be in accordance with the REST principles where we always want to manipulate the state of a resource by sending a representation[5]. To comply with the REST constraints we create an explicit resource every time an agent wants to execute an action and call that resource a *task*.

A task specifies which actions should be executed with which parameters (i.e. it can be directly mapped to a sequence of input symbols for the EFSM) and tasks can also specify an order among them (when there are multiple tasks submitted by agents at the same time or shortly after each other, the asset needs to decide which one should be executed next, e.g. using a priority for the tasks).

In Figure 8 we can see the RDF that we get when referencing the task endpoint (which is an LDP container) of the robot arm and all tasks that are listed under this endpoint. We see that the robot arm currently has two tasks, the first being the task ”pickUp”, the second being a move task with a parameter[6]. ”PickUpTask”, as well as ”MoveTask”, can be directly mapped to a sequence of input symbols for the EFSM (possibly of size one). If an agent wants to create a new task, it can send a POST request against the task container endpoint according to the LDP standard.

5.3. Events

Events that are created by assets are read-only for agents, thus we can just deference the event endpoint (again an LDP container) of the robot arm and get a list of events. How these events could look like can be seen in Figure 9. As we have a RESTful architecture events are not

[5] A POST request to an action-specific endpoint has more similarities with a Remote Procedure Call – an architecture for distributed applications opposing REST.

[6] As already mentioned we can treat tasks with parameters as equivalent to multiple tasks without parameters, i.e. instead of `:MoveTask` with a `:toPos` parameter we could have `:MoveToPos1Task`, `:MoveToPos2Task`, etc.

```
@prefix : <http://example.org/robotArm#> .
@prefix ldp: <http://www.w3.org/ns/ldp#> .

</tasks/> a             ldp:BasicContainer ;
          ldp:contains </tasks/1>,
                       </tasks/2> .
```

```
</tasks/1> a          :PickUpTask ;
           :priority 3 .

</tasks/2> a          :MoveTask ;
           :toPos     1
           :priority 2 .
```

Figure 8: Example RDF to describe tasks of the robot arm.

```
@prefix : <http://example.org/robotArm#> .
@prefix ldp: <http://www.w3.org/ns/ldp#> .

</events/> a             ldp:BasicContainer ;
           ldp:contains </event/1>,
                        </event/2> .
```

```
</events/1> a     :DeliveredItemEvent ;
            :time 43971 .

</events/2> a     :DeliveredItemEvent ;
            :time 43987 .
```

Figure 9: Example RDF to describe events of the robot arm.

proactively pushed to agents but instead have to be retrieved regularly by polling them[7]. By looking at the type of the event, an agent can determine the output symbol which maps to the type and thus which transition has been executed in the past.

6. Discussion

In the previous two sections, we described in accordance with [2][3] three basic types of interactions with assets that can be provided by semantic digital twins: properties, actions, and events. We showed that properties can be modeled using the variables of an EFSM and that the manipulation of those properties can be restricted using the admissibility function.

We defined a soft criterion (instantaneous change) for deciding whether a given capability of an asset should be modeled as a property or as a process defined by an action. We also gave a clear semantic to those actions by mapping them to a sequence of input symbols that can execute transitions – if necessary thereby also interacting with processes from the physical environment.

Events have been defined as the output of a transition, i.e. the result of a state change that can be recorded and made accessible in a RESTful way.

To verify that all of our requirements for the robot arm (moving is not instantaneous, items do appear only at the conveyor belt at position ”3”, etc.) can be expressed using our EFSM model, we give the complete model of the robot arm in Figure 10. We can see that there are different symbolic states for every position, such that the position can only be changed using an action. We also have the two epsilon transitions when an item appears at the conveyor belt or is delivered to the other conveyor belt; this is the only way the ”item” property can be changed.

We have four distinct input symbols A, B, C, and D each on mapping directly to a move action (e.g. ”moveToPos1”, etc.). There exists one event, namely the already mentioned ”delivery”

[7] If a use case requires it, pushing events (e.g. using WebSockets) would however not be a problem from our conceptual point of view.

