Stateful-WoT: Capturing the Behavior of Highly Dynamic Cyber-Physical Systems

Fady Salama[1] Ege Korkan[2], Sebastian Käbisch[2], Sebastian Steinhorst[1]
[1] *Technical University of Munich*, Germany, Email: {fady.salama, sebastian.steinhorst}@tum.de
[2] *Siemens AG*, Germany, Email: {ege.korkan, sebastian.kaebisch}@siemens.com

***Abstract*—By introducing the Web of Things (WoT) standard, W3C aims to provide a unified interface description format to counter the high fragmentation of the Internet of Things (IoT) landscape. This description format, called the Thing Description (TD), is a static, JSON-Linked Data (JSON-LD) document that is both human and machine-readable. It lists all possible interactions of a Thing and any additional metadata needed to perform these interactions, abstracting away from the internal behavior of Things. As such, the static TD lacks a formal way of describing highly dynamic, physical Things and how the availability of specific interactions may depend on the current physical state of the Thing. In this work, we introduce Stateful-WoT, a method and an open-source implementation that facilitates the modeling of Things and their behavior using State Chart XML (SCXML) state machines and Modelica models. The resulting hybrid state chart is serializable and exchangeable, enabling the exchange of the state chart alongside the TD. Our proposed extension to SCXML allows us to fully model a Thing and its interface, facilitating the automatic generation of implementation code and reactive Digital Twins (DTs). We showcase the benefits and viability of our approach and implementation by modeling and generating the DT of two highly dynamic Things with high accuracy. Our proposal makes developing WoT applications more accessible, faster, and much more reliable for complex industrial scenarios.**

***Keywords*—Internet of Things, Web of Things, State Machines, Digital Twins, Simulation**

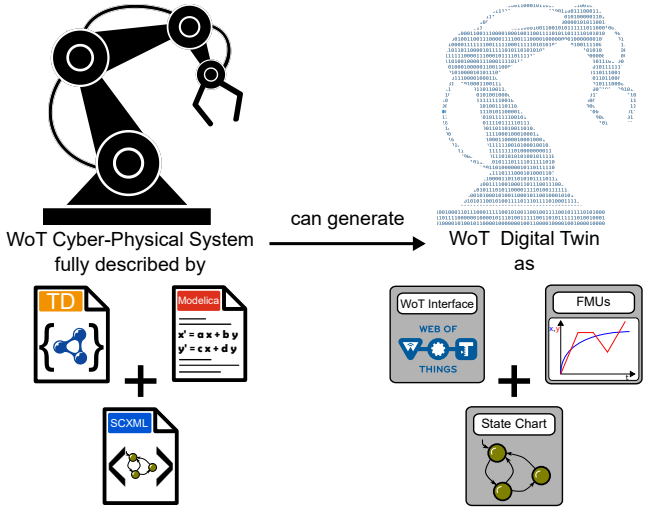

Fig. 1: In this paper, we propose a new building block for the Web of Things (WoT) for describing the behavior of Cyber-Phyiscal Systems (CPSs) using State Chart XML (SCXML) and Modelica models. We show that using our proposed method, it is possible to provide a serializable and exchangeable behavior description format alongside the WoT Thing Description (TD) that can be used to automatically generate a Digital Twin (DT) using state charts and Functional Mock-up Units (FMUs) that can be used for simulations.

I. Introduction

Interconnected Cyber-Phyiscal Systems (CPSs) are the core of Industry 4.0. Such systems can utilize their network for Machine-To-Machine Communication to achieve unprecedented automation and intelligence and, in doing so, maximize the efficiency of manufacturing and production. To achieve such a high interconnection of devices and systems, Industry 4.0 must employ Internet of Things (IoT) technologies that facilitate this. As such, many companies and vendors provide their own IoT solutions and architectures that aim to streamline the process of integrating CPSs and connecting them. However, this resulted in the high fragmentation of the IoT landscape, limiting the interoperability of different IoT solutions and defeating the goal of integrating IoT in the industry.

To achieve this, the W3C proposed the Web of Things (WoT), a set of building blocks built around existing web standards and technologies that aims to unify the heterogeneous IoT landscape. One of these building blocks is a metadata description format called the Thing Description (TD), a JSON-Linked Data (JSON-LD) document that is both human-and machine-readable. The TD describes the web interface provided by the described device or CPS, called a Thing in the context of this paper, including the protocol needed for the communication and payload structures.

Problem Statement*:* Using existing web technologies, the Web of Things (WoT) provides its building blocks using tried-and-true Web formulas, key enablers of the current Web ecosystem, including servers, browsers, search engines, and more. However, industrial Cyber-Phyiscal Systems (CPSs) provide challenges these technologies are ill-suited to handle.

