Safe Interoperability for Web of Things Devices and Systems

Ege Korkan, Sebastian Kaebisch, Matthias Kovatsch, and Sebastian Steinhorst

1 Introduction

The Internet of Things (IoT) brings connectivity to electronic devices and allows them to connect with each other. Due to the large variety of IoT devices and application scenarios, they all bring their own properties such as different processing speed or range of connectivity, desired run-time or energy consumption, safety features, etc. This creates a fragmentation in IoT, with different standards to interact with the devices and to represent them, each optimized for a specific application area or device type. Consequently, such fragmentation hampers composing applications beyond the functionality of the individual devices.

In the electronic design community, languages such as SystemVerilog have proven to be an effective standardized representation for the entire development cycle, from design to verification and for a very wide range of application areas. However, in the IoT domain, companies introduce siloed IoT platforms that come with proprietary standards even within similar application domains.

Consequently, there is a necessity that an IoT device can be represented with a description of capabilities, which can be understood and interpreted by other devices and standards. Here, a common ground can be created by enabling to describe an

E. Korkan (✉)
Technical University of Munich, Munich, Germany
e-mail: ege.korkan@tum.de

S. Kaebisch · M. Kovatsch
Siemens AG, Munich, Germany
e-mail: sebastian.kaebisch@siemens.com; matthias.kovatsch@siemens.com

S. Steinhorst
Technical University of Munich, München, Bayern, Germany
e-mail: sebastian.steinhorst@tum.de

T. J. Kazmierski et al. (eds.), *Languages, Design Methods, and Tools for Electronic System Design*, Lecture Notes in Electrical Engineering 611,
https://doi.org/10.1007/978-3-030-31585-6_3

interface to different standards in a well-defined representation. For this purpose, the Thing Description (TD) [1] was introduced recently as an open description format for devices with connectivity of any kind that is human-readable and machine-understandable. The TD is not a standard to replace other IoT standards, but it enables to describe them through syntactic and semantic information.

Consider a temperature sensor used with a cloud IoT platform and a local ventilator. Between them, TDs enable to create a temperature-controlled ventilation system directly composed of the capabilities of these two physical devices. The advantage of such interoperability for machine-to-machine communication is to enable system functionality without prior knowledge about the interfaces between the devices.

Such a sensor's functional capability, data structure, and access points will be referenced in the TD of the sensor. Hence, the ventilator will be able to access the sensor data due to the provided access points and will be able to understand the data due to the data structure described in the TD.

The previous ventilation system example is abstracted in Fig. 1. This system has three IoT devices, each possessing a TD. Within the system, each IoT device, to which we will in the following refer to as a Thing,[1] can read the TD of another Thing and interpret it to understand the information such as the Thing's interactions, supported protocols, data structure, how to access the data, etc., as described in the column on the right of Fig. 1 (TD Contents). During the course of this paper, an exposer Thing accepts requests provided in its TD, whereas the consumer Thing reads a TD and interacts with the exposer Thing.

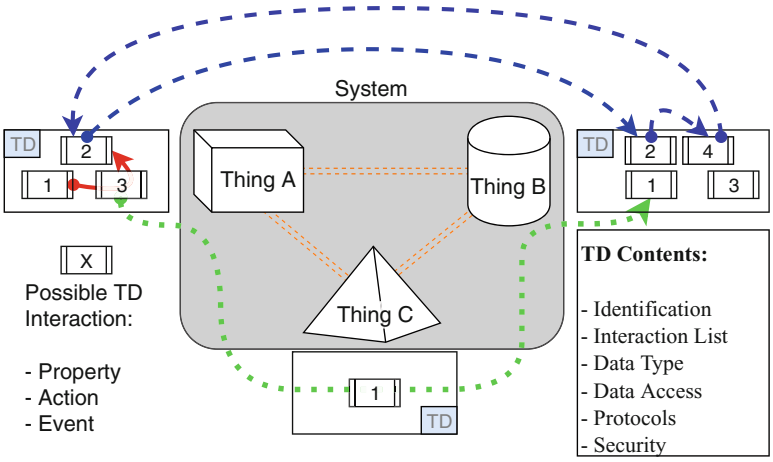

Fig. 1 An abstracted view of an IoT System with three IoT Devices each with an associated Thing Description (TD). The arrows demonstrate composition of greater functionality than the devices themselves, necessitating sequential behavior between devices

[1]When the word Thing is used with a capital letter, a Thing means an object, either virtual or physical, that can be communicated with.

An interaction is the description of a specific capability of the Thing, representing the data structure, access protocol, and access link. For example, reading the temperature value is such an interaction with the Thing. Similarly, rotating the fan is also an interaction that acts on the physical world. In a TD, one would find a list of interactions and how to access them. Interactions are illustrated by numbered boxes in Fig. 1 and they will be explained in Sect. 2 in more detail.

In Fig. 1, `Thing A` has three interactions and all these interactions can be used by `Thing B` and `C` to interact with `Thing A`. Referring to the temperature-controlled ventilation system example, interaction 1 of `Thing A` can be reading the temperature value and the interaction 4 of `Thing B` can be rotating the fan.

Problem Statement With the current TD standard, it is possible to build the system described in Fig. 1. However, the behavior represented by arrows has to be programmed manually, which results in an implicit description of the device or system.

An interaction can change the state of the Thing, making it accept only certain interactions (state transitions). For example, the red (continuous) arrow is a sequence describing such state transitions of `Thing A`. This can be requirements of sequential behavior, such as initializing the motor driver of the ventilator before setting a rotation speed. In order to execute this sequence of interactions, since such a sequence is not described in the TD, the person who implements the compositional system needs to have access to an operation manual of `Thing A`. This manual should describe the internal workings of the Thing (e.g., with a state machine) and give meaning to the causality between interactions.

Similarly, the green (dotted) and blue (dashed) arrows in Fig. 1 illustrate sequential behavior between multiple Things and are not expressed anywhere, thus need to be implemented manually. For example, we would like to express that the green (dotted) arrow represents the aforementioned temperature control functionality in the correct order and with a causal relation: reading a temperature value and then rotating the ventilator. This shows that executing multiple interactions can provide another meaning that is not previously given in a single interaction. To solve this problem, a new interaction can be implemented that provides the same meaning of executing multiple interactions. This is possible during the development phase of Things, but for non-reprogrammable, legacy devices there is no such option.

Contributions In order to avoid that each interaction is executable at any given time or multiple interactions can be executed in any given order, in this paper, we propose the specification of sequential behavior within TDs. The ability to represent valid sequences of interactions, which we call **paths**, in the TD of a device enables the designer of this device to restrict interactions and hence simplify the interaction of other devices with this device. Without such paths, arbitrary sequences of interactions could be triggered, which would either require knowledge about the inner workings of the device or create an unsafe and erroneous behavior.

