Sequential Behavioral Modeling for Scalable IoT Devices and Systems

Ege Korkan[1], Sebastian Kaebisch[2], Matthias Kovatsch[2], Sebastian Steinhorst[1]
[1] *Technical University of Munich,* Germany, Email: {ege.korkan, sebastian.steinhorst}@tum.de
[2] *Siemens AG,* Germany, Email: {sebastian.kaebisch, matthias.kovatsch}@siemens.com

***Abstract*—The Internet of Things (IoT) enables connectivity between devices, thereby allowing them to interact with each other. A recurring problem is the emergence of siloed IoT platforms due to proprietary standards. Recently, the World Wide Web Consortium (W3C) proposed a human-readable and machine-understandable format called Thing Description (TD). It allows to uniformly describe device and service interfaces of different IoT standards with syntactic and semantic information, and hence enables semantic interoperability. However, describing sequential behavior of devices, which is essential for many cyber-physical systems, is not covered. In this paper, we propose a systematic way to describe such sequential behavior as an extension within TDs, thereby increasing their semantic expressiveness through possible, valid state transitions. This enables safe and desired operation of devices as well as scalability by modeling systems as sequential compositions of Things. We show in a case study that previously unmodelable behavior can now be expressed and the overall manual intervention requirements of state-of-the-art implementations can be significantly reduced.**

***Index Terms*—Internet of Things, Thing Description, CPS, Model-driven development, System Testing**

I. 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 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 which 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.

With the support of the Technische Universität München – Institute for Advanced Study, funded by the German Excellence Initiative and the European Union Seventh Framework Programme under grant agreement n° 291763.

 DOI: 10.1109/FDL.2018.8524065

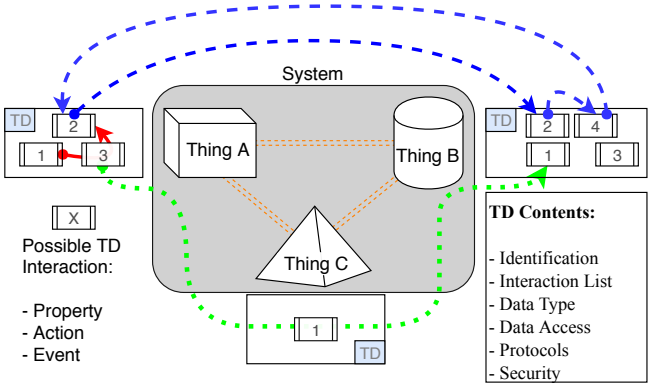

Figure 1: An abstracted view of an IoT System with 3 IoT Devices each with an associated Thing Description (TD). The arrows demonstrate composition of greater functionality than the devices themselves, necessitating sequential behavior between devices.

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[2], 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 the paper, an exposer Thing accepts requests provided in its TD, whereas the consumer Thing reads a TD and interacts with the exposer Thing.

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 Section II in more detail.

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

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 Section II, this paper has the following contributions:

- We propose an additional vocabulary[3] to describe sequential behavior in a Thing Description, called **path** in Section III-A. 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 Section III-B.
- 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 Section IV.

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

Related work is discussed in Section V and Section VI concludes.

II. Thing Description

The Thing Description (TD) approach has been introduced in September 2017 (First public draft) by the Web of Things (WoT) 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 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 Section III, 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 (Def. II.1) and publish-subscribe (Def. II.2).

Definition II.1. (Request-Response)
For a request $p \in \text{client}$ and a $q \in \text{server}$, the pair is defined as follows:

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

Definition II.2. (Publish-Subscribe)
Notifying an event only in matching subscription intervals is defined by [2] 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 which 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. Also, in case of an overheating of the motor, it can notify the consuming Things who are subscribed to this notification.

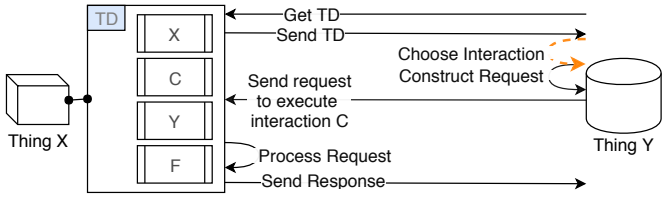

Figure 2: Request-Response sequence abstraction that can be used for interacting with a Thing. The orange (dashed) arrow demonstrates the missing part of the TDs which is the problem addressed in the paper.