In its current form, the Thing Description (TD) statically lists all possible interactions with a Thing. On the other hand, CPSs may include highly dynamic actors, such as conveyor belts and robot arms, that expose a dynamically updated interface based on their current state. For example, industrial motor drivers usually require a sequence of operations to start running, and each of these operations is tied to a certain motor drive state. Another example is a robot arm that has reached its workspace limit and, therefore, does not allow turning in a certain direction based on its current state. Describing these complex Application Programming Interfaces (APIs) using a static TD is currently impossible, as it requires an additional

```
1  {"@type": "Thing",
2  "title": "Smart Microwave",
3  "properties": {
4    "microwaveState": {
5      "title": "Microwave State",
6      "readonly": true,
7      "forms": [{
8        "href": "http://example.co/state",
9        "contentType": "application/json",
10       "op": ["readproperty","writeproperty"]
11     }]
12   }
13 },
14 "actions": {"turnOn": {"title": "Turn On"},
15             "turnOff": {"title": "Turn off"}},
16 "events": { "cookingDone": {}}}
```

Listing 1: This is an example Thing Description (TD) for a smart microwave that can be turned on and off and that sends a notification to listeners once the cooking is done. Each TD categorizes the interactions that the underlying Thing is exposing to one of three type: `"properties"` (Line 3), `"actions"` (Line 15) and `"events"` (Line 16). `"microwaveState"` (Lines 4-13) shows how a full interaction affordance could look like. The rest of the TD was omitted for brevity.

description of the underlying behavior of a device, which the TD abstracts from by design.

Contributions*:* In this paper, we propose **Stateful-WoT**, an approach to enhance the WoT by adding an additional building block for describing the behavior of Things using the open standards W3C State Chart XML (SCXML) and Modelica, facilitating the modeling of the Things, automatic generation of implementation code and the associated, extended TD and the exchange of behavioral models. In particular, we perform the following contributions:

- We introduce a formalized mapping between the semantics of the WoT model and state charts in Section III-A.
- We propose an extension to both the SCXML and TD that captures the proposed mapping in Section III-B.
- We introduce an algorithm that takes an extended SCXML as an input and automatically generates the implementation code of the modeled Thing and its extended TD in Section III-B.
- We showcase the open-source implementation of our proposed approaches in Section IV.
- We evaluate our approach and implementation based on different use cases in Section V.

The rest of the paper is structured as follows: We provide a short overview of the WoT, SCXML, Modelica, and Functional Mock-up Interface (FMI) standards as needed to understand the rest of this paper in Section II-A, Section II-B, and Section II-C, respectively. We discuss related work in Section VI and Section VII concludes.

II. Background

A. The Web of Things (WoT) and the Thing Description (TD)

The WoT [1] is an amalgamation of multiple web standards aimed at ensuring the interoperability of Internet of Things (IoT) technologies while keeping new prescriptions to a minimum. It introduces a set of building blocks that, together, form an ecosystem for interacting with any device or service, called Thing in the context of this paper, in an easy and streamlined manner. The core building block of the WoT is the TD [2], a description format written as a JSON-Linked Data (JSON-LD) document and is, thus, highly human- and machine-readable. The TD describes API provided by a Thing, regardless of the underlying protocol, and any metadata required for interacting with a Thing. It introduces three main types of interaction affordances that a Thing can provide:

1) **Property Affordances** provide operations for interacting with internal states of a Thing, such as reading, writing, or observing them.
2) **Action Affordances** provide operations for handling longer processes that a Thing can perform, such as invoking or canceling these actions.
3) **Event Affordances** provide operations for subscribing or unsubscribing to event streams and notifications.

Each affordance may include a JSON Schema describing the expected input and/or output payload of the affordance. The mapping between affordance operations and the underlying communication protocol is done using a Protocol Binding [3]. In WoT, Things that expose a TD are called Producers, while those that consume a TD are called Consumers. Furthermore, the WoT proposes an optional Scripting API for developing the application logic of Things in its ecosystem. Using the Scripting API simplifies the process of developing WoT applications. An excerpt of a TD for a smart microwave can be viewed in Listing 1.

B. State Chart XML (SCXML)

SCXML [4] is a standard introduced by W3C for describing Harel State Charts [5] in a serializable and machine-readable format with the original goal of providing a general-purpose execution environment for CCXML and VoiceXML. However, SCXML is not restricted to this use case and has been used for other scenarios, such as model-driven device coding or UI control.

SCXML allows the modeling of all state chart formalisms. In addition to normal states defined in a finite state machine, state charts introduce orthogonality and depth by introducing parallel states for modeling concurrency and compound states, which contain other child states, respectively. Modern state charts also introduce the notion of the extended state for modeling state variables that are infeasible or impossible to model using a finite amount of states. The state machine can perform actions on the entry or exit of any state or as a result of a transition. Transitions may occur when an event is triggered, a condition on the finite states or extended states or the triggering event is met, or always. Events may include additional data, such as data payloads. SCXML allows the state machine to raise or send messages to itself or external services. SCXML can also model state machines that invoke services, such as external web services or other state machines. The syntax of the executable code of the state machine and data expressions are based on the user-defined data model such as `"javascript"` or `"xpath"`.

```
1  <?xml version="1.0"?>
2  <scxml xmlns="http://www.w3.org/2005/07/scxml"
3  version="1.0" datamodel="ecmascript" initial="off">
4    <datamodel>
5      <data id="door_closed" expr="true"/>
6    </datamodel>
7
8    <state id="off">
9      <transition event="turn.on" target="on"/>
10   </state>
11
12   <state id="on" initial="idle">
13     <state id="idle">
14       <transition event="door.close"
         ↪  target="cooking">
15         <assign location="door_closed" expr="true"/>
16       </transition>
17     </state>
18
19     <state id="cooking">
20       <transition event="door.open" target="idle">
21         <assign location="door_closed"
           ↪  expr="false"/>
22       </transition>
23     </state>
24   </state>
25 </scxml>
```

Listing 2: This is an excerpt of an State Chart XML (SCXML) file that models a microwave. It can be turned on (Lines 9&12), and once it reaches the `"on"`, it can be either in `"idle"` or `"cooking"` sub-states (Lines 13-22). An important feature of SCXML is the capability to model an Extended State (Lines 4-6) containing variables that are not necessarily discrete and countable in nature.