Consequently, in the context of TDs introduced in Sect. 2, this paper has the following contributions:

- We extend our initial path vocabulary[2] contribution of [2] in Sect. 3.1 that uses the JSON Pointers instead of `hrefs`. This enables stronger semantics for describing how to interact with Things.
- We show that a system can be composed through sequential interactions of multiple Things by using the same **path** logic, presented in Sect. 3.2.
- We demonstrate a case study with sequential behavior in an industrial automation system composed of an industrial fan, a temperature sensor, and a system controller in Sect. 4.

Related work is discussed in Sect. 5, a discussion on an application of our methodology is provided in Sect. 6, and Sect. 7 concludes.

As the applications of TDs diversified, two new observations motivated us to improve our path vocabulary:

- Some Web of Things (WoT) devices use the same Uniform Resource Identifier (URI) as `hrefs` for multiple interactions where the interactions are differentiated via the method they require. For example, the Philips HUE [3] lights specify that sending an HTTP GET request to /api/<username>/lights would return all the light states and information, like a Property Interaction of a TD. However, sending an HTTP POST request to the same URI would start a search for new lights.
- In some cases, one interaction can have multiple forms that serve different purposes, such as one for observing possible updates of a sensor's measurement and one to get the current value. These forms can use the same `href` value or even use different protocols.

These new discoveries motivated us to abstract how the path vocabulary is serialized and not use URIs of `hrefs` in the path serialization. We have opted for the JSON Pointers standard as specified in [4] which is still in the URI format. JSON Pointers point to a specific place in a JSON document and in our contribution we use them to point to a specific `form` in the TD document.

2 Thing Description

The TD approach has been introduced in September 2017 (First public draft) by the WoT Working Group of World Wide Web Consortium (W3C). This section will explain the TD approach, but most importantly, its shortcomings and why our contribution is necessary to enable TDs to describe more complex cyber-physical

[2]The term vocabulary is used here since the TD standard [1] refers to actions, properties, etc. as a vocabulary.

systems. In the following, we will mainly focus on the relevant details of TDs for the context of our contribution, the proposed path vocabulary.

The path vocabulary that will be introduced in Sect. 3 describes a series of interactions. Further information on the characteristics of interactions is thus required before introducing this vocabulary. In this section, we will define interactions in order to argument the need for describing sequential behavior.

An interaction I can represent two types of messaging patterns: request–response (Definition 1) and publish–subscribe (Definition 2).

Definition 1 (Request–Response) For a request $p \in$ client and a $q \in$ server, the pair is defined as follows:

$$p \Rightarrow q \tag{1}$$

Definition 2 (Publish–Subscribe) Notifying an event only in matching subscription intervals is defined by Baldoni et al. [5] as follows:

$$\forall e \in \text{nfy}(x) \in h_i \Rightarrow \text{nfy}(x) \in S_i(C) \text{ s.t. } C(x) = \top \tag{2}$$

with

- e, the event the subscriber subscribed to;
- x, the information generated from the process;
- nfy, the notification of the information;
- h, a local computation that generated x;
- S, the interval between subscription and unsubscription;
- C, the subscription request by the subscriber;
- $\top$, the pattern of the event to subscribe to at the server side.

These formal definitions for interactions are mentioned in the TD standard [1] in three groups:

- Properties: A value provided by the Thing, such as sensor data, or values provided to the Thing, such as a desired temperature. This matches the request–response pattern.
- Actions: Requesting the Thing to do something that interacts with the physical world or with other Things that also takes some time, such as turning on a fan or LED. This matches the request–response pattern.
- Events: A message triggered due to a change in the Thing and sent to the consumer Things that have subscribed to it, such as an overflow alarm. This matches the publish–subscribe pattern.

In order to illustrate the different types of interactions in a practical example, we are showing a simplified TD of a ventilator in Listing 1. This ventilation Thing, as described by its TD, can rotate the motor of the ventilator at a given speed provided by the consumer Thing. It also has safety features such as requiring initialization by

the consumer Thing. In addition, in case of an overheating of the motor, it can notify the consuming Things who are subscribed to this notification.

Other than interactions, the TD provides identification information. In the order of appearance in Listing 1, the `title` provides a human-readable reference (identification) for this Thing, whereas `id` provides a unique identification for the Thing that stays unchanged through different networks or IP addresses. Similarly, the `base` (line 3) describes the protocol and the URI needed to communicate with this Thing.

By using the default protocol bindings described in [6], one can interact with the previously introduced ventilator in the following sequence:

- Read or write the rotation speed of the ventilator by reading/writing the `rotation` property (lines 6–10). Here, it is specified that the data structure should be an `integer`.
- Rotate the ventilator by invoking the `rotate` action (lines 13–15). This action can be invoked without sending any specific data and the response will not contain an `integer` as in the previous property.
- Initialize the motor driver by invoking the `initialize` action (lines 16–19). Here, it is specified that the data structure of the response should be a `string`.
- Subscribe to the `overheating` event (lines 22–25) and get notified if the motor heats up too much. The structure of the data received will be a `string` data structure.

This ventilation Thing represents a sequential behavior that is not explicitly described. If one reads and learns the internal workings of the Thing, it is specified that in order to rotate the motor, one needs to invoke the `initialize` action (lines 16–19). This problem is commonly encountered in cyber-physical systems and is illustrated in an abstracted fashion in Fig. 2. Generally, a consumer Thing reads a TD, understands what can be done with the associated Thing, sends a chosen request to execute the interaction, and waits for the response from the Thing. The orange (dashed) arrow `Choose Interaction` is thus handled implicitly by the Thing Y (consumer) and there is no vocabulary provided by Thing X that tells the consumer to execute interactions in a specific order. Without the contribution of this

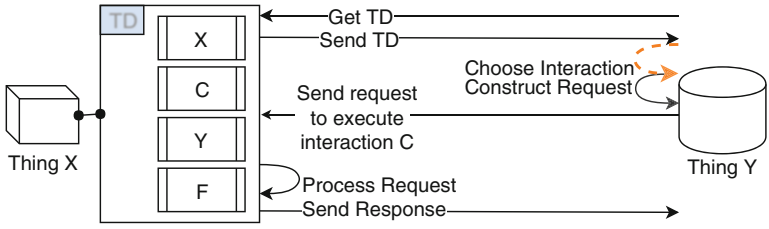

Fig. 2 Request–Response sequence abstraction that can be used for interacting with a Thing. The orange (dashed) arrow demonstrates the missing part of the TDs, which is the problem addressed in this paper

paper, Thing Y's developer had to know the internal workings of Thing X. With our contribution, presented in the following section, this becomes a more systematic and guided process.

```
{
  "title": "MyVentilator",
  "@context": "https://www.w3.org/2019/wot/td/v1",
  "id": "urn:dev:ops:32473-ventilator-1234",
  "securityDefinitions": {
    "basic_sc": {"scheme": "basic", "in":"header"}
  },
  "security": ["basic_sc"],
  "base":"coaps://vent.example.com:5683",
  "properties":{
    "rotation":{
      "type": "integer",
      "forms":[{"href": "/rotation"}]
    }
  },
  "actions":{
    "rotate":{
      "forms":[{"href": "/rotate"}]
    },
    "initialize":{
      "output":{"type": "string"},
      "forms":[{"href": "/init"}]
    }
  },
  "events":{
    "overheating":{
      "data":{"type":"string"},
      "forms":[{"href": "/oh"}]
    }
  }
}
```

