Streamlining IoT System Development with Open Standards

Ege Korkan, Sebastian Kaebisch, Sebastian Steinhorst

Abstract: The Internet of Things (IoT) is bringing Internet connectivity to a wide range of devices which results in an increasing number of products for smart home, industry 4.0 and/or smart cities. Even though IoT has the ambition to reach an increasing amount of devices and be scalable across different domains, lack of interoperability inhibits this scope to be attained. Recent standardization efforts by the World Wide Web Consortium (W3C) are addressing the interoperability problem by the means of Thing Description (TD) that allows humans and machines to understand the capabilities and communication interfaces of IoT devices. In this paper, we show a more systematic and streamlined development of IoT devices and systems that relies on the TD standard. We introduce three different complementary methods that can be applied independently in the different stages of the development, or as a framework to streamline the development of IoT devices and systems. As a result of using the TD standard, interoperability between IoT devices of various stakeholders is ensured from early stages and the time to market is reduced.

ACM CCS: Information systems → Semantic web description languages; Software and its engineering → Software development techniques; Information systems → RESTful web services; Computing methodologies → Simulation types and techniques

Keywords: Internet of Things, Web of Things, Thing Description, Simulation, Automated Testing, Software Development

1 Introduction

The Internet of Things (IoT) forms a system of Internet-connected devices such as sensors and actuators which communicate over the Internet Protocol (IP). These devices, which are also referred to as Things, allow humans or other devices to control and monitor them using technologies compatible with the conventional Internet-connected devices such as servers, personal computers and smartphones. In recent years, IoT devices from various manufacturers for business and private customers, as well as different business areas, have been introduced to the market, allowing a wide choice of solutions [1, 2].

This diversity also makes the development of IoT systems difficult for developers and integrators who have to work with these different devices. It requires them to understand different APIs and communication protocols or be locked to a platform or middleware. These are commonly called silos and in both cases, composing a system of IoT devices from different silos or manufacturers requires significant integration work. This is referred to as the interoperability problem, which inhibits the IoT from reaching its full potential.

The standards that are being developed by the World Wide Web Consortium (W3C) address this interoperability problem, while also enabling the use of widespread Web technologies on IoT devices, forming the Web of Things (WoT). How the WoT technologies can be applied to different use cases are introduced in the W3C WoT Architecture [3] specification, whereas the Thing Description (TD) standard in [4] describes network-facing interfaces of Things, no matter the manufacturer, protocol, data format or security configuration.

The WoT standards help with the interoperability aspect of IoT devices from the point of view of system integrators who need to build systems of IoT devices. On the other hand, the network-facing interface part of the software development of individual devices is not necessarily tied to a framework as an architectural principle in order to not constrain developers or companies to a certain solution. While this makes it easier to integrate already existing implementations to the standardized WoT, it also creates fragmentation, where different parts

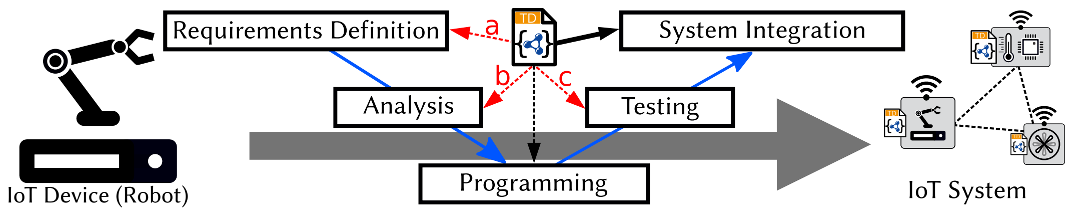


Figure 1: Our contributions allow a Thing Description (TD) to be used during the development cycle of a Thing even though the intended used of TD is the integration of a Thing to an IoT system. Here, as a part of the V model of a system development, the TD can be used independently to define the requirements (a) and analyze the requirements in a simulation environment (b). The device can be programmed using any language or method and the same TD can be used for testing (c).

of the development are not supported by state-of-the-art tools and results in a non-streamlined development process. This still leaves room to reduce the development efforts.

Problem Definition: Existing solutions to ease the development effort for WoT devices are based on methods that are meant for usual Web services and are generally protocol specific. For example, it is possible to simulate and mock HTTP servers with tools such as *Postman*[1] and benchmark them via *wrk*[2] or to assess the vulnerabilities of MQTT implementations via *MQTTSA* [5]. However, given the flexibility of TDs to describe any protocol, these solutions introduce fragmentation to the development cycle. Furthermore, since TDs already describe capabilities of the Thing, these solutions require manual extraction of information such as the Uniform Resource Identifiers (URIs), protocols and methods by the developer, which could be easily avoided since TDs are machine readable.

Contributions: A streamlined development process that does not rely on a complete framework or a specific protocol is thus needed, where requirement definition, development, and testing stages of the development are supported independently and abstracted from the protocols. This can be achieved by connecting different stages of the development by the use of TD documents, which can be used for different purposes based on the development stage. Thus, to streamline the development of network facing interfaces of WoT systems, we have the following contributions as shown in Figure 1:

(a) A method to give requirements as a TD, validate it and to choose an implementation based on the features required, introduced in Section 3.1.
(b) A no-code simulation environment to simulate virtual Things that can be used to confirm the requirements set before as a TD, introduced in Section 3.2.
(c) An automated black-box testing method for existing Things using their TDs in a protocol agnostic fashion and enable a Continuous Integration workflow, introduced in Section 3.3.

Consequently, we present the application of our contributions as case studies for the development cycles of 2 different IoT devices in Section 5. Section 6 discusses the related work and Section 7 concludes.