C. Modelica and Functional Mock-up Interface (FMI)

Modelica [6] is an effort to provide an open, mathematical modeling language for CPSs across multiple engineering domains and is currently maintained by the Modelica Association. Using Modelica, it is possible to describe systems of differential or discrete time equations that can be parsed and solved by dedicated software. The Modelica language is also object-oriented by design to facilitate easy development and code reuse. A Modelica model comprises the interface declaration and the equations declaration. The interface declaration describes all external and internal simulation variables, their types, and their direction in the case of external variables as either **`input`**, **`output`** or **`parameter`**. The equations declaration describes the model as a system of differential, algebraic, and discrete equations, which a Modelica tool can parse and usually convert to a set of ordinary differential equations and solve using appropriate solvers.

The Modelica Association has also introduced another standard called the Functional Mock-up Interface (FMI) [7] that enables the exchange of simulation models across the simulation environment by combining a description format and C-based binaries required to run the simulation. The description format in XML can be considered another representation of the equations system that a Modelica model can define. Based on the FMI version, the description format may include links to the executable files that can run the simulation. Two types of FMIs are Model Exchage (ME) and Co-Simulation (CS). While an extensive comparison between both types is out of scope, the main difference between an ME FMI and a CS FMI is that CS FMIs can be considered as a black box component that includes its own solver as well, while ME FMIs rely on the simulation environment providing its own solver for the simulation system. The executable code of am FMI is called Functional Mock-up Unit (FMU). Our paper here utilizes FMI 2.0 standard.

III. Stateful-WoT

In this section, we introduce our approach **Stateful-WoT** by first introducing the formalism for describing WoT producers as event-driven state charts and then explaining the extensions needed in SCXML and TD to accommodate the described formalism.

A. Formalism

Looking at possibilities to model the application WoT producers, an option would be an event-driven finite state machine, defined as a tuple $\langle S, s_0, E, E_s, O, en, ex, T\rangle$, similar to definitions in [8] where

- S is the finite set of States,
- $s_0 \in S$ is the initial state,
- E is the set of all possible external events triggering a transition in the state machine,
- E_s is the set of all events raised by the state machine,
- O is the set of all possible outputs,
- $en : S \to (E_s \cup O)^*$ are entry actions of a state,
- $ex : S \to (E_s \cup O)^*$ are exit actions of a state,
- $T \in S \times (E \cup E_s) \times G \times (O \cup E_s)^* \times S$ is the set of all transitions as a tuple $\langle s, e, g, o, t_s\rangle$, where $s \in S$ denotes the source state, $e \in (E \cup E_s)$ denotes the guard event, $g \in G$ denotes a logical guard over S, and $t_s \in S$ denotes the target state. $*$ denotes the Kleene Star operator.

Each request can be modeled as an event trigger, which would cause a transition in the application state if the state machine is in an appropriate state. But, as discussed in Section II-B, modeling complex systems using finite state machines can become quickly infeasible or outright impossible because of the limited memory that a state machine has, i.e., the finite amount of states. As such, finite state machines in their classical definition are not Turing Complete and thus cannot model any arbitrary application logic for a WoT producer.

By introducing the notion of the extended state S_{ext}, the finite state machine gains a powerful tool for emulating an unbounded memory, making it equivalent to a complete Turing machine. Formally, the tuple introduced above is extended to $\langle S, S_{ext}, s_0, s_{ext,0} E, E_s, O, en, ex, T\rangle$, where:

- S_{ext} is the set of all extended state variables. Each of these variables is in a finite, countable set, an infinite, countable set, or an infinite, uncountable set.
- $s_{ext,0}$ is the initial configuration of the extended state.
- O needs to be extended to allow for extended state assignments
- $T \in S \times S_{ext} \times (E \cup E_s) \times G \times (O \cup E_s)^* \times S \times S_{ext}$

While State Charts introduce the notion of compound states and parallel states, a formal difference to finite state machines discussed here, it is always possible to flatten a state chart to a state machine using various approaches [8], [9]. Thus, state charts and machines are mathematically equivalent, but state charts are more expressive, minimizing the number of states needed to model a system.

