Professorship of Embedded Systems and Internet of Things
Department of Electrical and Computer Engineering
Technical University of Munich

Web of Things System Description for Representation of Mashups

Adrian Kast

Master's Thesis

Web of Things System Description for Representation of Mashups

Master's Thesis

Supervised by Prof. Dr. phil. nat. Sebastian Steinhorst
Professorship of Embedded Systems and Internet of Things
Department of Electrical and Computer Engineering
Technical University of Munich

Advisor Ege Korkan

Co-Advisor

Author Adrian Kast
Eduard-Kandl-Str. 31b
82211 Herrsching

Submitted on April 2, 2020

Declaration of Authorship

I, Adrian Kast, declare that this thesis titled “Web of Things System Description for Representation of Mashups” and the work presented in it are my own unaided work, and that I have acknowledged all direct or indirect sources as references.
This thesis was not previously presented to another examination board and has not been published.

Signed:

Date:

Abstract

Internet of Things devices have become a nowadays technology and due to this progress, the question how to combine the capabilities of different Internet of Things devices from different manufacturers, has gained importance recently. The World Wide Web Consortium created the foundations for widespread interoperability in the Internet of Things with the publication of the Thing Description standard in the context of the Web of Things. Thing Descriptions allow to interact with new as well as existing Internet of Things devices by describing their network-facing interfaces and how to interact with them in a standardized way that is both human- and machine-readable. An important question that is left in this domain is how to create, represent and share systems of Internet of Things devices, called Mashups. The techniques introduced in this paper improve the management of such Mashups. We propose two representations for such systems that both have unique advantages and are capable of representing interactions with Things, combined with application logic: A subset of the Unified Modeling Language Sequence Diagram presentation, referred to as Web of Things Sequence Diagram, and a Thing Description that is enhanced with additional keyword-object pairs, referred to as Web of Things System Description. For the latter, we present an algorithm to automatically generate code that can be deployed to a device, making it act as a Mashup controller by implementing a Mashup's application logic. By stating their syntactical and semantical foundations, we show how each representation is defined and how it can be validated. Furthermore, we systematically show that both representations can be used interchangeably in the context of representing Web of Things Mashups and demonstrate this with conversion algorithms, which can convert one representation into the other one without losing information. The conversion of both representations thus allows combining their unique advantages, specifically providing an insight to a Mashup's application logic and representing it systematically with open standards, in order to represent Web of Things Mashups. In order to evaluate our research approach, we present the results of a case study with three different Mashup scenarios taken from different sources. We also make the definitions and validation methods for the proposed representations, the reference implementations of the mentioned algorithms and our evaluation publicly available. Our contribution thus allows safer system composition for Web of Things and enables a systematic approach to build Web of Things Mashups.

Contents

1 Introduction

The Internet of Things (IoT) has gained popularity in recent years [1, 2] with the interest in it shifting from its definition towards how one can profit from the possible advantages. It is now already part of many people's lives and, as a result, a huge market for smartwatches, smart TVs, cleaning robots, surveillance cameras and many other internet-connected devices has established itself. Apart from such devices, a variety of IoT platform providers, industrial IoT systems, data stores, analytics and other related services have also emerged.

While the number of IoT devices rises, one challenge gains importance: How to use their capabilities in the most beneficial way in order to create complex systems? Therefore, it is crucial to enable interoperability between devices from different manufacturers and domains that are designed for different frameworks and technical environments. The fact that there is not one best practice or standard has led to the development of many different standards and implementations, which, in turn, results in silos and high integration efforts.

This problem is addressed by the Web of Things (WoT) [3] and a corresponding World Wide Web Consortium (W3C) architecture approach [4]. It proposes Thing Descriptions (TDs) as the central element to describe the network-facing interfaces for composing applications of an IoT device, called a Thing in the context of this paper, and how to interact with them. A standard for a TD [5] has been published by the W3C to solve the problem of interoperability between IoT devices, describing their network-facing interfaces in a way that is human-readable and machine-understandable.