Listing 1 Simple Thing Description of a ventilator that exposes the rotation speed, motor initialization, and rotating actions and an overheat alarm that can be obtained from coaps://vent.example.com:5683/td

3 Describing Sequential Behavior

The contribution of this paper is the new path vocabulary that allows to describe sequential behavior. We start this section by listing some requirements of such a vocabulary in the context of TDs. The following subsections start by introducing the vocabulary for single devices and then extend it for systems composed from devices.

Many models for system representation are measured by their expressiveness. In the field of automata theory, there are different levels of expressiveness, from finite automata to Turing Machines.

For cheap and not powerful IoT devices, exhaustive modeling of the inner workings is too tedious. On the other hand, a behavior described in a TD needs to be parsed and understood by such resource-constrained devices. Hence, even if the device providing this representation has enough resources to provide it, the description will not be usable by other IoT devices that are resource-constrained. Furthermore, obliging interacting devices to understand such behavior is contradictory to the design philosophy that internet and web technology enabled in the last decades, which is also applied for IoT.

Often, web pages, services, or Application Programming Interfaces (APIs) are self-descriptive and the user does not need to understand the complete system to start using them. For example, in a simple web page, the user can simply understand the link that he/she is interested in and not look at the rest (e.g., a site-map), i.e., not understand the complete state machine to execute one interaction. Inspired by the success of this logic, it is primordial to follow the same logic for IoT systems and hence for TD, in order to enable easy adoptability and usability.

3.1 Describing Sequential Behavior in a Single Thing

The path vocabulary is based on describing sequential behavior for a single Thing. For this reason, we will formally define the path vocabulary in this section. The formal definition will be then embedded into the TD format and later on used in a system. In order to illustrate the problem and guide this paper, we will be using a state machine of a legacy motor driver of a ventilator, as shown in Fig. 3.

This device cannot be reprogrammed,[3] but requires strict sequential behavior in order to operate safely. A sequence of interactions is needed to make it ready for accepting speed commands or to bring it back to a safe stop.

We can see that the `initialize` action needs to be invoked to initialize the motor. This sets the rotation per minute (rpm) of the motor to 0. However, as a safety feature, the `rotate` action must be explicitly invoked before setting the rotation speed with the `rotation` property. At this point, we can write to the speed value and rotate the motor in a direction. For example, to rotate the motor at 1300 rpm, the following specific order of interactions is needed:

1. Initialize
2. Rotate
3. Write (1300 rpm as value)

A consumer Thing that will interact with this motor driver and that does not know this sequential behavior cannot control the machine the way it is designed.

[3]TDs allow precise description of the capabilities of a device even if the device cannot provide its own TD. In this case, we can use a gateway that stores and provides the TD.

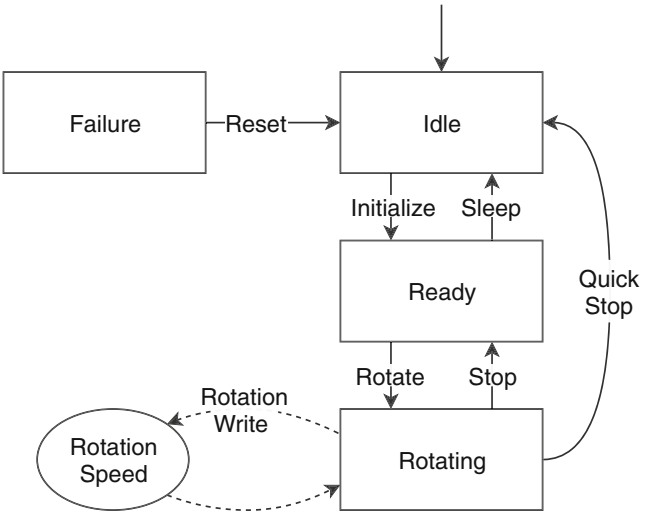

Fig. 3 State machine representation of a legacy motor driver. In order to enable setting the rotation speed to the desired value, Initialize, Rotate interactions have to be executed in this order

Furthermore, if the consumer Thing has access to this specific state machine in a machine-readable format (such as SCXML [7]), understanding the entire state machine for every application should not be necessary. For example, if the motor driver, i.e., the exposer Thing, chooses to expose only a safe stop sequence, the entire state machine that also describes the sequence to rotate the motor would contain unnecessary information.

By contrast, in our path vocabulary, we describe the behavior we want to expose with simple sequential interactions with interaction data that already exist in the TD. The aforementioned path of interactions, named `RotateMotor`, is shown in Fig. 4 along with the state machine from Fig. 3 that was used to generate the paths. We have given other valid path examples from the state machine for illustration.

In order to properly define the path vocabulary we need to introduce four definitions this vocabulary is composed of: path, name, @type, and paths.

Definition 3 (Path) From an ordered sequence of interactions I of sequence length l with $1 \leq i \leq l$, a path π with name t is defined as:

$$\pi_t = I_1, \ldots, I_i, \ldots, I_l \tag{3}$$

Definition 4 (Name) The name of the path is used within the TD to reference the JSON [8] object that contains the path information. Within the TD, the name allows the path to be referenced in the following fashion:

$$\pi_t = \text{derivePath}(t) \tag{4}$$

with derivePath being a function that finds the path t by parsing the TD.