2 W3C Web of Things

The Web of Things (WoT) was introduced as a concept in 2009 [6] as a unifying application layer for the IoT. This has led to its standardization by the WoT Working Group of World Wide Web Consortium (W3C), starting in 2016 [1] and resulting in the first versions of the standards in April 2020, namely the WoT Thing Description (TD) [4] and the WoT Architecture [3]. These specifications allow to describe existing IoT systems as well as establish design guidelines for new systems.

The TD specification, which is the core enabling technology of our contributions, allows to describe the network facing interfaces of IoT devices. These IoT devices are referred to as Things and other devices that interact with them are called Consumers. TD abstracts the capabilities of individual Things into 3 categories, namely Interaction Affordances:

- Property: Concrete data of the Thing that can be readable, writable, or observable. These includes configuration parameters, sensor values, etc.
- Action: Physical interactions with the environment that are performed by the Thing, such as moving a robot arm, brewing a coffee or closing the blinds.
- Event: Asynchronous data pushed by the Thing to its subscribers, such as a button press, overheat alert or high-speed position updates.

These definitions can be concretely demonstrated in the TD of Listing 1.1, which describes an IoT-enabled robot arm. The position of its base can be read and observed (lines 10-18), its base can be rotated with an Action Affordance (lines 20-28) and the press of an emergency button alerts the subscribers (lines 30-36).

In detail, each Interaction Affordance can contain terms of the Data Schema vocabulary, a subset of JSON Schema [7]. In the Listing 1.1, it can be seen in lines 14-15 and 26-29 that the position uses an integer between -90 and 90. More complex data structures such as nested objects (maps) or arrays (lists) are also possible but are left out of this paper due to space constraints.

Additionally, each Interaction Affordance contains one or multiple forms which contain the details such as the protocol, content type of the data and the operations

[1] `https://www.postman.com`
[2] `https://github.com/wg/wrk`

```
 1 {
 2   ...
 3   "title":"RobotArm",
 4   "base":"http://192.168.0.2:8080/",
 5   "securityDefinitions": {
 6     "basic_sc": {"scheme": "basic",
 7     "in":"header"}},
 8   "security": ["basic_sc"],
 9   "properties":{
10     "basePosition":{
11       "type":"integer",
12       "minimum":-90, "maximum":90
13       "readOnly":true,"observable":true,
14       "forms":[{
15         "href":"properties/basePosition",
16         "op":["readproperty",
17               "observeproperty"]}]
18     }},
19   "actions":{
20     "rotateBaseTo":{
21       "input":{
22         "type":"integer",
23         "minimum":-90, "maximum":90
24       },
25       "forms":[{
26         "href":"actions/rotate",
27         "htv:methodName":"POST"}]
28     }},
29   "events":{
30     "emergencyButton":{
31       "forms":[{
32         "href":"mqtt://broker:1883/button",
33         "mqv:controlPacketValue":"SUBSCRIBE",
34         "op":"subscribeevent"
35       }]
36   }}}
```
Listing 1.1: Example of a simple Thing Description. It describes a robot arm that exposes its position as an observable property, command to rotate its base as an action and the press of the emergency button as an event. The property and action are exposed with the HTTP protocol, whereas the event is exposed via MQTT protocol, showing that different protocols can be used in a Thing Description.

that are possible on the Interaction Affordance. For example, a Consumer can read and observe the property `basePosition` of Listing 1.1 by using the protocol and address defined in the `base` and the relative URI at `href` (line 19).

As with any W3C specification, the TD specification contains assertions, which are phrases that give prescriptions for TD instances. For example, contents of the `forms` term of a property has the following assertion:

> When a Form instance is within a PropertyAffordance instance, the value assigned to op MUST be one of readproperty, writeproperty, observeproperty, unobserveproperty or an Array containing a combination of these terms. [4]

As a requirement to become a standard, these assertions must be verified for their implementability. Thus, different WoT implementations have been submitted by different members of the W3C WoT Working Group and the results of implementations regarding what assertions they fulfill are grouped in the WoT TD Implementation Report [8].

In addition to the TD and Architecture specifications, two notes[3] from the working group are also of importance for this paper:

- **WoT Scripting API** [9] specifies a common programming interface for Thing implementations as well as Consumer applications. While TD can be used to describe existing Things, Scripting API prescribes an interface that can be implemented by different programming languages. As of April 2020, three implementations of the WoT Scripting API exist.
- **WoT Binding Templates** [10] is an informative note and explains how different application layer protocols and media types can be described in a TD. Together with the TD specification, it brings the protocol-independent modeling of IoT devices.

3 WoT Device Development Cycle

In this section, we detail the contributions to the WoT device development cycle we have mentioned in Section 1. We take the V-model shown in Figure 1 as a reference and concentrate on the parts marked by the red-dashed arrows. First, in Section 3.1 we introduce how to use TDs to define the requirements (arrow a), validate them and to choose a corresponding implementation. Secondly, we show in Section 3.2 how the defined requirements can be used to simulate the desired Thing implementation, which can be used to analyze the requirements in a system with other Things or Consumers (arrow b). Finally, we explain in Section 3.3 that the same TD can be used as an input for a black-box testing approach that requires no input from the developer or tester (arrow c).

3.1 Defining the Requirements and Validation

In addition to describing a concrete Thing, a TD can be used to specify the expected network facing capabilities of the device. It is thus possible to specify the required protocols, number, and types of interaction affordances in advance. Since the structure of a TD document is standardized, it makes it possible to validate it, thus validating the requirements. In this chapter, we present a guide for turning text-like requirements into a TD and our validation method that is now used by the W3C WoT Working Group.

Defining the Requirements: For the development of WoT devices, two types of requirements are needed:

1. **Application logic requirements:** Consists of defining how the device should be used as part of a system's application logic. For example, this Thing should deliver a temperature value between 20 and 50 degree Celsius and alert the Consumer when a button is pressed. These requirements are purely driven by the application logic and do not rely on lower level protocols.