The problem left in this context is how to create systems composed of different Things, to perform a joint task that one Thing is not capable of. These systems consist of an application logic, which is implemented as code executed by a controller as shown in Figure 1 (e), including interactions with involved Things and are called Mashups in the WoT context.

For example, a Mashup that incorporates environmental sensor devices that measure

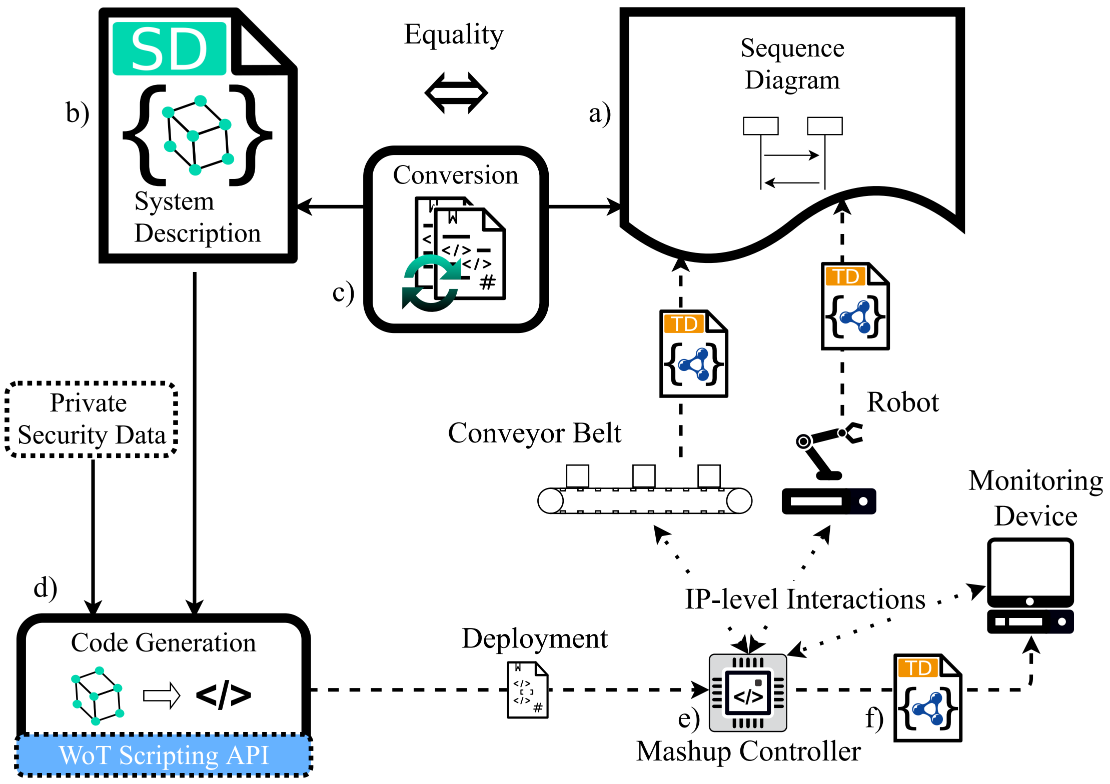

Figure 1: The two representations (Sequence Diagram (a), System Description (b)) for WoT Mashups proposed in this paper that can be converted (c) into each other to combine their advantages. Furthermore, Code Generation (d) according to the WoT Scripting Application Programming Interface (API) standard can be deployed in a Mashup Controller. The generated code implements the Mashup application logic (e) and allows simple exposure of own Interaction Affordances (f).

temperature, humidity and air pressure, can compute a rain probability for the near future and expose it. Other Mashups or devices, such as an actuated window, can read this value and execute actions based on it, without having to read all sensor values and compute the probability by themselves, which is illustrated with *Mashup 1* in Figure 3. As the number of IoT devices increases, the number of possible Mashups increases even more because every additional IoT device enables a multitude of possible Mashups. Adding to this, the possibility that every Mashup can hierarchically work as a virtual IoT device for other Mashups as mentioned above and also illustrated in Figure 1 (f), enables even more complex Mashups.