Other than interactions, the TD provides identification information. In the order of appearance in Listing 1, the `name` provides a human readable reference (identification) for this Thing, whereas `id` provides a unique identification that stays unchanged through different networks or IP addresses. Similarly the `base` (line 3) describes the protocol and the Uniform Resource Identifier (URI) needed to communicate with this Thing. Correspondingly, an interaction is always described with its URI (access protocol and address), allowing the interaction to be defined even if this URI is outside the TD.

By using the default protocol bindings described in [3], 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 (line 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 that tells the consumer to execute interactions in a specific order. Without the contribution of this 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.

III. 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 will start by introducing the vocabulary for single devices and then extend it for systems composed from devices.

```
1 {
2   "name": "MyVentilator",
3   "id": "urn:wot:com:servient:ventilator",
4   "base":"coaps://vent.example.com:5683"
5   "properties":{
6     "rotation":{
7       "type": "integer",
8       "writable": true,
9       "forms":[{"href": "/rotation"}]
10    }
11  },
12  "actions":{
13    "rotate":{
14      "forms":[{"href": "/rotate"}]
15    },
16    "initialize":{
17      "output":{"type": "string"},
18      "forms":[{"href": "/init"}]
19    }
20  },
21  "events":{
22    "overheating":{
23      "type":"string",
24      "forms":[{"href": "/oh"}]
25    }
26  }
27 }
```

Listing 1: Simple Thing Description of a ventilator that exposes the rotation speed, motor initialization and rotating actions and an overheat alarm.

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 which 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 whole 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 whole 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.

A. 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 the 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[4], but requires strict sequential behavior in order to operate safely. A sequence of

[4]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

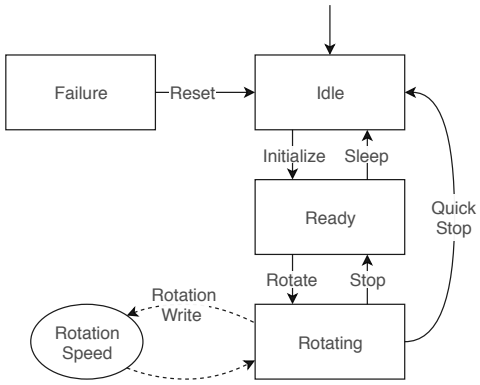

Figure 3: State machine representation of a legacy motor driver. In order to enable setting the rotation speed to the desired value, Initialize, Rotate interactions have to be executed in this order.

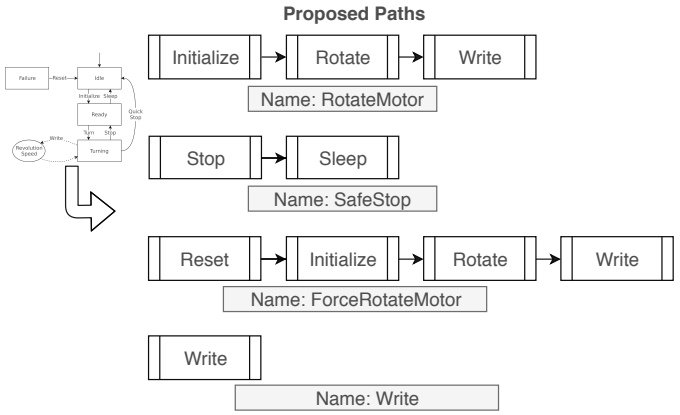

Figure 4: Illustration of Thing Description paths based on the state machine of a legacy motor driver for an industrial fan. The paths are composed of interactions that execute state transitions. Note that the path just contains a single interaction, which is still a valid representation.

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 to 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. E.g., 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. Furthermore, if the consumer Thing has access to this specific state machine in a machine readable format (such as SCXML [4]), 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 describe 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 III.1. (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:

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

Definition III.2. (Name)

The name of the path is used within the TD to reference the JSON [5] object that contains the path information. Within the TD, the name allows the path to be referenced in the following fashion:

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

with $\mathrm{derivePath}$ being a function that finds the path *t* by parsing the TD.