[3] A note is a specification-like document that undergoes a less strict review process and is generally used for complementary specifications

2. **Protocol and security requirements:** Consists of the protocols that Thing will use to deliver data to perform the application logic as well as the payload types (JSON, JPEG etc.). In addition, there are security requirements that need to be specified and a security mechanism needs to be chosen. These requirements are independent of the application logic and are influenced by the Consumer's capabilities of supporting different protocols or security mechanisms. For example, if the Consumer application is a browser application where the supported security mechanism is Bearer Tokens, the Thing can only provide Interaction Affordances with HTTP and must use Bearer Tokens as a security mechanism.

One can also see how these requirements can be separated in a TD. In fact, for the `basePosition` property, the lines 11-13 of Listing 1.1 correspond to application logic requirements, whereas the lines 4-8 and 14-17 correspond to the security requirements and protocol, respectively. Thus, in Requirements Definition stage of Figure 1, the application logic requirements should be fixed and then the appropriate solutions for the protocol and security mechanism should be defined.

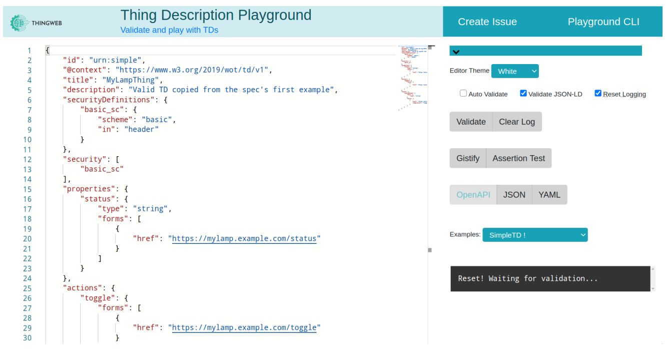
Validation: The requirements that are presented in the form of a TD can be validated for conformance. We present a multi-step method to validate both types of requirements introduced above. Since the default serialization format of TDs is in JSON-LD [11], we propose a JSON Schema [7] based validation, where the TD vocabulary is modeled with a JSON Schema document.

We use the TD Information Model defined in [4] for validation of the TDs. We follow this hierarchical model to define the types and structures of the different vocabulary terms, which are used to validate a TD instance.

3.1.1 Choosing an Implementation

A TD is designed to hide the implementation details of a Thing and only offer the capabilities that are relevant for the Consumer. However, during the concrete development of a Thing based on a TD as a requirement definition, the implementation details need to be specified. Due to the inherent flexibility of WoT technologies, not every WoT software library supports the features that can be required from a given TD. For example, one can support delivery of events and another one could offer only properties that are read on demand from the Consumer. Thus, when analyzing the requirements, a corresponding WoT software library has to be chosen, where a library is a code base that can be used to develop the source code of a concrete Thing or Consumer, which we refer to as an implementation instance.

The W3C WoT TD Implementation Report [8] lists the existing WoT implementations with a detailed set of assertions of the TD specification that they implement. An assertion in this context is a clause that needs to be satisfied by a TD instance of a WoT implementation.



```
{
    "id": "urn:simple",
    "@context": "https://www.w3.org/2019/wot/td/v1",
    "title": "MyLampThing",
    "description": "Valid TD copied from the spec's first example",
    "securityDefinitions": {
        "basic_sc": {
            "scheme": "basic",
            "in": "header"
        }
    },
    "security": [
        "basic_sc"
    ],
    "properties": {
        "status": {
            "type": "string",
            "forms": [
                {
                    "href": "https://mylamp.example.com/status"
                }
            ]
        }
    },
    "actions": {
        "toggle": {
            "forms": [
                {
                    "href": "https://mylamp.example.com/toggle"
                }
```

Editor Theme: White

☐ Auto Validate ☑ Validate JSON-LD ☑ Reset Logging

Examples: SimpleTD !

```
Reset! Waiting for validation...
```

Figure 2: Thing Description Playground is the reference validator used by the standardization activities of W3C. On the left hand side, the editor supports live validation while more specific log messages can be seen in the terminal-like fields below the editor. It also allows to identify the corresponding assertions of a given Thing Description.

For example, *const: Provides a constant value. MAY be included. Type: any type.* specifies the meaning and use of the *const* keyword. Additionally, an implementation refers to a software (library) that can be used to develop the source code of a concrete Thing or Consumer, which we refer to as an implementation instance.

As the TD specification is designed to be flexible in order to specify different types of devices, different WoT implementations concentrate on a use-case and not necessarily implement all of the assertions. Following on the previous example, it is possible that an implementation cannot describe the constant values provided by a Thing. However, W3C, as a standardization organization, ensures that all the assertions of the TD standard is implemented by at least two implementations, guaranteeing that any TD given as a requirement, can be implemented. Thus, we can use this guarantee to find a corresponding implementation for a Thing instance.

3.1.2 Implementation: TD Playground

For both the validation of the requirements and for choosing an implementation, we have developed the TD Playground, a set of software packages that uses TD files for different purposes. It is open sourced at our GitHub repository[4] where each package can be used as a module on its own. It is also hosted online[5], whose graphical overview is seen in Figure 2.

Validation and Editing: The user is able to edit a TD in the left hand side of Figure 2 and see errors and warnings of the TD validation process automatically with a live-validation approach on the right hand side. In addition to validating according to the TD Information Model, we verify that the TD is a valid JSON-LD instance and also provide additional checks to obtain a more expressive TD.