Finally, we can extend our model further with continuous-times and time-discrete models that take effect in certain states and thus achieve a hybrid state chart model capable of modeling any WoT CPS, including its physical behavior. A continuous-time model M is defined as:

$$M : P_d \times I_d \rightarrow O'_d \tag{1}$$

where

- $P_d = dom(P^*)$; $P \subseteq S_{ext}$ is the set of domains of time-invariant parameters from a subset P of S_{ext},
- $I_d = dom(I^*)$; $I \subseteq S_{ext}$ is the set of domains of time-variable inputs from a subset I of S_{ext},
- $O'_d = dom(O'^+)$; $O' \subseteq S_{ext}$ is the set of domains of time-variable outputs from a subset O' of S_{ext}. The apostrophe was added to distinguish this set from the set O defined for state machines.

To utilize our extended state chart for modeling WoT CPSs, a mapping between the semantics of both models needs to exist. As such, we consider the interaction affordances model of the WoT. A TD can be considered as a set containing three disjoint sets of interaction affordances, i.e.:

$$TD = \{Props, Actions, Events\} \tag{2}$$

where

- $Props$ is the set of all property affordances
- $Actions$ is the set of all action affordances
- $Events$ is the set of all event affordances
- $Props \cap Actions = \varnothing$; $Props \cap Events = \varnothing$; $Actions \cap Events = \varnothing$

Each of these affordances may allow a certain set of operations to be performed on them. We define a function Op that outputs the operations of a certain affordance as follows:

$$Op : Props \rightarrow \{read, write, observe, unobserve\}^+ \tag{3}$$

$$Op : Actions \rightarrow \{invoke\} \tag{4}$$

$$Op : Events \rightarrow \{subscribe, unsubscribe\}^+ \tag{5}$$

In WoT, each operation is performed by sending a request. The arrival of the request to the Producer can be modeled as an event trigger, causing a transition in the state chart, as discussed before. The request's payload is sent along with the event trigger for processing in the state chart. As such, it is possible to model systems in which the API exposed by the Producer depends on the current state. However, some operations may not necessarily change the state of the state chart and do not need to be modeled as a transition. These operations are $\{read, observe, unobserve, subscribe, unsubscribe\}$, as these are the operations meant for getting data from a Producer and usually do not change the state. Therefore, it should be possible to model these operations without having to tie them to an event trigger if not needed. This implies that if an operation is not tied to an event, it can be performed without any restrictions. However, this may not be the desired model. As such, there needs to be a way to specify in which states an eventless operation can be performed.

Property Affordances represent operations that can be performed on a Producer's internal state variables. In a state chart, these are either a subset of its compound states S_{com} or a subset of its variables in the extended state S_{ext} or formally:

$$Props \subseteq S_{com} \cup S_{ext} \tag{6}$$

This exact subset needs to be specified in the model.

On the other hand, Event Affordances represent a subset of all internal signals that a state chart raises, or formally:

$$Events \subseteq E_s \tag{7}$$

This subset needs to be specified in the model as well.

Finally, each operation can have a synchronous or asynchronous response. A synchronous response contains all the information about the result of the operation, while an asynchronous response usually only signifies whether a request was received successfully or not. If a response is handled synchronously, the state chart must include at least one on-entry, on-transition, or on-exit $send$ action that responds to the desired request.

B. Extending SCXML and TD

While SCXML is capable of modeling all aspects of state charts, it does not provide semantics for modeling WoT interactions. As such, we propose an extension in the form of XML elements that describe the mapping between an SCXML instance and its related TD. The extension does not redefine any of the SCXML specifications but adds the information needed to describe the Thing's interface. Specifically, our extension describes the following:

1) All the interaction affordances that the state chart-modeled Thing provides.
2) Each operation that an affordance provides and the following information about it based on the detailed discussion in Section III-A:
   a) The event $e \in E$ that the request raises in the state chart once received. This is mandatory for an $invoke$ operation but optional for all others.
   b) If an event is not specified, the operation can still be modeled as state-dependant, available only in the specified states.
3) Schemas for payloads as SCXML does not provide a way to describe schemas for its events and messages, nor the domains of its extended state.
4) If a property affordance is tied to a state $s \in S$ or a variable $s_{ext} \in S_{ext}$ following Equation 6.
5) Which state chart signal is an event affordance tied to following Equation 7.

The proposed extension is a child element of the root element of the SCXML **<scxml:scxml>** called **<wot:affordances>**.

<wot:affordances> can have three types of children **<wot:property>**, **<wot:action>** and **<wot:event>**.

A **<wot:property>** must have either the attribute `stateElement` or `dataElement`. The value of the `stateElement` attribute is a string pointing to the `id` of either a compound **<scxml:state>** or **<scxml:parallel>** element. Similarly, the value of the `dataElement` is a string pointing to the `id` of an **<scxml:data>** element, which is a child of the **<scxml:datamodel>** that represents the state charts extended state.

A **<wot:event>** element must specify the attribute `emitEvent`, whose value is a string that points to an event sent by the state chart using the **<scxml:send>** element.

Each of the three elements above can have two types of child elements: **<content>** element, which can occur at most once, and **<wot:op>**. The **<content>** element can only have text content as a child and is meant to be populated with the JSON description of the interaction affordance according to the TD syntax. The **<wot:op>** has the required attribute `type` and the optional attributes `event`, `stateDependant` and `availableIn`. `type` is a string value that marks the types of the affordance. Possible values are the same values for the `"op"` keyword in the TD such as `"readproperty"`, and `"invokeaction"`. The `event` string attribute ties the operation to a transition event in the state chart. If `event` is not specified, the optional boolean attribute `stateDependant` marks the operation as state-dependant and the string attribute `availableIn` lists the `id`s of the states accordingly as a space-delimited list.