Definition III.3. (@type)

The `@type` optionally allows to annotate semantics with the path. It uses the JSON-LD [6] 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[5] and used for linking the data.

Definition III.4. (Paths)

The set of paths offered by the Thing is denoted by Π and defined as follows:

$$\Pi = \bigcup \pi_k \ \mid \pi_k \in \mathrm{TD} \tag{5}$$

These formal definitions translate to a path description in a TD[6] as shown in Listing 2. It is 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` which brings the motor to the initial state by executing `stop` and `sleep`, in these respective orders.

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 the paper, we have used *modern* protocols such as CoAP [7] in the TD listings. However, the advantage of TDs is the capability to also describe older protocols such as Modbus [8], 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.

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

[6]Path description can be also made using the id URN, but this is left out of the scope of the paper.

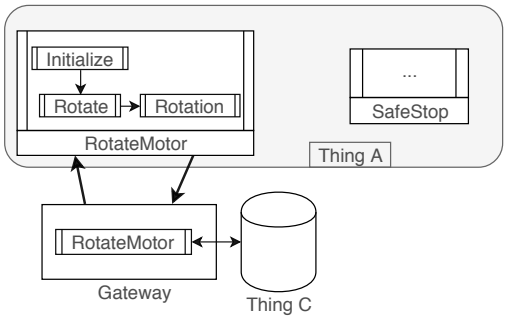

Figure 5: Using a gateway brings IoT connectivity to a legacy motor driver (Thing A). The gateway can execute a path offered by this device and offer a simple Thing Description action to be executed by Things that do not have physical access to Thing A, such as Thing C.

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`.

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

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

B. 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.

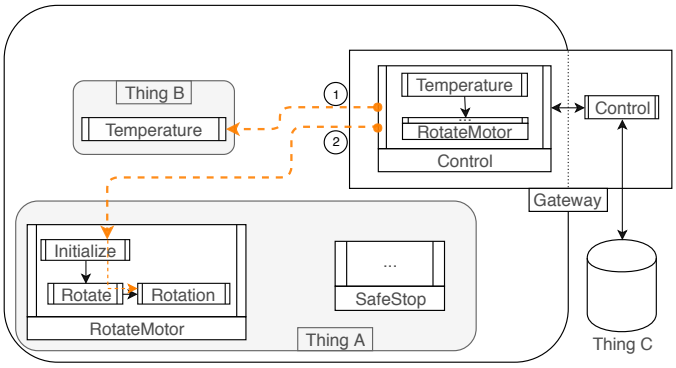

Figure 6: A gateway can compose a system through the use of the path vocabulary. Here, the system is a temperature control system with a temperature sensor and an industrial ventilator. Things, such as Thing C, that do not have physical access to the system components can execute simple Thing Description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

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 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. Fig. 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.

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 for a single Thing contains interaction URIs, 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 loose their uniqueness outside the TD.[7]

IV. 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.

[7]A URI in a TD such as `/initialize` can be combined with the base 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/initialize`.

```
{
  "id": "urn:wot:com:example:system:tempCont",
  "name": "SystemController",
  "@context": ["https://w3c...jsonld",
               {"iot": "http://iot.schema.org/"}],
  "paths":{
    "control":{
      "@type":"iot:temperatureControl",
      "path":[
        "http://fdlSensor.com:5683/temperature",
        "coaps://vent.example.com:5683/initialize",
        "coaps://vent.example.com:5683/rotate",
        "coaps://vent.example.com:5683/rotation"
      ]
    }
  }
}
```

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.

It is even more difficult to implement everything correctly when designing and implementing a system because of the interlinked behavior of devices that compose the system. During both development processes for testing 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 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.