Assertion Testing: Similar to the TD validation, we define atomic JSON Schemas for each assertion defined

[4] `https://github.com/thingweb/thingweb-playground`
[5] `http://plugfest.thingweb.io/playground/`

in the TD specification. Compared to the validation of a TD with respect to the TD Information Model, these JSON Schemas contain less validation points and thus validate only the targeted assertion. It is also integrated into the TD Playground web interface shown in Figure 2.

Adoption by the W3C WoT Standardization: Our implementation is adopted by the W3C WoT Working Group and our JSON Schema for validating a TD instance is incorporated into the TD specification[6]. It can be also used as a dependency in another software project to validate TDs. The assertion testing functionality is also used by the TD standard to analyze the existing WoT implementations and produce the automated validation results shown in the table of Section 8.1 of [8]. Finally, these results can be cross-checked with required assertions generated from the TD given as a requirement to find a corresponding implementation.

3.2 Simulation of the Things

Once the requirements are set via a TD, this TD can be used to simulate a Thing. A simulation is needed in this stage of the development in order to verify the requirements set in the beginning of the V-model. Since the requirements are shown in the form of a TD, which will be also used by Consumers, a simulation that satisfies the behavior described by the TD would simulate the requirements and allow to confirm them.

In order to verify the requirements earlier in the development process, we propose a method to generate a simulation instance based on a TD, which is illustrated in Figure 3. Our method requires no configuration or programming effort by a developer and can be used by the same person who creates the requirements in form of a TD.

We use the TD that is set as a requirement to initialize the required protocol stack as shown in the step c. This protocol stack handles the requests from the Consumers and emits the events that are all specified as Interaction Affordances. The payloads of the Interaction Affordances are also automatically generated according to their Data Schema, as illustrated by step b. Without any out-of-band information, it is not possible to automatically generate the internal logic shown in step a, since interdependencies between affordances or internally triggered state changes are not deducible from a TD.

We can take the example of a temperature control system of a room involving a Thing to increase or decrease the temperature and a temperature sensor that is yet to be developed. We can define the requirements of the temperature sensor with a TD, such as its different Interaction Affordances and their corresponding Data Schema, the required security schemes and the protocols that it should use. All of these requirements can be then simulated and the system-level program can be

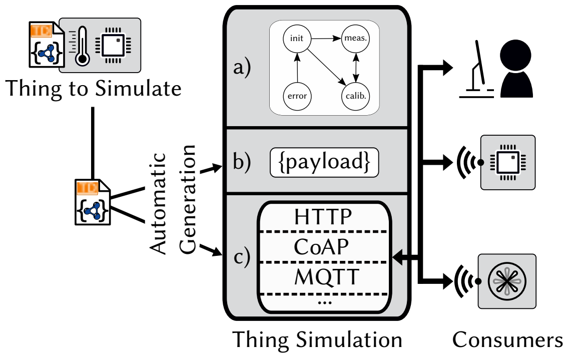


Figure 3: Our simulation method allows to generate the protocol stack (c) of a Thing and to mimic the responses and event data (b) of the Thing according to its Data Schema definitions in its Thing Description. The internal logic of the Thing (a) can be created by a developer since a TD alone cannot be used to generate such logic.

written to verify whether the initial requirements were correctly set in the TD. In case the requirements were not set correctly, the TD can be changed and the Thing can be simulated again with no other change required.

3.2.1 Implementation: Shadow Thing

Our simulation method is implemented with the Node.js framework under the name Shadow Thing and it is open sourced in our repository[7] and npm library[8]. The lower level protocol stacks are implemented using the `node-wot` library[9] from the Eclipse Foundation's Thingweb project [12]. Furthermore, it is possible to override the automatic payload generation with custom models that mimic the internal logic of the Thing. Such custom models can be reused in the Programming step of the Figure 1 by completing them with the sensor or actuator drivers.

3.3 Black-box Testing of Things

After or even during the development of a Thing, its TD can be used to test the Thing automatically. Since the TD hides the implementation details, a testing approach based on a standard TD can only be black-box testing. Different from other black-box testing approaches, we propose a method that requires no effort from the developer, such as configuring the testing parameters. Furthermore, compared to other testing approaches for IoT devices and protocols as explained in [13], we start from the application's point of view and allow testing of any protocol described in the TD of the Thing under Test.

In Figure 4, we detail how our method functions, starting from a TD to involve different Interaction Affordances. We focus on separating the TD into its different interaction affordances as well as meta-interactions, which involve executing multiple interactions and are described on the root of the TD document. After testing each

[6] `https://w3c.github.io/wot-thing-description/#json-schema-for-validation`

[7] `https://github.com/tum-esi/shadow-thing`

[8] `https://www.npmjs.com/package/shadow-thing`

[9] `https://github.com/eclipse/thingweb.node-wot`

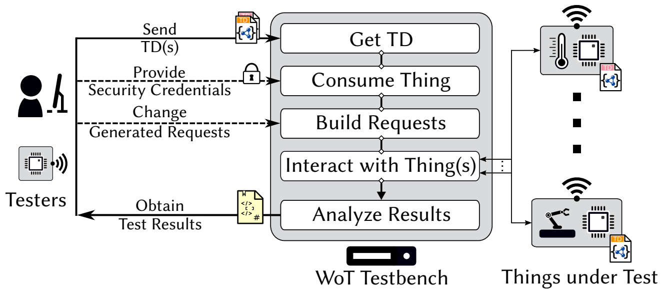


Figure 4: The high-level architecture of the WoT Test Bench can be visualized here, where a human or Thing user can interact with the Test Bench. It is possible to provide the security credentials of the Thing under Test and to change the automatically generated requests. Once the test is finished, a human and machine readable report is provided to the user.

interactions as detailed in Figure 5, a report needs be generated so that the results can be analyzed.