With these extensions, we fulfill all the requirements for describing highly dynamic APIs in a declarative manner. Additionally, we want to capture physical behavior in the model as well and, therefore, introduce an additional element **<wot:model>**. This element can only exist as a child element of **<scxml:onentry>** inside a **<scxml:state>** or **<scxml:parallel>** element. It has only one required attribute `id` and can have two types of child elements: a **<content>** element, which has to occur exactly once, and one or more **<variable>** elements. The **<content>** element can only have text content as a child, and that content must be a Modelica model. **<variable>** elements have two required string attributes `name` and `location` and no children. The **<variable>** elements are meant to map **input**, **parameter** and **output** variables of the Modelica model to a corresponding **<scxml:data>** element. The `name` stands for the variable name inside the Modelica model, and the `location` points to the `id` of the corresponding **<scxml:data>**.

With these extensions in place, we are capable of modeling all aspects of the hybrid WoT CPSs as discussed in the Section III-A.

We also propose a set of extensions for the TD itself to accommodate the added information about a Thing's behavior. Each interaction affordance that includes operations restricted based on the current state of the state chart has an additional keyword `"scxml:{{interactiontype}}"`. The value of this keyword is an object containing each restricted operation as a JSON property. These, in turn, are objects that contain the properties `"affects"`, `"availableInState"` and `"event"`. The `"affects"` property lists all the property affordances that are affected or changed by performing this operation. The `"availableInState"` property lists all the state variables restricting the operation that are exposed as property affordances, as only exposed states can be read and checked by Consumers. The `"event"` property ties the operation to the specific event inside the linked SCXML file. Additionally, each of the states listed in `"availableInState"` may appear as a keyword, and its value is an array listing all conditions and guards that are needed to perform the transition. All the additional information provided inside the TD is automatically generated from the SCXML file and is meant to help developers build applications more easily and with fewer mistakes or undefined behaviors.

C. Code Generation

The extended SCXML we propose includes enough information about the described Thing's behavior and its exposed API. As such, it can be used to automatically generate both the extended TD and the underlying application logic and behavior of the Thing using only the SCXML file as input. We are not interested in discussing the generation of the interpreted state chart, i.e. the code that implements the state machine itself and the transitions based on events. Rather, we are interested in discussing the process of pre-processing the state chart for simulation purposes as well as the process of generating the extended TD.

Once the SCXML document is parsed, we check all the operations under the **<scxml:affordances>** element. While we allow property writes not constrained by state or tied to an event, under the hood, all assignments must be triggered by an event in the state chart. This means an additional transition is needed to permit writing properties without restrictions. If such an operation is detected while parsing, a wrapper **<scxml:parallel>** state is added to the state chart on the top level that has both the original state chart and an additional **<scxml:state>** as children. The added child **<scxml:state>** has a transition to itself that detects property write events and assigns the input of the operation to the correct **<scxml:data>** location. A similar thread of reasoning is needed when handling Modelica models. Whenever a Modelica model is detected, wrapper **<scxml:parallel>** state is added that has an additional child **<scxml:state>** acting as a clock that triggers tick events at constant intervals and allows the start chart to handle variable updates. As a result, the state chart can also model the notion of time in a quantized fashion.

To generate the extended TD, we perform the following step:

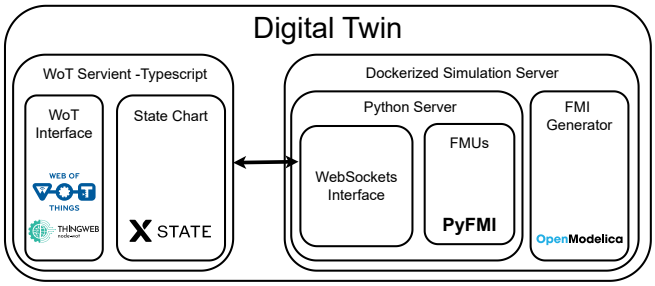

Fig. 2: Our tool generates a Digital Twin (DT) as a combination of a NodeJS script and a dockerized environment acting as a simulation server. The NodeJS script, written in Typescript, uses the **node-wot** library to handle the WoT stack generation, and the **XState** library to handle the implementation of the state chart. The dockerized environment uses **OpenModelica** for generating FMIs from the Modelica models, which are then parsed by a Python script using the **PyFMI** and executed as FMUs. The NodeJS script and the simulation server are connected using WebSockets.

1) We gather all the operations that are listed under `<scxml:affordances>` in an array and include its interaction type, the `event` it is tied to if applicable or otherwise the states listed in the `availableIn` attribute.
2) While parsing the `<scxml:transition>` elements, we check if this transition is triggered by an `event` tied to an operation in our array.
3) If the previous condition holds true, we check if the state that the `<scxml:transition>` element is in is the child of a compound `<scxml:state>` or `<scxml:parallel>` that is mapped to a property affordance.
4) If the previous condition holds true, we now add the `id` of that state to the `"availableInState"` of the corresponding entry in the TD and we add the name of the corresponding property affordance to the `"affects"` keyword.
5) We check if the transition has any `<scxml:assign>` elements that point to a `<scxml:assign>` element which is mapped to a property affordance. If that is the case, we add the name of the associated property affordance to the `"affects"` keyword.