1.1 Motivation

Considering the significantly rising number of possible useful Mashups, it is necessary that creating, changing, accessing and sharing Mashups becomes as simple as possible to unfold the potential benefits of the WoT. Especially, a developer with two or more Things should be supported to create executable code, which implements any application logic to perform a Mashup task, by minimizing his/her required manual effort.

Furthermore, creating Mashups with at least simple application logic should be possible for anyone, even without the need to program code manually.

1.2 Problem Statement

The introduction of the WoT simplified the creation of IoT device systems, as one does not have to manually look up every required interface of every involved device and how to interact with it. However, creating a Mashup is still a manual task that is done by programming the application logic. This makes it difficult to benefit from advantages such as less error prone and faster development, reusing Mashup application logic and low maintenance effort due to the implementation specific differences.
There already exist Model Driven Engineering (MDE) approaches to create IoT- or even WoT-Mashups that benefit from the mentioned advantages. While they present a major improvement to Mashup creation, there is currently a big fragmentation concerning the representation of the created Mashups. The main reason is that many MDE approaches are built upon frameworks, such as the Eclipse Modeling Framework [6]. This fragmentation does not negatively affect the functionality of the created Mashups, but it limits their reusability, documentation of functionalities and exposure of these functionalities for other Things.

1.3 Contributions

To further improve the management of WoT Mashups, we present the following contributions in Section 3:

- two new formats for representing a Mashup. The first representation is using a subset of Unified Modeling Language (UML) Sequence Diagrams. Representing Mashups with these diagrams comes with the advantage of being human-understandable even for complex application logic.

  The second format represents Mashups with a valid TD that is enhanced with further keywords and called System Description (SD) in this paper. It has the advantage of representing the Mashup in a well-defined textual way, building on existing standards. Also, an SD can be consumed such as a TD by other Things, allowing them to interact with the Mashup as consumers.
- a proof of equality of both representations for the given context, making it possible to convert the representations into each other, as shown in Figure 1 (c). The conversions allow one to save and transmit a Mashup in a standardized textual way while being able to present it human-understandable with a standardized graphical representation.

- an algorithm to automatically generate executable code that implements the application logic of a Mashup, represented by the SD that is the input of the algorithm.

We discuss related work in Section 4, evaluate our contributions in Section 5 with a case study that involves a set of example Mashups taken from different sources and conclude with Section 6.

2 Web of Things

THE WoT concept was first proposed in 2009 [3] to facilitate the interoperability and usability in the IoT. This has then resulted in a standardization group formed in the W3C and the initial standards being published in 2019 [4]. The core concept of the WoT is that Things expose Interaction Affordances, by providing well described interfaces for them and Consumers such as other Things, cloud services or browsers can interact with the Exposed Things via these Interaction Affordances. In this context, an Interaction Affordance is, for example an exposed property that can be read, thus resulting in a *readProperty* interaction.