A TD can describe synchronous and asynchronous operations, where reading or writing properties and invoking actions are mostly synchronous and observing properties, receiving events are mostly asynchronous. In our method, we handle these two cases separately as shown by the columns A and B in Figure 5.

Synchronous testing is illustrated in the column A of Figure 5. For Property Affordances, our method starts by reading the property values and validating them according to their Data Schema in their TD. For Property Affordances that are writable (not `"readOnly":true`), it proceeds by fuzz-testing the possible inputs. Our method also checks whether the written values are correctly written to the property values by a sequential read operation. However, this behavior is not required from the Things according to the TD specification and are reported only as warnings.

Similar to testing Property Affordances, Action Affordances can be tested in a synchronous fashion. They are fuzz-tested based on their input Data Schemas when the action requires an input and verify whether the response data conforms to the output described in the TD, similar to verifying the result of reading a property.

Asynchronous testing is shown in the column B of Figure 5. Different from reading-writing properties and invoking actions, other operations can describe asynchronous communication patterns with the Things. Observing a Property Affordance and subscribing to an Event Affordance are two asynchronous communication patterns that a Thing can exhibit.

For asynchronous testing, we start by subscribing to the resources and keep the test running for a configurable amount of time. Any data that is received during this period is tested to its Data Schema definition in the TD. A Property Affordance that is both observable and writable introduces a special case, where its observability can be tested by writing to it. In this case, the Thing should return the written value since observing a property should report all of the changes. Our method takes

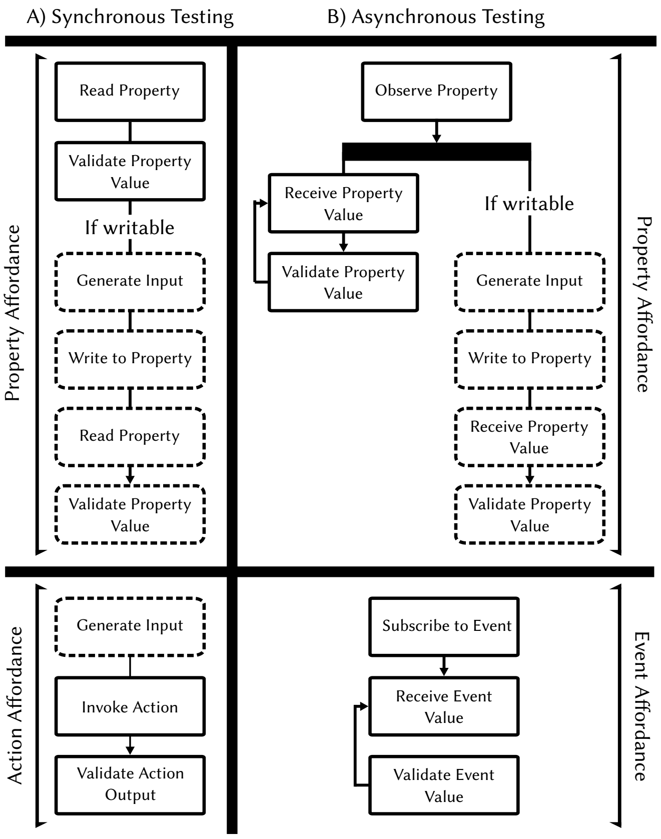


Figure 5: Synchronous and asynchronous Interaction Affordances in a Thing Description (TD) needs to be treated separately in a black-box testing approach. Here, reading and writing to Property affordances and invoking Action affordances are tested in a sequential fashion, whereas in column B),observing Property Affordances and listening to Event Affordances data is done asynchronously.

this into account and it is illustrated in parallel steps in Figure 5: validating new values continuously and forcing the delivery of a new value by writing to the property.

3.3.1 Implementation: WoT Testbench

Our testing methodology is implemented under the name WoT Testbench. It is a Node.js implementation and publicly available[10]. It also uses the `node-wot` library[11] from the Eclipse Foundation's Thingweb project [12].

Furthermore, WoT Testbench can be used as a service over the network, like a Thing. This allows the testing to be customized in runtime since different steps such as the initialization, request generation, testing, and the reports can be offered as different Interaction Affordances.

Security and Safety Considerations: It is important to note that the methodology we have described above interacts with physical devices. Our method does not to include a penetration testing approach where the goal would be to verify if the device has undescribed and unprotected inputs. In the case of WoT devices, safety can be compromised if the device does not check for inputs outside of its accepted schemas or the ranges are

[10] `https://github.com/tum-esi/testbench`
[11] `https://github.com/eclipse/thingweb.node-wot`

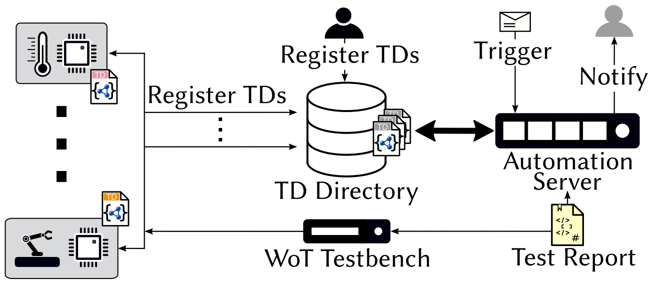


Figure 6: A system of WoT Things can tested continuously for errors. An automation server checks the changes in a database containing Thing Descriptions and in their source code, which results in WoT Testbench black-box testing the Things. The results are communicated to a system developer.

too wide. On the other hand, a Thing can describe its security requirements via its TD, however the security penetration testing would require protocol specific exploits, which are outside of the scope of our methods.