IV. Implementation

We have implemented our methodology as a fully open-source solution[1] that is able to parse an extended SCXML file, generate the code for implementing the state chart logic described in the state chart, generate the extended TD based on the SCXML file and generate the WoT stack that handles exposing the interaction affordances.

Furthermore, our solution can automatically generate FMUs for the models inside the extended SCXML and execute them as part of the state chart.

Our solution is built on top of **node-wot**[2], the reference implementation of the WoT Scripting API, **XState**[3], a library for handling state charts in Typescript, **OpenModelica**[4], an open-source modeling and execution environment for Modelica models, and **PyFMI**[5], a Python-based library for executing FMUs. The exact architecture can be viewed in Figure 2.

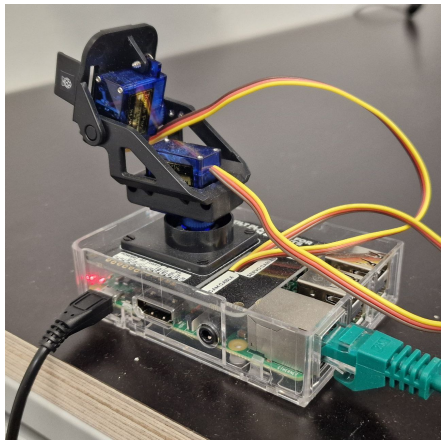

Fig. 3: Shown here is the Pantilt-HAT Module used in our second evaluation. The small robot consists of two servo motors, one for panning and one for tilting. Each servo motor can be controlled separately.

Our implementation is written in Typescript. After parsing the SCXML document, our solution extracts all Modelica models inside the state chart to different `.mo` files, automatically generates the XState representation of the state chart, and generates the node-wot interface that provides the WoT interaction affordances. The communication between the WoT interface and the state chart is done using an eventing system, where the WoT interface would send events coupled with payloads to the state chart. The state chart would raise events inside an event bus to signal if a request's response is available, sending the appropriate response body as well. If an operation is not allowed in the current state of the modeled Thing, the generated DT automatically rejects the request, stating that the current operation is not allowed.

When executing a state chart that includes Modelica models, a dockerized environment that includes OpenModelica and PyFMI is used to handle the simulation. First, the `.mo` files are copied over to the docker environment and converted to ME FMUs using OpenModelica. Then, these models are loaded PyFMI in a Python Script that also provides a WebSockets interface for starting, resetting, stepping through, and terminating each FMU. All FMUs implement an explicit Euler solver.

V. Use Cases and Evalution

To evaluate our approach and implementation, we showcase the models of two real-world devices using our proposed extended SCXML and TD. The first device we model is Schneider Electric's Altivar 320 Variable Frequency Drive[6]. Specifically, we model the operating state diagram defined in the related Modbus manual [10]. The operating state diagram defines a set of operating states and a set of strict transitions that need to be performed in a certain sequence to reach the operating state. Normally, the TD is not expressive enough

[1] Available through link [2] https://github.com/eclipse-thingweb/node-wot
[3] https://stately.ai/docs/xstate [4] https://openmodelica.org/
[5] https://pypi.org/project/PyFMI/ [6] https://www.se.com/us/en/product-range/63440-altivar-

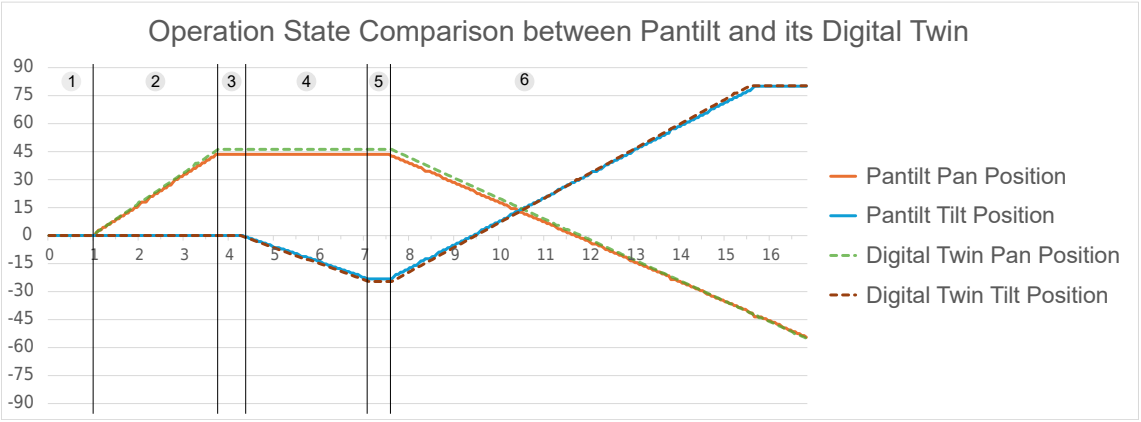

Fig. 4: This graph shows the comparison between the pan and tilt positions of the Pantilt module and its DT. The numbering follows the sequence of interactions described below. The continuous lines represent the positions of the real motors, dashed lines are for the DT. Please note that the timing is not fully accurate due to the NodeJS environment not guaranteeing exact timeouts.

to describe these restrictions on the operations. However, using our proposed extensions, we were able to fully model the operating state diagram, ensuring operation can only be performed in the appropriate drive state and are able to generate a reactive DT to test the behavior of the device and build applications without access to an actual device. The resulting state chart and DT are omitted for brevity but are accessible in our repository.