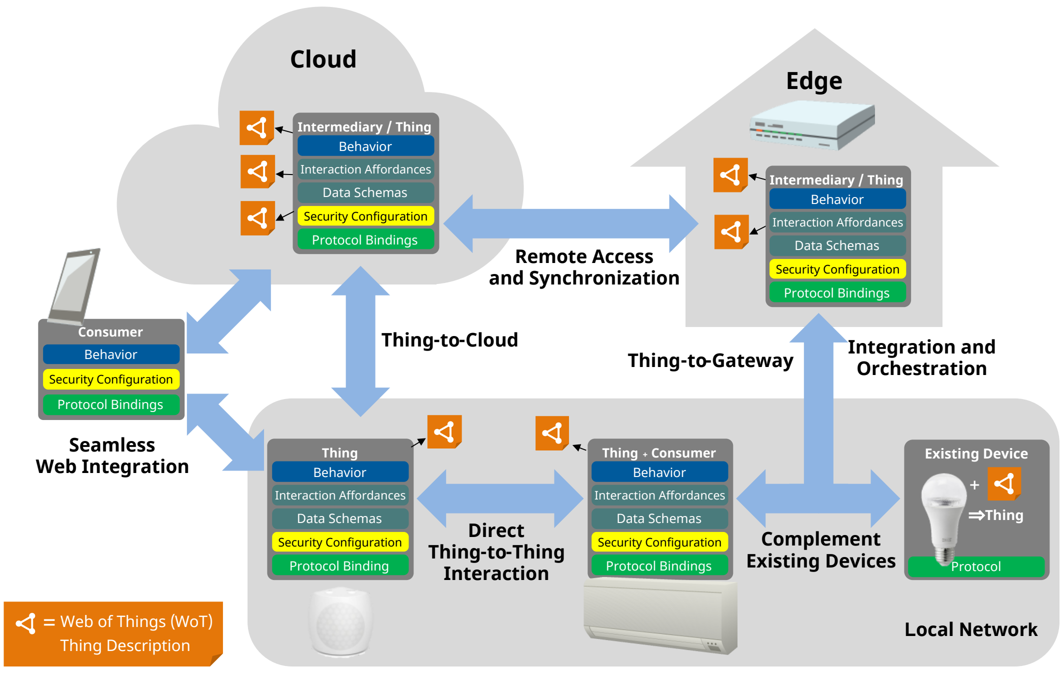

Figure 2: Overview of the W3C WoT abstract architecture, showing interactions between different actors and involved WoT components, from [7].

```json
{"@context": ["https://www.w3.org/2019/wot/td/v1"],
  "id":"urn:dev:home:coff:type123-SNR123456",
  "@type": "Thing",
  "title": "MyCoffeeMaker-Home",
  "description": "Order your coffee remotely!",
  "properties": {
    "status" : {
      "type" : "string",
      "enum" : ["Standby", "Grinding", "Brewing", "Filling","Error"],
      "readOnly" : true,
      "forms" : [{
        "href" : "http://mycoffeemaker.example.com/status",
        "op" : "readproperty",
        "contentType" : "application/json"}]}},
  "actions": {
    "brewCoffee" : {
      "input" : {
        "type" : "string",
        "enum" : ["Espresso", "EspressoDoppio", "Coffee","HotWater"]},
      "forms" : [{
        "href" : "http://mycoffeemaker.example.com/brewcoffee",
        "op" : "invokeaction",
        "contentType" : "application/json"}]}},
  "events": {
    "coffeeEmpty" : {
      "data" : {"type":"number", "minimum": 0,"maximum": 10},
      "forms" : [{
        "href" : "http://mycoffeemaker.example.com/coffeeempty",
        "op" : "subscribeevent",
        "contentType" : "application/json",
        "subprotocol": "longpoll"}]}}}}
```

Listing 2.1: A simple TD that describes the network-facing interfaces of a coffee machine. The TD was shortened by fields such as the security information.

2.1 Architecture of the Web of Things

The WoT architecture standard of W3C defines use cases, requirements and the abstract architecture of the Web of Things, which is illustrated by Figure 2. It is complemented by:

- The **Thing Description** standard defines an equally named representation that is shown as an example in Listing 2.1 and provides information about an IoT device, especially about the possible interactions via the network-facing interfaces of the Thing. These interactions are grouped into:

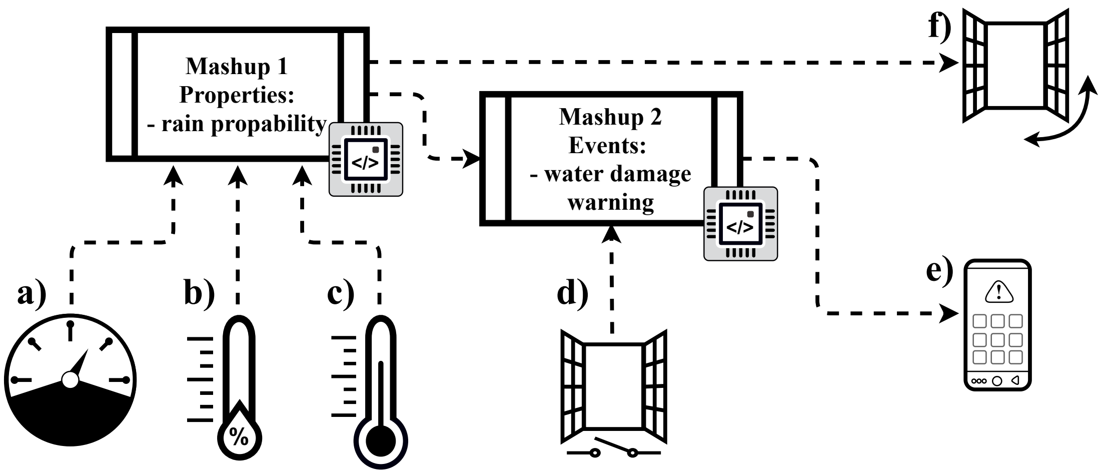

Figure 3: A hierarchical WoT Mashup cascade indicated by the arrows from Exposed Thing to Consumers. The two Mashups are both Consumer and Exposed Thing at the same time and involve the Things: Barometric air-pressure sensor (a), humidity sensor (b), temperature sensor (c), window position sensor (d) and the Consumer: Smartphone (e), actuated window (f).

  - *Properties* can be read, written and observed and represent a state of the Thing, such as a sensor value. An example for a property is *status* in Listing 2.1, which represents the current operation state of the coffee machine and can only be read.
  - *Actions* can be invoked and execute a function of the Thing, which might manipulate its state, e.g., executing a movement. In the example TD (Listing 2.1) the action *brewCoffee* illustrates how such an action can be described.
  - *Events* can be subscribed to and result in a notification each time the event occurs, for example any alert such as the *coffeeEmpty* event, which notifies a user that if the machine runs out of coffee beans, in the Listing 2.1.

- The **Binding Templates** [8] provide information on how IoT platforms can be integrated to create Things in a WoT-conform way. This information can be used to create protocol bindings for a TD and implementations of these bindings to interact with a Thing as a Consumer.
- The **Scripting API** standard [9] supports the development of scripts to define the behavior of Things and Consumers. Therefore, it provides specifies functions to simplify the creation of scripts that discover, fetch, consume, produce and expose TDs. The specification defines an ECMAScript-based API, which has a reference implementation[1].

Furthermore, security and privacy guidelines are provided in the architecture specification, but they are out of the scope of this paper.

1 https://github.com/eclipse/thingweb.node-wot

2.2 Mashups in the Web of Things

One of the major benefits in the WoT is the simple creation of WoT Mashups[2], which refer to the similarly named mashups in the Web 2.0 and are mentioned in the WoT context since its presentation [3, 10]. The manually created code for such a Mashup is shown as an example in Listing 2.2. These Mashups combine different Things and application logic to perform a joint task. Therefore, the Mashup controller consumes the TDs of all Things involved in the Mashup and executes interactions and processing according to the Mashup's application logic. Every Mashup can also expose the added functionality by exposing own Interaction Affordances for other Consumers.
Figure 3 illustrates this for a smart-home use case, where Things are exposed hierarchically. The use case mentioned in Section 1, which involves environmental sensors and a Mashup controller that computes a rain probability to expose it to an actuated window, is demonstrated with *Mashup 1* in this figure. Additionally, a second window that is equipped with a position sensor, is shown together with a second Mashup (Figure 3-*Mashup 2*) to provide an alert event that notifies a subscribed Consumer (smartphone) that it will probably rain in the near future if the window is opened.

```
WoTHelpers.fetch("http://localhost:8081/Mashup1").then(async (tdMashup)
    ↪  => {
  WoTHelpers.fetch("http://localhost:8082/windowSensor").then(async (
    ↪ tdWindow) => {
    try {
      const mashup = await WoT.consume(tdMashup)
      const window = await WoT.consume(tdWindow)
      const rainProbability = await mashup.readProperty("
    ↪ rainProbability")
      const windowPosition = await window.readProperty("windowPosition"
    ↪ )
      if (rainProbability > 50 && windowPosition === "open") {
        console.log("You should close the window!")
      } else {
        console.log("Nothing to do.")
      }
    } catch (err) { console.error("Script error:", err)}
  }, (err) => {console.error("Cannot fetch window td:", err)})
}, (err) => { console.error("Cannot fetch mashup td:", err) })
```

Listing 2.2: Example for a manual created mashup code, which implements the Mashup 2 in Figure 3 without an own event emission, but just printing a message if the window should be closed. The code uses the *wot-servient* command of the Scripting API reference implementation to handle protocol bindings.