Effect of Physical Properties: Another point to discuss is the challenge of conducting tests over a network and asserting the correct implementation of the Thing in all abstraction levels. Even though our method is able to verify the behavior of the Thing on the network interface level, it is not able to assert whether a physical change in the Thing or the environment around has happened correctly or has happened at all. To our knowledge, black-box testing through the network interface with such verification capabilities has not been researched.

4 Continuous Integration of Web of Things Devices

Our testing methodology requires no change in configuration if the TD of the Thing under test changes. This makes it possible to be used to test the changes in Thing implementations automatically. For this, we use the term Continuous Integration similar to the usual software development pipelines. This allows changes to the Things in a system to be tested automatically and enables notification of the developers of the system in case of a change that results in an error.

In Figure 6, we detail the different components needed in a WoT System that can support a continuous integration procedure. As seen with the component named TD Directory, a database to store TDs is necessary to detect first a difference between two versions but also to store TDs that are not hosted on the Thing. This is especially required for brownfield devices where a TD needs to be generated externally since their source code cannot be modified.

Differently, there can be also changes in a Thing that do not introduce a change in its TD. These can include updates to the lower level software libraries and dependencies, correcting errors in the Thing implementation, etc.

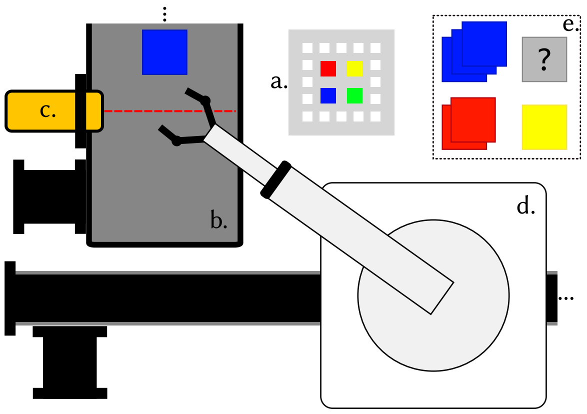


Figure 7: We illustrate the system in question for the first case study where are new color sensor (a.) is to be integrated to the system. The system is comprised of colored cubes moving on a conveyor belt (b.) and that can be detected by an infrared sensor (c.). A robot arm (d.) can pick up the cubes and sort them in a designated area (e.) according to their color once measured by the color sensor. The color sensor has LEDs that need to be activated before a color measurement is taken.

Our continuous integration method can also take these changes into account, where the changes to a source code repository and its following deployment on a device can be tracked and can trigger a test. This is shown in Figure 6, where other ways to trigger a test can be added like plugins.

5 Proof of Concept with Case Studies

In this section, we present two case studies to formulate our proof of concept. We show the use of our multiple contributions, where the development and integration cycle of two IoT devices are streamlined, which allows automation in all phases of the development of a Thing. First, we introduce a color sensor that can be used to detect the colors of objects placed near it. We follow all the steps of the development cycle pictured in Figure 1, including the programming part. Secondly, we take the example of a closed-source IoT light bulb and follow only the steps a and c of Figure 1. For both devices, supporting material is publicly available[12].

5.1 Development of an IoT enabled Color Sensor

We want to integrate a color sensor to a conveyor belt system where colored cubes are moving on a conveyor belt and need to be sorted according to their color via a robot arm as illustrated in Figure 7. The other Things already exist and can be interacted with. The requirements we have for the color sensor are detailed in the Table 1 and their corresponding TD vocabulary terms following the separation of requirements introduced in Section 3.1. Based on these requirements, we manually write a corresponding TD that can be found in our repository. This TD allows to extract the assertions to be

[12] https://wotify.org/library/Philips%20HUE%20White%20Light/general and https://wotify.org/library/Color%20Sensor/general contain the TDs as well as implementation information

	Application Logic		Protocol and Security	
	Textual Requirement	*TD Requirement*	*Textual Requirement*	*TD Requirement*
Case Study 1	Reading color value as a string	Readonly Property Affordance with String Schema and enum	CoAP as protocol	all `href` values starting with `coap://`
	Enabling LEDs	Action Affordance with no input or output	JSON for payload	all `contentType` equals `application/json`
	Disabling LEDs	Action Affordance with no input or output	PSK Authentication	`securityDefinitions` use the `psk` scheme
Case Study 2	Reading light bulb state	Readonly Property Affordance with Boolean Schema	HTTP as protocol	all `href` values starting with `http://`
	Changing light bulb state	Action Affordance with input	JSON for payload	all `contentType` equals to `application/json`
			APIkey Authentication in URI path [13]	`securityDefinitions` use the `nosec` scheme and `href` values contain API Keys

Table 1: The requirements defined in textual form can be translated into requirements specific for a Thing Description (TD). An insight into this process is shown in this table for two case studies. The resulting TDs can be found in our repository. Note: The Application Logic column and Protocol and Security column should be interpreted independently in each case study.

implemented and the node-wot library from the Eclipse Thingweb project satisfies these assertions.

In order to confirm the requirements we have set, we deploy a simulation of the color sensor, as explained in Section 3.2. This allows us to write the application logic in the Consumer side and to confirm that the color sensor will work well with the system.

Once the color sensor's implementation is finished, we follow our black-box testing method introduced in Section 3.3. The `enable` and `disable` actions require no input and produce no output and can be thus tested once. For the color value, we test whether the color values correspond to the Data Schema. Finally, once the tests are complete, the color sensor can be integrated into the system.

5.2 Integration of a closed-source IoT Light Bulb

With the second case study, we aim to show the flexibility and modularity of our contributions with the integration of a closed source IoT light bulb into a WoT system. Thus, we cannot set the requirements of the device and have to use the device as it is. As a result, in this case study we focus on the integration of such a device into a WoT system where the analysis of the requirements and programming steps are skipped. Different from other approaches that are introduced in Section 6, we can use our methods even for closed-source IoT devices since we are not dependent on a concrete framework.