To test our approach regarding modeling the physical behavior of IoT, we modeled a small 2-Degrees of Freedom (DoF) robot attached to a Raspberry Pi 3, shown in Figure 3. One servo motor is used for panning, while the other is used for tilting. We modeled the state chart such that the DT would have the same exact interface as the physical device, i.e. both the physical device and TD provide the same operations. We described the motion profile of the two servo motors using Modelica models in the extended SCXML. Each servo motor can be triggered to move separately, and therefore, both motors were modeled as children to a `<wot:parallel>` state. We then tested the accuracy of the generated DT's behavior compared to the physical device by sending the same requests to both the physical device and the DT and comparing the position of both servo motors of both entities. The sequence of interactions performed was:

1) Waiting for 1 s
2) Panning Continuously for 3 s with a speed of 15 °/s
3) Stopping any movement and waiting for 0.5 s
4) Tilting Continuously for 3 s with a speed of -8 °/s
5) Stopping any movement and waiting for 0.5 s
6) Panning and tilting continuously with a speed of -10 and 12 °/s, respectively, for 10 seconds.
7) Stop logging

The resulting motion of the real motors and their DTs can be viewed in Figure 4. The numbering on top of the figure corresponds to the sequence of interactions listed above. The position of the servo motors was polled every 50 ms. The resulting graph shows the alignment of both motion profiles to a very high degree. The deviation between the actual and the virtual position was calculated to be 1.54° and 0.73° for the pan and tilt position, respectively. The maximum deviation was 3.7° and 2.7° for pan and tilt position, respectively.

As such, we conclude that our method provides a viable solution for modeling WoT CPSs accurately, allowing the generation of highly representative DTs that can be used for development, simulation, testing, and prediction.

VI. Related Work

Modeling and capturing the behavior of devices and even entire systems in the context of the WoT has been a continuous endeavor in the last decade.

[11] introduced a model-driven approach for designing and implementing WoT Servients, promoting an easier and more streamlined approach for developing WoT Things on top of the Eclipse Modeling Framework. The model can be designed using a GUI, and given a model, the framework is able to generate the device's implementation code. However, the modeling framework only implements the WoT interface of the Thing and is not exchangeable. The Thing's behavior needs to be implemented separately. This stands in contrast to our methodology that aims to provide an exchangeable modeling format that models both the behavior and WoT interface.

[12] proposes a description format called the System Description (SD) for modeling entire WoT mashups and the behavior of the mashup controller. The SD has a similar structure to a TD with added keywords to describe execution paths as an ordered list of WoT interaction. Being a JSON-LD document, it is exchangeable and machine-readable and can be converted to an equivalent UML Sequence Diagram representation for viewing. It is also used to automatically generate the application logic for the mashup controller.

[13] aims to capture the behavior of physical entities by modeling Things as Markov Decision Processes, learning the behavior as probabilistic state-action transitions similar to a state machine. The method assumes that Thing's state-space is spanned by its properties. As properties can have infinite and uncountable domains, they assume all properties to be discrete, limiting the applicability of the method to more constrained and limited devices. By training the model, it is capable of

generating a Digital Twin that exhibits or mimics the same behavior as the real device.

Outside the context of the WoT, there have been other efforts to use exchangeable data formats for automatically generating simulation-based DTs.

Utilizing a Platform Industrie 4.0 standard called the Asset Administration Shell (AAS), several approaches were made to utilize the standard as an exchangeable format for simulation-based engineering [14] and DT generation [15], using FMUs directly to exchange simulations. While none of these approaches targets WoT or IoT, specifically, an integration of WoT in the AAS has been proposed [16]. As such, approaches utilizing the AAS may, in the future, utilize the TD as well.

In contrast to current related work, our approach aims to capture the behavior of highly dynamic and complex systems in a declarative, human- and machine-readable format, that can be exchanged and utilized by Consumers and clients. Using our approach, it is now possible to describe the effect of interactions on the state of the described Thing, which was not possible using only the TD. With the added information, it is now possible to develop WoT Mashups that can reason and handle statefulness more reliably or develop goal-driven frameworks that can traverse the state chart to achieve the goal of being in a certain state.