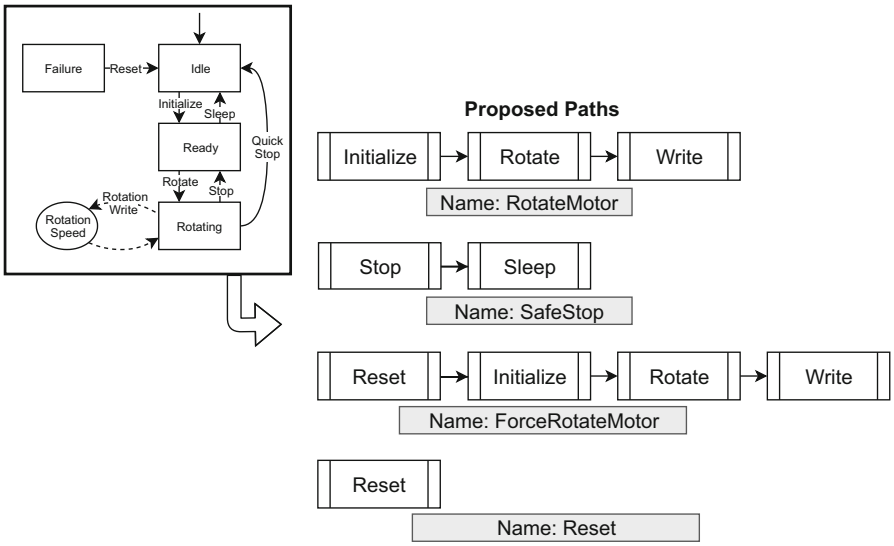

Fig. 4 Illustration of Thing Description paths based on the state machine of a legacy motor driver for an industrial fan. The paths are composed of interactions that execute state transitions. Note that even if the path just contains a single interaction, it is still a valid representation

Definition 5 (@type) The `@type` optionally allows to annotating semantics with the path. It uses the JSON-LD [9] format to reference to another resource on the Web that gives a meaning to the path, making it machine-readable. In a TD, this semantic annotation is given in a compacted form. The value written in `@type` will be combined with a URI in the `@context` field of the TD, exactly the same way as it is combined in the TD standard [1]. Currently used semantic annotations can be found in the iot.schema.org library[4] and used for linking the data.

Definition 6 (Paths) The set of paths offered by the Thing is denoted by Π and defined as follows:

$$\Pi = \bigcup \pi_k \quad | \; \pi_k \in \text{TD} \tag{5}$$

These formal definitions translate to a path description in a TD as shown in Listing 2. Paths are an extension of the TD in Listing 1, with `...` symbolizing the interactions of this TD. This specific TD offers only two paths: `rotateMotor` to rotate the motor from an initial state by executing `initialize`, `rotate`, and `rotation`, as well as `safeStop` that brings the motor to the initial state by executing `stop` and `sleep`, in these respective orders.

[4] http://iot.schema.org/.

The paths in the TD are serialized using the JSON Pointers standard [4]. JSON Pointers are URIs, so they can be easily parsed by Things. In Listing 2, the paths have relative JSON Pointers, where the # sign points to the root of the TD document. After this sign, the path points hierarchically to the form that is a member of the path.

Dealing with Legacy Devices TDs are envisioned for any device that needs to be connected to an IoT system. As we have mentioned before, the motor driver of the ventilator is a legacy device. During the course of this paper, we have used *modern* protocols such as CoAP [10] in the TD listings. However, the advantage of TDs is the capability to describe also older protocols such as Modbus [11], widely used in industrial automation. Such devices might be also non-reprogrammable, which means that they cannot provide a TD themselves. In this case, the TD of such a device has to be retrieved from a database. Thus, the TD of the ventilator has been retrieved from a local database and used by a gateway.

The use of a gateway is necessary to provide access to the functionalities of the legacy device to devices that do not have direct access to the legacy device, such as not supporting the protocol of the legacy device or not having a physical connection. Such a configuration is illustrated in Fig. 5 with `Thing C` as the device that does not have direct access to `Thing A`, the legacy device.

The gateway can then proceed on making the paths of the legacy device simple to use for consumer Things, such as `Thing C`. In the context of IoT, path descriptions should not be imposed to consumer Things that are not part of the system.

We are expecting to see our path vocabulary to be used inside the system and not in the TD of a device such as a gateway. Hence, the TD of the gateway should present simple interactions that should be executable without any causality. In Fig. 5, the path `RotateMotor` becomes an interaction with the same name that will be executed as a normal TD interaction by `Thing C`.

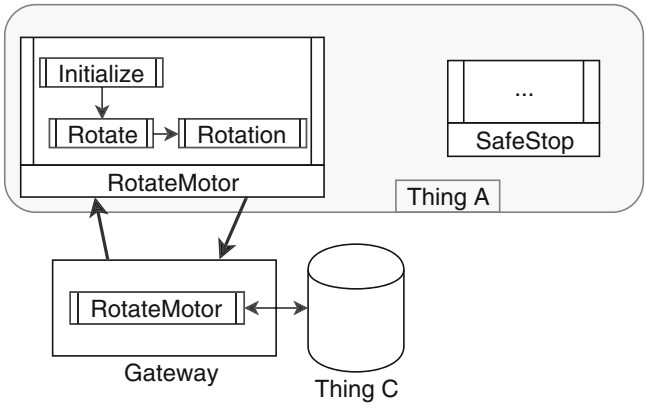

Fig. 5 Using a gateway brings IoT connectivity to a legacy motor driver (Thing A). The gateway can execute a path offered by this device and offer a simple Thing Description action to be executed by Things that do not have physical access to Thing A, such as Thing C

```
{
  "name": "MyVentilator",
  ...
  "paths":{
    "rotateMotor":{
      "@type":"iot:rotate",
      "path":[
        "#/actions/initialize/forms/0",
        "#/actions/rotate/forms/0",
        "#/properties/rotation/forms/0"
      ]
    },
    "safeStop":{
      "@type":"iot:stop",
      "path":[
        "#/actions/stop/forms/0",
        "#/actions/sleep/forms/0"
      ]
    }
  }
}
```

Listing 2 Thing Description of the motor driver with the paths that represent the interaction sequences

3.2 Composing a System

In the context of IoT, we are considering resource-constrained devices that are not able to offer a lot of functionality on their own. This is why composing a system by bringing multiple devices together to orchestrate more functionalities is highly relevant. Consider the system illustrated in Fig. 6, with a `Thing B` that can measure room temperature and another `Thing A`, which is a ventilator, to reduce room temperature. We will illustrate the composition of a system by using the two devices that can control the temperature of a room, bringing additional functionality just by combining their abilities.

We will be using the same path vocabulary introduced in the previous section for this system composition. The path vocabulary is not limited to describe a single Thing, but can be used for a system of Things and the causality between interactions of multiple Things. By using the same vocabulary, we will enable a scalable design approach.

The aforementioned temperature control system can be described by simply using the JSON Pointer URIs from different TDs to describe a system level functionality in a path. Such a path can be executed through a system controller or a Thing of the system. Figure 6 illustrates this system with a system controller where the gateway device takes the responsibility of describing the system behavior and executing system level functionalities.