We start with step a of Figure 1 and match capabilities of a Philips Hue Smart Bulb[14] to a TD, as shown in Table 1. This is similar to the previous case study, but instead of writing a TD based on any possible given requirements, it is written for a subset of the already existing capabilities of the device. In this case, Philips Hue API[15] explains the data structures and the possible endpoints for interacting with the light bulb but we choose only one endpoint that allows to change its state.

The TD can be still used for the black-box testing in an automatic fashion and the Philips Hue Smart Bulb can be integrated into a Continuous Integration pipeline as shown in Figure 6. The testing results of this case study are shown in Table 5.2, where the column a provides an excerpt of the individual testing results and the column b gives an overview of the results. These constitute the test report and can be also represented to the developer in a user interface.

5.3 Discussion

The two case studies introduced in this section showed that the TD can be used in the development cycles of single IoT devices and of systems of IoT devices in addition to describing them once they are deployed. Most importantly, in the testing phase our black-box testing approach WoT Testbench made it possible to verify the device behavior in an automatic fashion, first removing the need to understand the TDs and secondly generating the requests without any manual effort. Since the TD makes it possible to abstract the implementation details, we have shown that our method is also applicable for closed-source devices such as the Philips Hue light bulbs as reflected in Table 5.2. We see that there is more improvement needed to enable the testing of security and safety related features of devices.

6 Related Work

Our previous contributions were addressing the simulation and testing stages of WoT devices as independent problems. Most notably in [14], we explain how to simulate virtual Things and how to deploy smart reverse

[14] `https://www2.meethue.com/en-us/bulbs`

[15] Philips Hue API requires a free account to see the contents of the API at `https://developers.meethue.com/develop/hue-api/`

a) Single Test Result

```
{
  "interactionName": "setState",
  "interactionType": "ActionAffordance",
  "testResult": false,
  "sentPayload": {"on":true},
  "receivedPayload": {...},
  "errorId": 550,
  "errorMessage": "Expected Response ..."
}
```

b) Analyzed Results

```
{
  "NonExecutedActions":[],
  "RepetitionDifference":false,
  "PassedAllTests":["rename"],
  "FailedAllTests":[],
  "HasSameErrorId": ["setState"]
}
```

Table 2: Test results of WoT Testbench can obtained in a granular way (a) or in an analyzed fashion (b). The granular way documents every message that has been sent and received during the testing (`sentPayload` and `receivedPayload`) but also the test results for a given Interaction Affordance (`testResult`, `errorId`, etc.). An excerpt of the results of the second case study is shown and point to an error in the `setState` action.

proxies for more resource constrained IoT devices with no manual effort required by the developer. In [15], we use the black-box testing of WoT devices as a case study when introducing a new vocabulary term called *Path* for the TDs that increases the expressiveness of TDs and improve the testing results. However, the use of TDs for defining and validating the requirements, which we introduce in this paper, was the missing piece in order for the TDs to be used for the entire development cycle.

Comparable to our method for simulating IoT devices, DPWSim [16] is also a simulation toolkit but based on the DPWS [17] standard from OASIS standardization body. Fundamentally, DPWS cannot be used for describing existing devices and prescribes conditions to be satisfied by IoT devices. Our simulation method can simulate any IoT device that can be described by a TD, which itself can be used to describe IoT devices with any protocol as long as that protocol has a corresponding URI scheme. On the other hand, Mozilla's Virtual Things Adapter [20] for the WebThings Gateway can simulate IoT devices. However, it requires manual configuration and is constrained to work on the WebThings Gateway software stack.

Another simulation framework, named Cooja, introduced in [18] allows cross-level simulation for IoT devices running the Contiki operating system[19]. It has the advantage of covering more abstraction layers and being able to generate code that can be deployed in an emulated instance or the actual hardware. On the other hand, it imposes the Contiki operating system as a requirement and needs more manual intervention by the developer.

In [13], authors propose an automated testing framework for IoT devices that can use different protocols similar to our approach. However, the focus is testing the interoperability of protocol stacks of different implementations using the F-Interop platform [21]. We start from the application point of view and use the protocol described by the Thing the way it describes it. While we cannot assert on the protocol conformance of the IoT device, we can assert on its conformance to its TD.

7 Conclusion

In this paper, we have introduced our contributions to the development cycle of IoT devices. Out methods allow to use the new Thing Description (TD) standard, which is intended for describing networking interfaces of IoT devices, to be used in the development cycle. We show that one can use TDs for defining requirements, simulating a virtual device and after the programming of the device to test it automatically, which in turn streamline and thus speed up the development of IoT devices. Furthermore, our case studies show that our contributions do not necessarily constitute a framework and can be used independently by relying on the TD standard. With their open source implementations, our contributions can be adopted by anyone and improve their IoT development workflow.

Literature

[1] D. Raggett, K. Ashimura, Y. Chen, "White Paper for the Web of Things", W3C, Tech. Rep., 2016. [Online]. Available: `http://w3c.github.io/wot/charters/wot-white-paper-2016.html`

[2] Guinard, D. and Trifa, V., 2016. Building the Web of Things. Shelter Island: Manning.

[3] M. Kovatsch, R. Matsukura, M. Lagally, T. Kawaguchi, K. Toumura and K. Kajimoto, "Web of Things Architecture", W3C, Tech. Rep., 2020. [Online].Available: `https://www.w3.org/TR/2020/REC-wot-architecture-20200409/`

[4] S. Kaebisch, T. Kamiya, M. McCool, V. Charpenay, M. Kovatsch, "Web of Things Thing Description", W3C, Tech. Rep., 2020. [Online]. Available: `https://www.w3.org/TR/2020/REC-wot-thing-description-20200409/`

[5] A. Palmieri, P. Prem, S. Ranise, U. Morelli and T. Ahmad, "MQTTSA: A Tool for Automatically Assisting the Secure Deployments of MQTT Brokers", 2019 IEEE World Congress on Services (SERVICES), Milan, Italy, 2019, pp. 47-53.