A. 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.

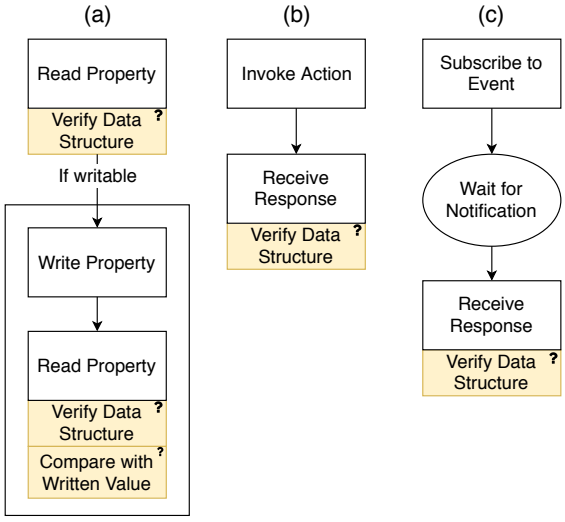

Figure 7: Architecture of the proposed testing methodology of any interaction of a Thing with a given Thing Description. The yellow boxes (with a ?) symbolize a test that can find either a faulty or correct behavior. The data needed to invoke an action or write to a property is generated using data generation tools.

We have developed the test architecture in Fig. 7 to test each of the three interaction patterns introduced in Section II. 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:

- Property (Fig. 7(a)): 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. 7(b)): 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. 7(c)): 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.

Fig. 8 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. 8, 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 found while testing, but the same problem can occur when a consumer Thing (e.g. gateway) is trying to interact with the exposer Thing.

Fig. 8 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

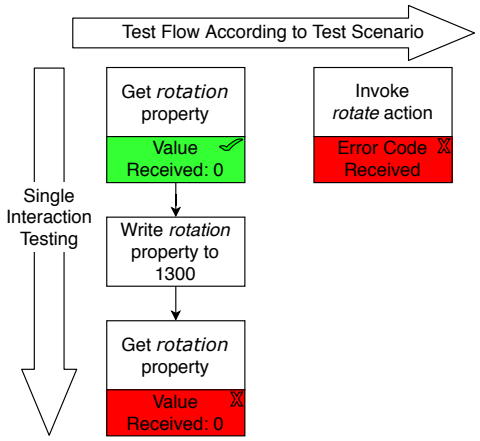

Figure 8: Illustration of a test path generated from the Thing Description of the industrial ventilator that does not support the path vocabulary. The red boxes (with an X) should symbolize a fault in the ventilator. However, these are the correct responses if the sequential behavior is not respected. The lack of expressiveness in the Thing Description causes this misinterpretation error.

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. 7 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. 9 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. 8, this red result (with an X) in Fig. 9 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.

B. System Level Testing

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

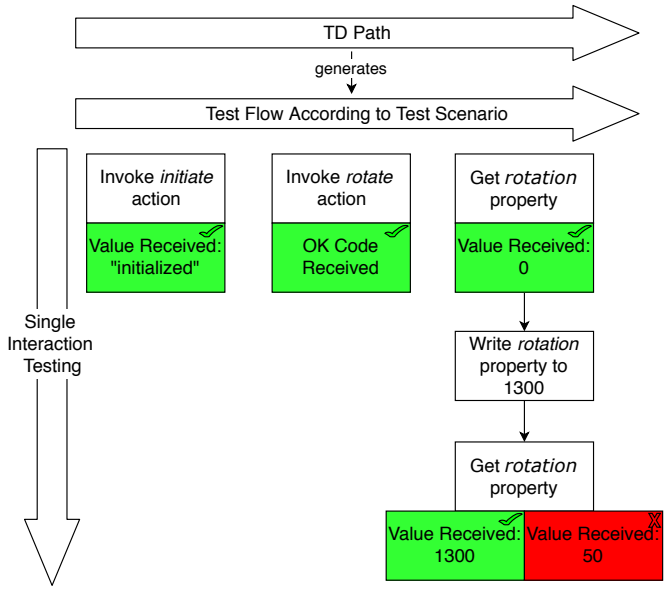

Figure 9: Illustration of a test path generated from the Thing Description of the industrial ventilator that supports the path vocabulary. Differently from Figure 8, the correct sequential behavior can be tested and real faults in the system can be identified. The last test case is shown with two possible outcomes, depending on whether the Thing has faults or not, which are both valid test results.

As mentioned in Section III-B, 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.