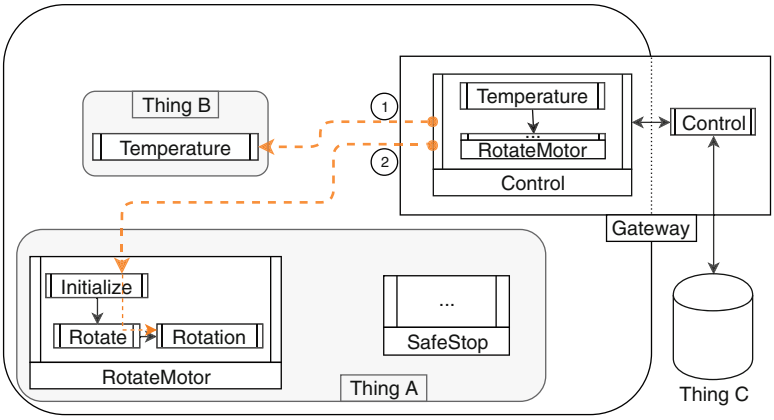

Fig. 6 A gateway can compose a system with the use of the path vocabulary. Here, the system is a temperature control system with a temperature sensor and an industrial ventilator. Things, such as Thing C, that do not have physical access to the system components can execute simple Thing Description interactions to interact with the system through the gateway

The dashed orange arrows in Fig. 6 demonstrate a path executed by the system controller. The system controller is thus able to execute paths or interactions of other devices due to its system controller TD.

Since a path can be also referenced, like an interaction form, with a JSON Pointer, a path and an interaction can be mixed into another path. This is illustrated in Fig. 6 by the `control` path that has the `temperature` interaction and the `rotateMotor` path combined. This means that our path vocabulary can scale well and create a compositional design flow for IoT systems. Listing 3 shows the TD of the gateway illustrated in Fig. 6. The path called `control` can either be offered as an interaction to the consumers of the gateway or directly used, just as the gateway is using the path of the ventilator. As a result, based on thoroughly tested simple interactions and paths, more complex behavior can be described and offered to higher level system controllers.

Note that the URIs have to be absolute URIs in a system controller, since relative URIs lose their uniqueness outside the TD.[5]

[5] A path URI in a TD such as `#/actions/initialize/forms/0` can be combined with the URI of the TD to create a URI that is valid also outside a TD. In this case, it would be coaps://vent.example.com:5683/td#/actions/initialize/forms/0.

```
{
  "id": "urn:dev:ops:32473-controller-1234",
  "title": "SystemController",
  "@context": [
    "https://www.w3.org/2019/wot/td/v1",
    {
      "iot": "http://iot.schema.org/iot"
    }
  "paths":{
    "control":{
      "@type":"iot:temperatureControl",
      "path":[
"http://fdlSensor.com:5683/td#/properties/temperature/forms/0",
"coaps://vent.example.com:5683/td#/actions/initialize/forms/0",
"coaps://vent.example.com:5683/td#/actions/rotate/forms/0",
"coaps://vent.example.com:5683/td#/properties/rotation/forms/0"
      ]
    }
  }
}
```

Listing 3 Thing Description of a system controller/gateway of the temperature control system with a path composed of URIs of interactions of system components

3.3 Worldwide Scalability

As seen in Listing 3 the path URIs can contain domain names that are globally available. These domain names resolve to a particular IP address of a device belonging to the system. However, this device can belong to any network in the world since it is an Internet connected device.

This illustrates that the TD can be used to represent any device in the world; thus, paths can describe behavior of a system composed of devices anywhere in the world. In Fig. 7, we illustrate such a system where a central controller can interact with single Things like `Thing C` (bottom right), systems like in Fig. 6 (top right), or with virtual Things in the cloud like `Thing X` (top left). In this scenario, the central component is able to compose a water level monitoring service that gets weather predictions from a virtual Thing in the cloud in another location, can combine with controlled temperature from a third location, and finally control the water level by pumping water in a fourth and final location.

4 Case Study: Testing with Path Semantics

Ideally, a TD describes what a Thing can do, but it is up to the developer of the Thing to properly implement the capabilities. It is even more difficult to implement everything correctly when designing and implementing a system because of the

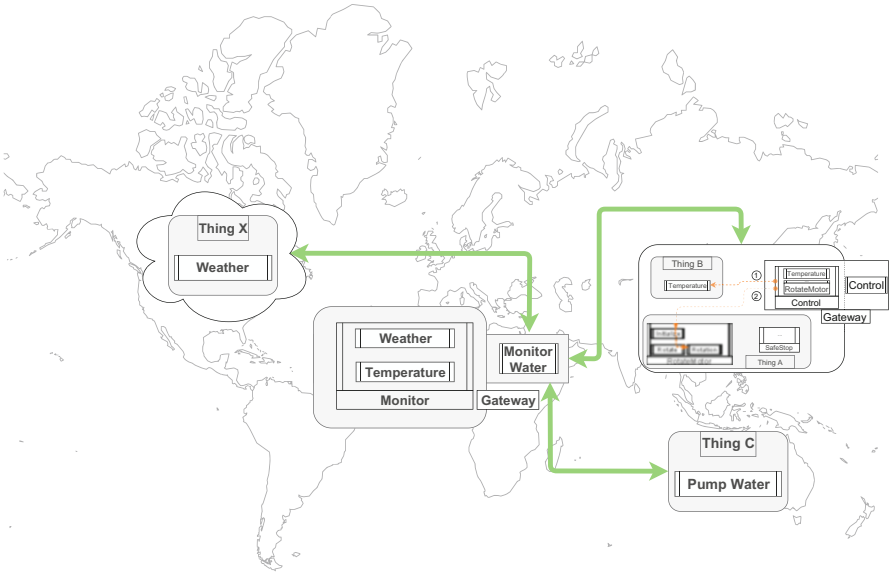

Fig. 7 Another IoT system using other systems, devices, and cloud to compose itself. The ubiquity of Internet and World Wide Web allows Thing Descriptions and the path annotations to be scalable on a worldwide scale

interlinked behavior of devices that compose the system. During both development processes for single Things as well as for systems of Things, testing becomes helpful to detect any errors in the implementation. However, manual testing is a tedious process and for this reason, automatic testing methods are widely used in many application domains.

In a case study, we will show how to apply TDs with the new path vocabulary to facilitate automated testing. In order to show the advantages of our contribution, we will compare the test coverage of our new path-enabled approach to the state-of-the-art testing without paths through an example. Similar to the previous section, we will first present this for a single Thing and then for a system. In the end, an algorithm that is applicable to test both single Things and systems will be shown.

TDs, with or without the path vocabulary, describe exposer Things that the consumer Things will interact with. Since a TD is human-readable, it can be used for specifying a Thing to develop (product), read by the developers who are not familiar with the internal workings of the device during implementation and more importantly, since it is machine-understandable, it can be used for automatic testing to generate test scenarios.

In the following, for automatic testing, we will use the black-box testing approach. In black-box testing, inputs are given to a device under test and the outputs are observed. This type of interaction is equal to a consumer Thing interacting with an exposer Thing. Since the consumer interacts with the exposer based on

the information obtained from its TD, black-box testing of an exposer Thing implementation can be automatized by using its TD.

4.1 Single Thing Testing

We will demonstrate testing a single device with the ventilation Thing introduced earlier in Listing 1. The first case will be without using paths to illustrate the state-of-the-art approach and the second case will apply the path vocabulary.