[6] D. Guinard and V. Trifa, "Towards the Web of Things: Web Mashups for Embedded Devices", in Workshop MEM 2009, in proc. of WWW, Madrid, Spain, vol. 15, 2009, p. 8.

[7] A. Wright, H. Andrews, B. Hutton, "JSON Schema: A Media Type for Describing JSON Documents", [Online]. Available: `https://tools.ietf.org/html/draft-handrews-json-schema-02`, 2019.

[8] M. McCool, E. Korkan, "Web of Things (WoT) Thing Description: Implementation Report. Version: 6 Dec 2019", [Online]. Available: `https://w3c.github.io/wot-thing-description/testing/report.html`

[9] Z. Kis, D. Peintner, J. Hund, K. Nimura, “Web of Things (WoT) Scripting API”, W3C, Tech. Rep., October 2019. [Online]. Available: `https://www.w3.org/TR/2018/WD-wot-scripting-api-20191028/`

[10] M. Koster and E. Korkan, “WoT Binding Templates”, Tech. Rep., 2020. [Online]. Available: `https://www.w3.org/TR/2020/NOTE-wot-binding-templates-20200130/`

[11] 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/`

[12] D. Peintner, M. Kovatsch, C. Glomb, J. Hund, S. Kaebisch, V. Charpenay, “Eclipse Thingweb Project”, 2018, [Online; accessed April 11, 2020]. Available: `https://projects.eclipse.org/projects/iot.thingweb`

[13] Kim, Hiun, et al. “IoT-TaaS: Towards a Prospective IoT Testing Framework.” IEEE Access 6 (2018): 15480-15493.

[14] E. Korkan, E. Regnath, S. Kaebisch, S. Steinhorst. “No-Code Shadow Things Deployment for the IoT”. In: Procc of the 2020 IEEE WFIOT. New Orleans, USA.

[15] E. Korkan, S. Kaebisch, M. Kovatsch, S. Steinhorst. “Safe Interoperability for Web of Things Devices and Systems”. In Languages, Design Methods, and Tools for Electronic System Design - Selected Contributions from FDL 2018. Springer International Publishing

[16] S. N. Han, G. M. Lee, N. Crespi, K. Heo, N. Van Luong, M. Brut, and P. Gatellier, “DPWSim: A Simulation Toolkit for IoT Applications using Devices Profile for Web Services”, in *2014 IEEE World Forum on Internet of Things (WF-IoT)*, March 2014, pp. 544–547.

[17] D. Driscoll, A. Mensch “Devices Profile for Web Services Version 1.1”, [Online]. Available: `http://docs.oasis-open.org/ws-dd/dpws/1.1/os/wsdd-dpws-1.1-spec-os.pdf`, 2009.

[18] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne and T. Voigt, Cross-Level Sensor Network Simulation with COOJA,Proceedings. 2006 31st IEEE Conference on Local Computer Networks, Tampa, FL, 2006, pp. 641-648, doi: 10.1109/LCN.2006.322172.

[19] A. Dunkels, B. Groenvall and T. Voigt, Contiki - a lightweight and flexible operating system for tiny networked sensors”, Proceedings of the First IEEE Workshop on Embedded Networked Sensors Tampa Florida USA, Nov. 2004.

[20] Mozilla Foundation. (2020) “Mozilla WebThings Gateway Virtual Things Adapter”, [Online]. Available: `https://github.com/mozilla-iot/virtual-things-adapter`

[21] S. Ziegler, S. Fdida, T. Watteyne, C. Viho. “F-Interop - Online Conformance, Interoperability and Performance Tests for the IoT. Conference on Interoperability in IoT (InterIoT)”, Oct. 2016, Paris, France.

Ege Korkan studied in INSA de Lyon, France, after which he has joined the Embedded Systems and Internet of Things group in the Department of Electrical and Computer Engineering of Technical University of Munich (TUM) in Germany as a Doctoral Researcher. His research revolves around using Web of Things technologies during the design, test and diagnosis of decentralized IoT systems. He is also a contributor to and an active member of the Web of Things Working Group of the W3C.

Address: Technical University of Munich, Embedded Systems and Internet of Things, Arcisstr. 21, 80333, Munich, Germany, E-Mail: ege.korkan@tum.de

Dr. Sebastian Kaebisch is a Senior Key Expert at Siemens Corporate Technology in Munich, Germany. His work focuses on the efficient realization and usage of standardized Internet and Web technologies for the Industrial IoT domain. Sebastian Kaebisch is an active member and contributor of international standardization groups such as ISO/IEC 15118, IEC 63110 and W3C Web of Things. In the latter he is co-chair and coordinates the topics around the W3C Thing Description.

Address: Siemens AG, Otto-Hahn-Ring 6, 81739 Munich, Germany, E-Mail: sebastian.kaebisch@siemens.com

Prof. Dr. Sebastian Steinhorst is an Associate Professor at Technical University of Munich in Germany. He leads the Embedded Systems and Internet of Things group in the Department of Electrical and Computer Engineering. He is also a Co-Program PI in the Electrification Suite and Test Lab of the research center TUMCREATE in Singapore. The research of Prof. Steinhorst centers around design methodology and the hardware/software co-design of distributed embedded systems for use in IoT, smart energy and automotive applications.

Address: Technical University of Munich, Embedded Systems and Internet of Things, Arcisstr. 21, 80333, Munich, Germany, E-Mail: sebastian.steinhorst@tum.de