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 URIs identify resources. Thus, a path does not have a URI since it is not a resource on its own, but its contents are interactions, which are well-defined URIs. For this reason, when including a path in an higher level device, we will need to parse the contents of the path, remove its name and create a new list of paths. This can be seen in the TD of the system controller in the `control` path where the `RotateMotor` path has been decomposed into the URIs of its interactions (lines 11-13).

The testing logic with TDs containing paths represented for single Things can be applied here with a small modification. The path URIs refer to the interaction, but the type of an interaction (e.g. action) can only be found in the TD that contains this interaction. Since the system controller has access to the TDs of every device in the system (e.g. stored in its memory), the interaction with a specific URI can be found by searching through these TDs. Hence, the test path will still be generated through the paths of the system controller, but the interaction with the Things of the system will be performed through their TDs. As the URIs are unique, there can be no mismatch of interactions.

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

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
```

This algorithm allows us to cover the whole system that has arbitrary 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 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. 9 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 to increase the meaning of test results as well as the quality of tests and hence contribute to improve the testability of IoT systems.

V. Related Work

Thing Description (TD) is a new standard which has resulted from research on Web of Things and Semantic Web technologies, all trying to address the interoperability problem in IoT. As discussed in [9], Web of Things has found application in industry, resulting in its wide adoption and [10] 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 [11], [12]. 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 [11], 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.

[13] introduces 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. Also, [11] and [13] 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.

VI. 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 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] T. Kamiya and S. Käbisch, "Web of Things (WoT) Thing Description", W3C, W3C Working Draft, 2018, https://www.w3.org/TR/2018/WD-wot-thing-description-20180405/.

[2] R. Baldoni, M. Contenti, S. T. Piergiovanni, and A. Virgillito, "Modeling publish/subscribe communication systems: towards a formal approach", in *Proc. of the Eighth Int. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, 2003.

[3] M. Koster, "Web of Things (WoT) Protocol Binding Templates", W3C, Tech. Rep., 2018, https://www.w3.org/TR/2018/NOTE-wot-binding-templates-20180405/.

[4] J. Barnett, R. Akolkar, R. Auburn, M. Bodell, D. C. Burnett, J. Carter, S. McGlashan, T. Lager, M. Helbing, R. Hosn, T. Raman, K.Reifenrath, N. Rosenthal, and J. Roxendal, "State Chart XML (SCXML): State Machine Notation for Control Abstraction", W3C, W3C Recommendation, 2015, https://www.w3.org/TR/2015/REC-scxml-20150901/.

[5] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format", IETF RFC 7159, 2014.

[6] M. Sporny, M. Lanthaler, and G.Kellogg, "JSON-LD 1.0", W3C, W3C Recommendation, 2014, http://www.w3.org/TR/2014/REC-json-ld-20140116/.

[7] Z. Shelby, K. Hartke, and C. Bormann, "The Constrained Application Protocol (CoAP)", IETF RFC 7252, 2014.

[8] The Modbus Organization, "Modbus Application Protocol Specification V1.1b3", http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf, 2012.

[9] D. Guinard, "A Web of Things Application Architecture – Integrating the Real-World into the Web", Ph.D. thesis, ETH Zurich, Zurich, Switzerland, 2011.

[10] V. Charpenay, S. Käbisch, and H. Kosch, "Introducing Thing Descriptions and Interactions: An Ontology for the Web of Things", in *Stream Reasoning + SemanticWeb technologies for the Internet of Things @Int. Semantic Web Conf.*, 2016.

[11] A. Thuluva, A. Bröring, G. Medagoda, H. Don, D. Anicic, and J. Seeger, "Recipes for IoT Applications", in *Proc. of the Seventh Int. Conf. on the Internet of Things*. ACM, 2017.

[12] A. Bröring, S. Schmid, C. K. Schindhelm, A. Khelil, S. Käbisch, D. Kramer, D. L. Phuoc, J. Mitic, D. Anicic, and E. Teniente, "Enabling IoT Ecosystems through Platform Interoperability", *IEEE Software*, vol. 34, no. 1, 2017.

[13] S. Mayer, R. Verborgh, M. Kovatsch, and F. Mattern, "Smart Configuration of Smart Environments", *IEEE Transactions on Automation Science and Engineering*, 2016.