2 Throughout this thesis, the word *Mashup* is capitalized to denote a WoT Mashup.

3

Mashup Management Approach

We present our approach to improve the management of WoT Mashups in this section. Therefore, we start explaining the motivation and scope of the approach and continue as listed:

- We define the Sequence Diagram (Section 3.2) and System Description (Section 3.3) representation and present how to validate them.
- In Section 3.4 we show that using the representations interchangeably is possible by proving their equality.
- We note the conversion and code generation algorithms in Section 3.5.

Finally, we present a discussion of our approach in Section 3.6.

3.1 Motivation and Scope of Work

The WoT Scripting API provides functions to write communication protocol- and implementation-independent application code for a Mashup. To remedy this currently manual process of writing application code, a model to express the application logic in a standardized and textual way is necessary. This can be used as input for the automatic code generation, to create program code that can be deployed to a Thing to act as a controller, as shown in the industrial Mashup example in Figure 1. To make it accessible and use existing standards, especially in the context of the WoT, we propose the SD to describe a Mashup including its interactions and application logic.
In order to provide an insight, even for complex application logic, we propose an accompanying graphical representation based on the UML Sequence Diagram. The graphical representation is also standardized by the use of UML elements and gives a human-understandable overview of Mashup application logic and executed interactions. To be able to always represent a Mashup in the required format, which allows one to benefit

from advantages of both representations, we provide conversion algorithms.
As input for the creation of Mashups including application logic, sequences of interactions, written in UML Sequence Diagram representation, are used. We refer to these sequences as Atomic Mashups in the scope of this work and they can be simply created manually (low effort) or can be also generated automatically in the future.
To constrain the Atomic Mashups to useful combinations and react to asynchronous data-pushes, we work with the assumption that they consist of at least one receiving interaction: *ReadProperty*, *subscribeEvent*, *observeProperty* or *invokeAction* with a return value, followed by at least one sending interaction: *WriteProperty* or *invokeAction*, where the interaction direction is defined by the Mashup's perspective. Thus, we can define the sending interactions to be executed upon reception of the asynchronous data-pushes, resulting from subscriptions or observations.
The event on which the sending interactions are executed can be defined for every Atomic Mashup, to be either on reception of the first asynchronous data-push, or on reception on the last asynchronous data-push. The sending on first reception can for example enable faster responses to polled values of redundant sensors. If one Mashup includes two sensors, which measure the same physical value, but they are only responding sporadically due to for example energy restrictions, one can poll both sensors and execute the sending interactions as soon as the first response is received. The sending after reception of all data-pushes is for example suitable if different values are measure, which are all required for processing data before the sending interactions. If no asynchronous receiving interactions, i.e. *subscribeEvent* or *observeProperty*, are part of the Atomic Mashup, the sending interactions are executed immediately after the receiving interactions.

3.2 Sequence Diagram

The creation of visual Mashup presentations is motivated by the idea of giving an insight into the application logic which is simpler to understand than for example program code, which is a state-of-the-art representation. In the remainder of this section, we define WoT Sequence Diagrams, present how one can validate instances of these diagrams and show, by comparing them to UML standard elements, that our diagram presentations are UML-conform. Finally, we present how one can create a WoT Sequence Diagram manually.