Testing Without Paths Before adding the path vocabulary, one can automatically test a Thing by sending requests described in its TD in a random order, called a test scenario. Combined with the data structure represented in the TD, it is possible to cover every interaction described in the TD of the Thing under test.

We have developed the test architecture in Fig. 8 to test each of the three interaction patterns introduced in Sect. 2. This architecture allows us to systematically test a Thing by using its TD. We run the corresponding interaction pattern's test method (the vertically aligned boxes) for each interaction in the test scenario as follows:

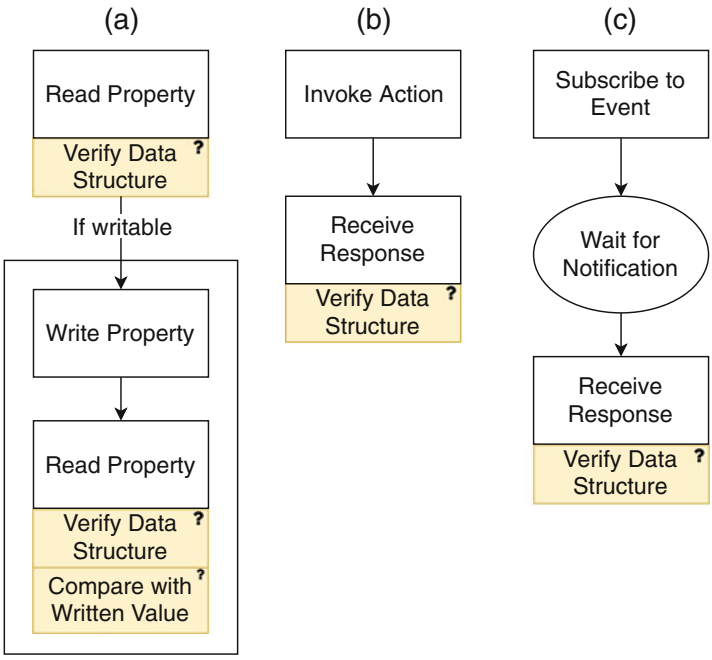

Fig. 8 Architecture of the proposed testing methodology of any interaction of a Thing with a given Thing Description. The yellow boxes (with a ?) symbolize a test that can find either a faulty or correct behavior. The data needed to invoke an action or write to a property is generated using data generation tools

- Property (Fig. 8a): The property value is read and then compared with the structure given in the TD. If the property is writable, a value is generated according to the described data structure and sent to the Thing. The same property is read again to check whether the write request has been successful.
- Action (Fig. 8b): If the action needs input data to execute, the input data is generated and sent to the Thing to invoke the action. Then the response value is compared with the structure given in the TD.
- Event (Fig. 8c): First the event subscription is performed. Once the event is triggered, the value is received and it is compared to the structure given in the TD.

Figure 9 shows an execution trace extract of a test scenario that includes the test of the `rotation` property and the `rotate` action. Here, the Thing under test has interactions that require sequential execution to properly function, but the testing was performed in random order, as the sequence could not be expressed in the TD without paths. This lack of expressiveness makes the test results unreliable. As illustrated in Fig. 9, invoking the `rotate` action and writing to the `rotation` property does not change anything in the system since the `initialize` action has not been invoked before. This is shown as an error because the write operation was not successful, but the real problem is in the order of interactions. This is a problem

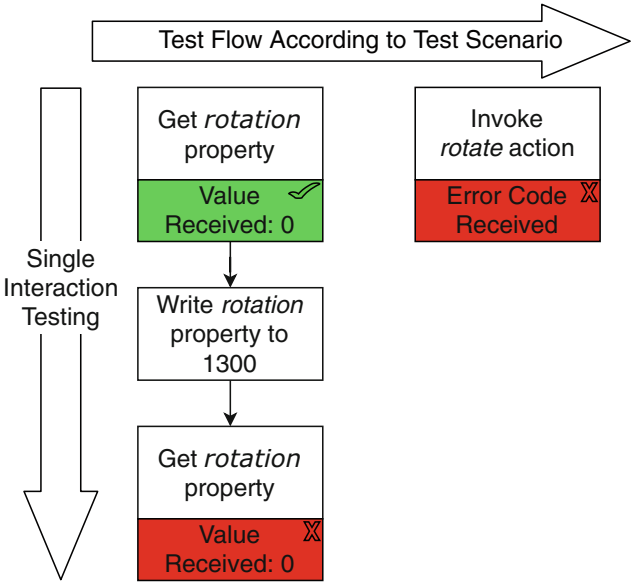

Fig. 9 Illustration of a test path generated from the Thing Description of the industrial ventilator that does not support the path vocabulary. The red boxes (with an X) should symbolize a fault in the ventilator. However, these are the correct responses if the sequential behavior is not respected. The lack of expressiveness in the Thing Description causes this misinterpretation error

found while testing, but the same problem can occur when a consumer Thing (e.g., gateway) is trying to interact with the exposer Thing.

Figure 9 shows two problems originating from the lack of expressiveness regarding sequential behavior:

- The `Rotate` action will *probably* not be used as it is meant to. The only way to do it systematically would be to read a document such as an operation manual and manually write the test scenario.
- Errors in the implementation of the `Rotate` action will never be detected in a systematic way. The `Rotate` action will be used the way it is designed only if the random order of interactions during testing matches the sequential behavior.

Testing with Paths By using the path vocabulary, the randomness of the order of requests can be mitigated. Test scenarios can be generated in a systematic way instead of a random way and thus the actual behavior of the system can be tested. The testing method with vertically ordered boxes of Fig. 8 for testing a single interaction stays the same and only the ordering of the test scenario changes.

By using the path vocabulary, one can automatically generate a test scenario that tests the described sequential behavior. This is illustrated in Fig. 10 where the last test `Get rotation property` is shown to have two outcomes. Normally, there would be only one response. For demonstration purposes, we have illustrated one faulty and one correct response. Compared to the red results (with an X) in Fig. 9, this red result (with an X) in Fig. 10 detects an actual error of the device. In the case of the error outcome, we see a value smaller than the intended one, which can be because of the developer not properly implementing the rotation function of the motor driver. We can conclude that following the correct path allowed us to systematically test the desired behavior of the `write` functionality of the Thing.

There are two advantages of the added expressiveness for testing single Things:

- Test scenarios test the actual behavior of the Thing and show real faults of the Thing under test with respect to its intended behavior.
- More features of the Thing can be tested since following a path describes additional functionality compared to the single interactions alone.

4.2 System Level Testing

In this use case, we will illustrate the testing of the previously introduced temperature controlling system during its development cycle.

As mentioned in Sect. 3.2, it is possible to describe an IoT system in a TD with the path vocabulary. For this specific use case, our gateway/system controller device does not bring any extra functionality and is used only for composing the system. Thus, in its TD, there is no interaction but only paths. It is still the same Thing as described by Listing 3.