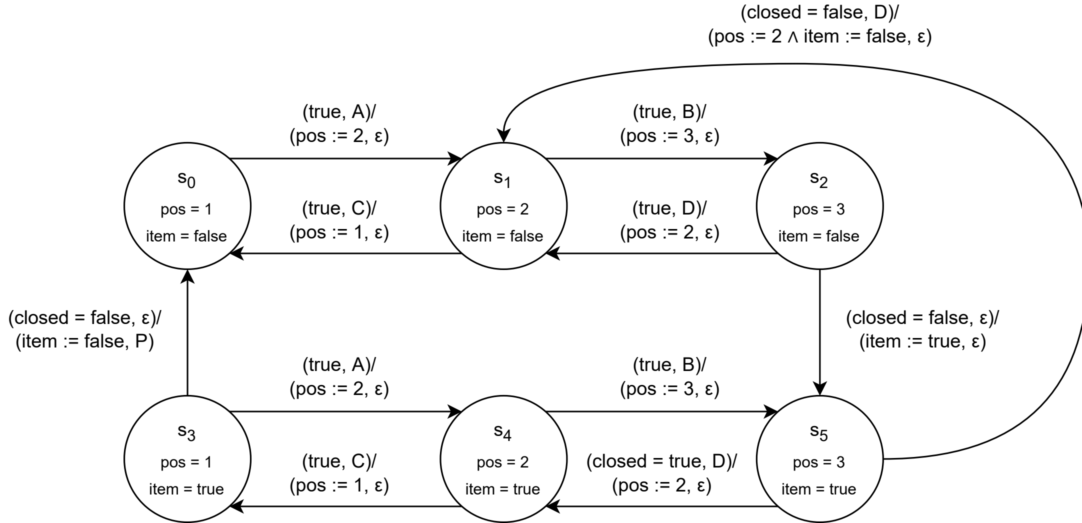

Figure 10: Complete EFSM for the robot arm.

represented by the output symbol P. Also, note the two transitions that go out of s_5 for input symbol D: They have two mutually exclusive activation functions as the item is dropped when the clamp is not closed while moving to position ”2” which leads to symbolic state s_1 and to symbolic state s_4 otherwise when the clamp is closed.

7. Conclusion & Future Work

We provided a formal model in the form of EFSMs extended by admissibility functions for the interaction of HTTP agents with the digital twins of arbitrary assets and described a RESTful interface that can be directly derived from the EFSM. This formal model can on the one hand be used to understand the interactions between different agents and assets better, on the other hand, it could be communicated to the agent to help it to decide how to interact with the asset. If an agent knows e.g. that it wants to transport an item from one conveyor belt to the other, it could use the EFSM to deduce which properties to change, which tasks to submit, and for which events to wait in which order.

We think that our formal model is general enough, to also have applications outside the field of digital twins, e.g. for describing the semantics of properties, actions, and events for the Web of Things or interactions of general RESTful interfaces. A formal interaction model could be used in these cases for model checking certain properties of an asset’s behavior, e.g. whether there exists a sequence of HTTP requests that leads the asset to a deadlocked state and thus makes it unusable.

In the future, we also plan to investigate the connection between different EFSM that can be used to model the same asset. A still open question is if there is a way to transform different EFSMs modeling the same asset into each other and if we can define some canonical form.

Acknowledgments

This work was funded by the German Federal Ministry of Education and Research through the MOSAIK project (grant no. 01IS18070A).

References

[1] S. R. Bader, M. Maleshkova, The semantic asset administration shell, in: M. Acosta, P. Cudré-Mauroux, M. Maleshkova, T. Pellegrini, H. Sack, Y. Sure-Vetter (Eds.), Semantic Systems. The Power of AI and Knowledge Graphs, Springer International Publishing, Cham, 2019, pp. 159–174.

[2] V. Charpenay, S. Käbisch, H. Kosch, Introducing thing descriptions and interactions: An ontology for the web of things., in: SR+ SWIT@ ISWC, 2016, pp. 55–66.

[3] D. Ewert, T. Jung, T. Tasci, T. Stiedl, Assets2036–lightweight implementation of the asset administration shell concept for practical use and easy adaptation, in: Advances in Automotive Production Technology–Theory and Application, Springer, 2021, pp. 153–161.

[4] A. Harth, T. Käfer, Towards specification and execution of linked systems, in: Proceedings of the 28th GI-Workshop Grundlagen von Datenbanken (GvD), 2016, pp. 62–67.

[5] J. M. Gómez-Berbís, A. de Amescua-Seco, Sedit: Semantic digital twin based on industrial iot data management and knowledge graphs, in: R. Valencia-García, G. Alcaraz-Mármol, J. Del Cioppo-Morstadt, N. Vera-Lucio, M. Bucaram-Leverone (Eds.), Technologies and Innovation, Springer International Publishing, Cham, 2019, pp. 178–188.

[6] C. Boje, A. Guerriero, S. Kubicki, Y. Rezgui, Towards a semantic construction digital twin: Directions for future research, Automation in Construction 114 (2020) 103179. URL: https://www.sciencedirect.com/science/article/pii/S0926580519314785. doi:https://doi.org/10.1016/j.autcon.2020.103179.

[7] R. T. Fielding, Architectural styles and the design of network-based software architectures, University of California, Irvine, 2000.

[8] K.-T. Cheng, A. Krishnakumar, Automatic functional test generation using the extended finite state machine model, in: 30th ACM/IEEE Design Automation Conference, 1993, pp. 86–91. doi:10.1109/DAC.1993.203924.

[9] S. Speicher, J. Arwe, A. Malhotra, Linked data platform 1.0, W3C Recommendation, February 26 (2015).