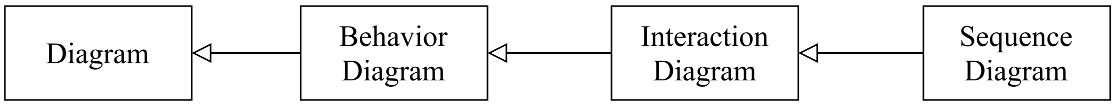

Figure 4: Hierarchical order of Sequence Diagrams inside the UML specification.

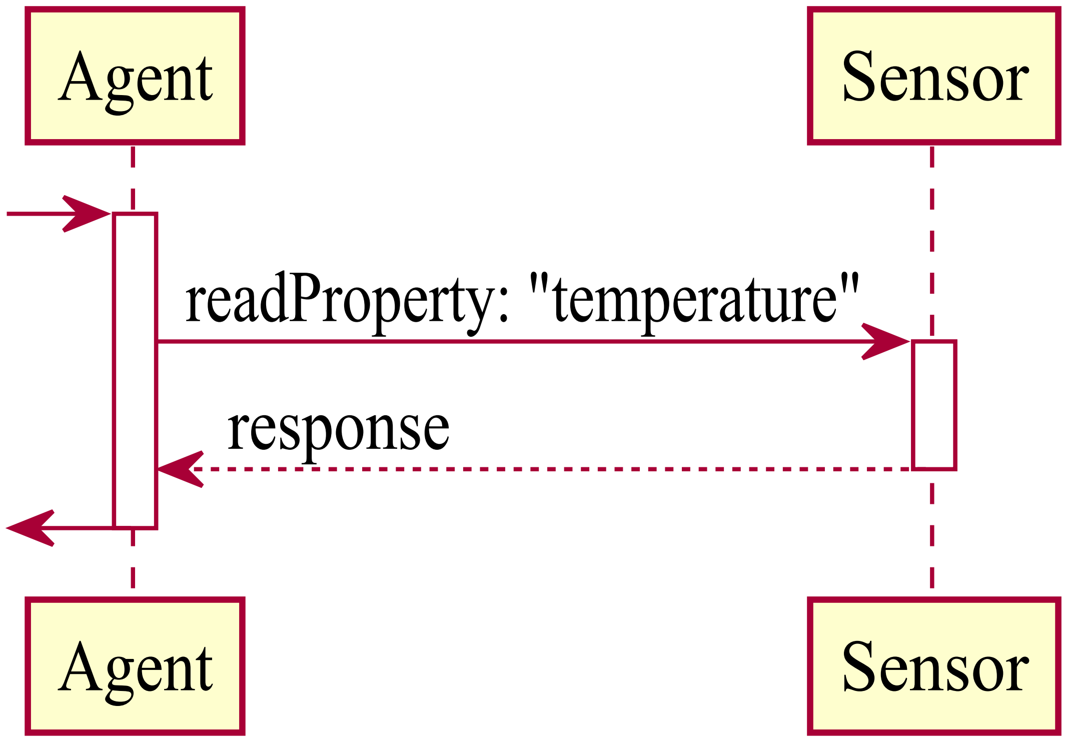

Figure 5: UML Sequence Diagram presentation, generated by a PlantUML implementation, of a Mashup controller reading the property *temperature* from an Exposed Thing by sending a synchronous message and receiving the response.

3.2.1 Representation and Validation

To generate a presentation of a Mashup, we propose a subset of UML Sequence Diagrams, which are a standardized presentation method that is retrieved in the UML specification as shown in Figure 4. More specifically, we define a subset of PlantUML [11], which provides a broadly supported, simple to integrate diagram generator implementation, with a corresponding textual representation. A simple example of this textual notation is the Listing 3.1 that defines the presentation shown in Figure 5.
PlantUML and its entire functionality are not always UML specification compliant. Thus, to ensure the UML compliance, we reference every graphical element to its corresponding UML element. The usage of only the defined subset of PlantUML can be checked by validating it against a context-free grammar[1] which we define in extended Backus-Naur form (EBNF) to specify the allowed language. A Backus-Naur form [12] (BNF) definition of a grammar is machine-readable and BNF extensions are often used to define programming languages, thus a multitude of parser generators and visualization algorithms exist.
We use a common EBNF notation defined by the W3C in [13]. In contrast to BNF,