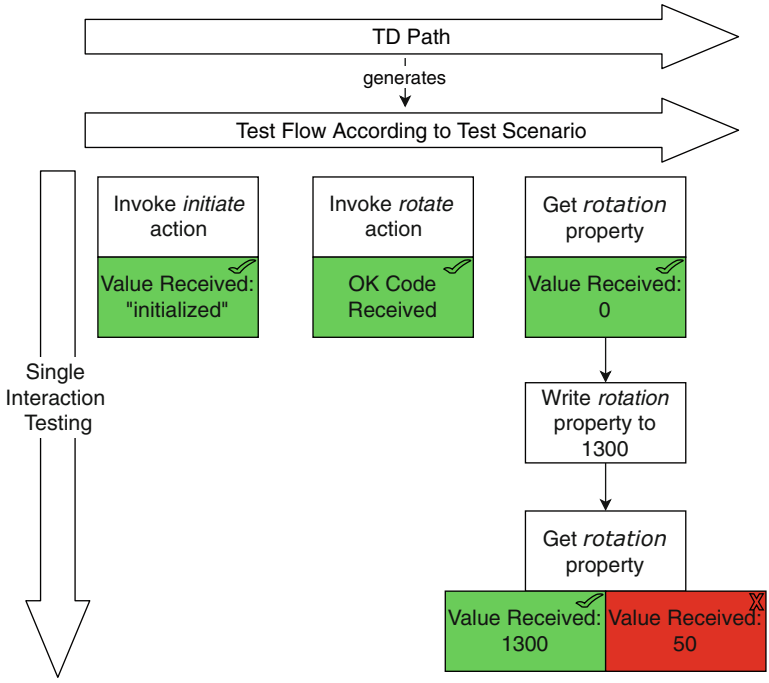

Fig. 10 Illustration of a test path generated from the Thing Description of the industrial ventilator that supports the path vocabulary. Differently from Fig. 9, the correct sequential behavior can be tested and real faults in the system can be identified. The last test case is shown with two possible outcomes, depending on whether the Thing has faults or not, which are both valid test results

All the URIs (lines 10–13) are absolute and they refer to interactions of Things in the system. By following the path named `control` (starting at line 7), the gateway can regulate the temperature of the system. To do so, it gets the temperature value from the temperature sensor, then initializes, and rotates the motor of the ventilator.

Note that paths are composed of interactions as per Definition 3. This means that even though we can use URIs of JSON Pointers that point to paths in a TD, we will decompose a path to its interactions and then include these interactions in a new path. This allows us to keep the same testing logic and not require to modify the definition of a path. The logic stays the same but the system controller will need to fetch the TD that contains the path by using the absolute URI found in the path.

To generalize the testing approach in order to adapt to any TD of a system, and thus to be able to test the complete system, we propose Algorithm 1.

This algorithm allows us to cover the whole system that has arbitrarily many Thing or inter-Thing sequential behaviors. To do so, for every TD of the system (including system controllers) (line 1), it iterates through each path (line 2). In a path, with the listed URIs (line 3), it finds the interaction from every TD using the `findInteraction` function (line 4) and tests the interaction depending on its

Algorithm 1 Algorithm for testing a system of Things based on their TDs that support the path vocabulary

```
1:  for TD ∈ System do
2:      for path ∈ TD do
3:          for uri ∈ path do
4:              interactionUnderTest ← findInteraction(uri)
5:              switch (interactionType)
6:              case property:
7:                  result ← testProperty(interactionUnderTest)
8:              case action:
9:                  result ← testAction(interactionUnderTest)
10:             case event:
11:                 result ← testEvent(interactionUnderTest)
12:             end switch
13:             store  TD.path.uri.result
14:         end for
15:     end for
16: end for
```

type (lines 5–12). In the end, the test results are stored to allow diagnostics of the system (line 13).

The test scenario in Fig. 10 can be generated by using Algorithm 1, even if the `initiate`, `rotate`, and `rotation` interactions are in different TDs. Hence, we can automatically generate test scenarios composed of interactions of different Things and test the system composed of several Things.

In this case study, we have demonstrated that paths allow increasing the meaning of test results as well as the quality of tests and hence contribute to improve the testability of IoT systems.

5 Related Work

Thing Description (TD) is a new standard that has resulted from research on Web of Things and Semantic Web technologies, all trying to address the interoperability problem in IoT. As discussed in [12], Web of Things has found application in industry, resulting in its wide adoption and [13] defined the Thing Description standard by using Semantic Web technologies.

For composing an interoperable IoT system, there have been approaches based on marketplaces for IoT devices, such as in [14, 15]. These marketplaces would offer device descriptions for other devices to search for and consequently to use the devices based on their description. For automatically composing a system, a system controller would look for devices it needs, referred to as recipes in [14], from the marketplace and compose the desired system with the devices it finds. However, there is no description of sequential behavior that can link the capabilities of Things in a sequential order.

Mayer et al. [16] introduce a more generic approach where a goal is set using the RESTdec format, such as controlling the temperature, and the system is composed based on this goal. However, the RESTdec format is not human-readable. In addition, Thuluva et al. [14] and Mayer et al. [16] present top-down approaches and the core technology they are using is not standardized as it is with TDs.

In our approach, however, our first contribution is solving the ambiguity of sequential behavior in TDs in a human-readable format on device level by adding the path vocabulary. As a further contribution, we can use it for composing system behavior in a sequential fashion.

Moreover, the path vocabulary is very similar to formal property specification. Hence, in the future, it might enable the application of formal verification methods.

6 Discussion

The black-box testing approach that we have shown in this paper will be used to automate the testing during the standardization activities of the W3C Thing Description standard. As with any W3C standard, the TD standard document has assertions to describe what devices should do and how their TD should be represented. Chapter 8 “Behavioral Assertions” of [1] provides very clear assertions on how a Thing should behave regarding data types of payload and what are the assumptions on protocols.

The existing WoT implementations provided by the community are used to generate a W3C Implementation Report. Together with assertions in other chapters, this report is a mandatory step for the TD standard to be published. The path vocabulary is not yet part of the TD standard, but our testing methodology is equally valid for TDs without paths.

In the current state, the Web of Things Thing Description Implementation Report as seen in Fig. 11 contains 14 implementations that are tested manually for their conformance to the standard. With the increasing number of implementations, applying an automated testing methodology as presented in this paper will be inevitable.