VII. Conclusion

In this paper, we introduced our novel approach **Stateful-WoT**, a building block in the WoT environment that provides a standardized behavior description, which is human- and machine-readable, extensible, exchangeable, and built using existing standards. Our method utilizes SCXML and Modelica models together to describe the operational behavior of WoT CPSs, including their physical behavior. We formally discuss the extensions to fully realize our vision and how this is executed using the aforementioned standards. We also introduce extensions to the WoT TD to describe dynamic systems based on their state charts. We then introduce our fully open-source implementation of the proposed methodology based on node-wot, OpenModelica, and PyFMI and evaluate it in two use cases. In the first case, we generated a DT that mimics the operational behavior of an industrial motor drive, ensuring that the generated DT does not allow interactions that are not possible based on the drive's state. In the second use case, we modeled a small Pantilt module attached to a Raspberry Pi 3 and compared the actual physical behavior to the model of the generated DT. We show that the generated model mimics the behavior of the real device to a high degree and can, therefore, be used as a viable DT for simulation, testing, and prediction. We discuss related work and explain that, in contrast to current literature, our approach allows the handling of more complex systems and scenarios due to including a declarative description of the statefulness of Things.

Future work includes a deeper look into manageable actions in the context of the WoT, a user-friendly tooling for our methodology that provides a GUI for modeling WoT Things, and utilizing the extended SCXML document alongside the extended TD for automated Mashup generation using goal-driven approaches.

References

[1] M. Lagally, R. Matsukura, M. McCool, and K. Toumura, "Web of Things (WoT) Architecture 1.1," Dec 2023. [Online]. Available: https://www.w3.org/TR/wot-architecture11/

[2] S. Käbisch, M. McCool, and E. Korkan, "Web of Things (WoT) Thing Description 1.1," Dec 2023. [Online]. Available: https://www.w3.org/TR/wot-thing-description11/

[3] M. Koster and E. Korkan, "Web of Things (WoT) Binding Templates," Sep 2023. [Online]. Available: https://www.w3.org/TR/wot-binding-templates/

[4] J. Barnett, R. Akolkar, R. Auburn, M. Bodell, D. C. Burnett, J. Carter, S. McGlashan, T. Lager, M. Helbing, R. Hosn, and et al., "State Chart XML (SCXML): State Machine Notation for Control Abstraction," Sep 2015. [Online]. Available: https://www.w3.org/TR/scxml/

[5] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987. [Online]. Available: https://doi.org/10.1016/0167-6423(87)90035-9

[6] S. E. Mattsson, H. Elmqvist, and M. Otter, "Physical System Modeling with Modelica," *Control Engineering Practice*, vol. 6, no. 4, pp. 501–510, 1998. [Online]. Available: https://doi.org/10.1016/S0967-0661(98)00047-1

[7] T. Blochwitz, M. Otter, J. Akesson, M. Arnold, C. Clauß;, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauß;, D. Neumerkel, and et al., "Functional Mockup Interface 2.0: The Standard for Tool Independent Exchange of Simulation Models," in *Proceedings of the 9th International MODELICA Conference*, Nov 2012. [Online]. Available: http://dx.doi.org/10.3384/ecp12076173

[8] A. Wasowski, "Flattening statecharts without explosions," in *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 257–266. [Online]. Available: https://doi.org/10.1145/997163.997200

[9] X. Zhan and H. Miao, "An Approach to Formalizing the Semantics of UML Statecharts," in *Conceptual Modeling – ER 2004*, P. Atzeni, W. Chu, H. Lu, S. Zhou, and T.-W. Ling, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 753–765. [Online]. Available: https://doi.org/10.1007/978-3-540-30464-7_56

[10] "Altivar Machine 320 Variable Speed Drives for Synchronous and Asynchronous Motors Modbus Serial Link Manual," p. 42, April 2016. [Online]. Available: https://download.schneider-electric.com/files?p_Doc_Ref=NVE41308&p_enDocType=User+guide&p_File_Name=ATV320_Modbus_manual_EN_NVE41308_01.pdf

[11] M. Iglesias-Urkia, A. Gómez, D. Casado-Mansilla, and A. Urbieta, "Enabling easy Web of Things compatible device generation using a Model-Driven Engineering approach," in *Proceedings of the 9th International Conference on the Internet of Things*, ser. IoT '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3365871.3365898

[12] A. Kast, E. Korkan, S. Käbisch, and S. Steinhorst, "Web of things system description for representation of mashups," in *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, 2020, pp. 1–8. [Online]. Available: https://doi.org/10.1109/COINS49042.2020.9191677

[13] L. Sciullo, A. Trotta, F. Montori, L. Bononi, and M. Di Felice, "WoTwins: Automatic Digital Twin Generator for the Web of Things," in *2022 IEEE 23rd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2022, pp. 607–612. [Online]. Available: https://doi.org/10.1109/WoWMoM54355.2022.00095

[14] S. Heppner, T. Miny, T. Kleinert, M. Becker, K. Schmitz, and R. Alt, "Asset Administration Shells as Data Layer for Enabling Automated Simulation-based Engineering," in *2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2023, pp. 1–7.

[15] D. Göllner, T. Pawlik, and T. Schulte, "Utilization of the asset administration shell for the generation of dynamic simulation models," in *2021 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, 2021, pp. 808–812.

[16] H. K. Pakala, K. O. Oladipupo, S. Käbisch, and C. Diedrich, "Integration of asset administration shell and Web of Things," 2021. [Online]. Available: http://dx.doi.org/10.25673/39570