7 Conclusion

In this paper, we introduced a new vocabulary called paths for the Thing Description standard. Using the path vocabulary, we have described sequential behavior of Things in TDs and made it possible to test such behavior automatically, which was not possible in the current standard. We have shown that the same vocabulary can be used for describing a system composed of individual Things without preprogrammed interfaces. Hence, the methodology to test a single Thing was generalized to test systems composed of individual Things. In a case study, we have

ID	Category	Context	Assertion	Results			
				P	F	N	T
bindings-requirements-scheme	Behavior	Form	Every form in a WoT Thing Description MUST follow the requirements of theProtocol Binding indicated by the URI scheme of its href member.	13	0	3	16
bindings-server-accept	Behavior	(TDConsumer)	Every form in a WoT Thing Description MUST accurately describe requests (including request headers, if present) accepted by the Thing in an interaction.	14	0	3	17
client-data-schema	Behavior	(TDConsumer)	A Thing acting as a Consumer when interacting with another target Thing desciibed in a WoT Thing Description MUST generate data organized according to the data schemas given in the corresponding interactions.	6	0	9	15
client-data-schema-accept-extras	Behavior	(TDConsumer)	A Thing acting as a Consumer when interacting with another Thing MUST accept without error any additional data not described in the data schemas given in the Thing Description of the target Thing.	5	1	9	15
client-data-schema-no-extras	Behavior	(TDConsumer)	A Thing acting asa Consumer when interacting with another Thing MUST NOT generate data not described in the data schemas given in the Thing Description of that Thing.	5	1	9	15
client-uri-template	Behavior	(TD Consumer)	A Thing acting asa Consumer when interacting with another Thing MUST generate URIs according to the URI Templates, base URIs, and form href parameters given in the Thing Description of the target Thing.	3	0	12	15
iana-security-alter	IANA	(TD Consumer)	For this reason, Consumer again SHOULD vet and cache remote contexts before allowing the system to use it.	1	0	6	7
iana-security-execution	IANA	(TD Consumer)	Since WoT Thing Description is intended to be a pure data exchange format for Thing metadata, the serialization SHOULD NOT be passed through a code execution mechanism such as JavaScript's eval() function to be parsed.	5	0	2	7
iana-security-expansion	IANA	(TD Consumer)	Consumers SHOULD treat any TD metadata with due skepticism.	1	0	4	5
iana-security-remote	IANA	(TD Consumer)	While implementations on resource-constrained devices are expected to perform raw JSON processing (as opposed to JSON-LD processing), implementations in general SHOULD statically cache vetted versions of their supported context extensions and not to follow links to remote contexts.	3	0	5	8
server-data-schema	Behavior	(TD Producer)	A WoT Thing Description MUST accurately describe the data returned and accepted by each interaction.	17	1	3	21
server-data-schema-extras	Behavior	(TD Producer)	A Thing MAY return additional data from an interaction even when such data is not described in the data schemas given in its WoT Thing Description.	7	1	13	21
server-uri-template	Behavior	(TD Producer)	URI Templates, base URIs, and href members in a WoT Thing Description MUST accurately describe the WoT Interface of the Thing.	10	0	1	21

Fig. 11 A snapshot of the Web of Things Thing Description Implementation Report as of 06 May 2019. This is a small section of the report that contains the behavioral assertions. The implementation report is constantly updated with new implementations and it can be found on the official GitHub repository of W3C at https://github.com/w3c/wot-thing-description/blob/master/testing/report.html

shown how testing benefits from the enhanced expressiveness in TDs. Thus, this contribution allows us for the first time using Thing Descriptions to systematically compose and test cyber-physical systems.

References

1. Kaebisch, S., Kamiya, T., McCool, M., & Charpenay, V. (2019). Web of Things (WoT) Thing Description. Candidate recommendation, W3C, https://www.w3.org/TR/2019/CR-wot-thing-description-20190516/.
2. Korkan, E., Kaebisch, S., Kovatsch, M., & Steinhorst, S. (2018). Sequential behavioral modeling for scalable iot devices and systems. In *2018 Forum on Specification Design Languages (FDL)* (pp. 5–16). https://doi.org/10.1109/FDL.2018.8524065.
3. Philips Lighting B.V. (2019). Hue API. https://developers.meethue.com/develop/hue-api/.
4. Bryan, P. C., Zyp, K., & Nottingham, M. (2013). JavaScript Object Notation (JSON) Pointer. RFC 6901. https://doi.org/10.17487/RFC6901. https://rfc-editor.org/rfc/rfc6901.txt.
5. Baldoni, R., Contenti, M., Piergiovanni, S. T., & Virgillito, A. (2003). Modeling publish/-subscribe communication systems: Towards a formal approach. In *Proceedings of the Eighth International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*. https://doi.org/10.1109/WORDS.2003.1218097.
6. Koster, M. (2018), Web of Things (WoT) Protocol Binding Templates. Tech. rep., W3C. https://www.w3.org/TR/2018/NOTE-wot-binding-templates-20180405/.
7. Barnett, J., Akolkar, R., Auburn, R., Bodell, M., Burnett, D. C., Carter, J., et al. (2015). State Chart XML (SCXML): State Machine Notation for Control Abstraction. W3C Recommendation, W3C. https://www.w3.org/TR/2015/REC-scxml-20150901/.
8. Bray, T. (2014). The JavaScript Object Notation (JSON) Data Interchange Format. https://rfc-editor.org/rfc/rfc7159.txt. https://doi.org/10.17487/RFC7159.
9. Sporny, M., Lanthaler, M., & Kellogg, G. (2014). JSON-LD 1.0. W3C Recommendation, W3C. http://www.w3.org/TR/2014/REC-json-ld-20140116/.
10. Shelby, Z., Hartke, K., & Bormann, C. (2014). The Constrained Application Protocol (CoAP). https://rfc-editor.org/rfc/rfc7252.txt. https://doi.org/10.17487/RFC7252.
11. The Modbus Organization. (2012). Modbus application protocol specification v1.1b3. http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf.
12. Guinard, D. (2011). http://www.vs.inf.ethz.ch/publ/papers/dguinard-awebof-2011.pdf. PhD thesis, ETH Zurich, Zurich, Switzerland.
13. Charpenay, V., Käbisch, S., & Kosch, H. (2016). Introducing Thing Descriptions and Interactions: An Ontology for the Web of Things. In *Stream Reasoning + Semantic Web technologies for the Internet of Things @Int. Semantic Web Conference*.
14. Thuluva, A., Bröring, A., Medagoda, G., Don, H., Anicic, D., & Seeger, J. (2017). Recipes for IoT Applications. In *Proceedings of the Seventh International Conference on the Internet of Things*. New York: ACM. https://doi.org/10.1145/3131542.3131553.
15. Bröring, A., Schmid, S., Schindhelm, C. K., Khelil, A., Käbisch, S., Kramer, D., et al. (2017). Enabling IoT Ecosystems through Platform Interoperability. *IEEE Software, 34*(1), https://doi.org/10.1109/MS.2017.2.
16. Mayer, S., Verborgh, R., Kovatsch, M., & Mattern, F. (2016). Smart configuration of smart environments. *IEEE Transactions on Automation Science and Engineering*. https://doi.org/10.1109/TASE.2016.2533321.