1 A parser generator, such as REx (`https://bottlecaps.de/rex/`), can generate a validator from the grammar.

```
@startuml diagramName
[-> "Agent"
activate "Agent"
"Agent" -> "Sensor":readProperty: "temperature"
activate "Sensor"
"Sensor" --> "Agent":response
deactivate "Sensor"
[<- "Agent"
deactivate "Agent"
@enduml
```

Listing 3.1: PlantUML notation instance, defining the interaction between a Mashup controller, referred to as agent and a temperature sensor.

it allows to include symbols in the specified language, which are required for the used PlantUML subset. Additionally, it has the advantage of resulting in shorter grammar notations and allows referencing production rules of other grammars. Other EBNFs or augmented Backus-Naur forms with a similar expressiveness could be used interchangeably. In Listing 3.2, an extract from this grammar is shown and the full grammar can be obtained as a listing from the Appendix A.9 or as a file from our repository[2]. The grammar shown in the listing defines how a *readProperty* interaction is composed, therefore:

- *interactionReceive* optionally begins and/or ends with a *getset* and intermediate must be *interactionPre* followed by *receiveSubs*, *receiveInv*, *receiveObs* or *receiveRead* (line 1).
- *interactionPre* is composed of the strings *' "Agent" '* and *' − > '* followed by *interactionTo* followed by the string *':'*, with an *S* after every element (line 2).
- `<?TOKENS?>` defines the end of non-terminal definitions and begin of terminal definitions (line 8).
- *L* optionally begins with S, followed by the Unicode symbol 0xA or the Unicode symbol sequence 0xD 0xD or the Unicode symbol 0xD. The Unicode symbols occur one or more times, then it ends optionally with S again (line 9). This Terminal denotes a line break.
- *S* is defined as one or more concatenations of the Unicode symbol *0x20* or *0x9* and represents the whitespace (line 10).

The usage of this grammar results in any further Sequence Diagram notation to be a valid PlantUML instance, which can be interpreted by any PlantUML implementation. The possible resulting Sequence Diagram presentations are visualized in Figure 6, where one example of every graphical element we can use to represent a WoT Mashup is

2 For the remainder of the thesis, our repository refers to the ZIP file that is submitted together with this document.

```
1 interactionReceive ::= getset? interactionPre ( receiveSubs |
    ↪ receiveInv | receiveObs | receiveRead ) getset?
2 interactionPre ::= '"Agent"' S '->' S interactionTo S ':' S
3 receiveRead ::= 'readProperty:' receiveMiddle readResponse L deactTo L
4 receiveMiddle ::= S interactionName L actTo L
5 readResponse ::= interactionTo S '-->' S '"Agent"' S ':' S 'response'
6 deactTo ::= 'deactivate' S interactionTo
7 interactionTo ::=  '"' Ntitle '"'
8 <?TOKENS?>
9 L ::= S? (#x000A | #x000D #x000A?)+ S?
10 S ::= [#x0020#x0009]+
11 Ntitle ::= [a-zA-Z] ([a-zA-Z0-9] | '-' | '_')+
```

Listing 3.2: Extract from EBNF grammar definition for the used PlantUML subset, the whole grammar is included in the Appendix A.9. The upper part defines the non-terminals (lines 1–7) and is separated from the terminals (lines 9–11) by the *tokens* expression (line 8).

visualized. The labels in the figure are explained with referring[3] to the corresponding elements of the UML standard [14] and describing the meaning of these elements in the Sequence Diagram subset:

1. The UMLShape (Gate) and UMLEdge (Message, synchronous call or reply) with UMLLabel (Message) and the literal *top:, function:, action:* or *property:* followed by a name. It represents the equivalent of a *function call* for programming languages and defines, which application logic is described in the diagram.
2. The UMLShape (Lifeline, line) with UMLLabel (Lifeline, rectangle) represents the lifeline of either the device executing the Mashup application logic as a controller, called *Agent* or one of the Things the controller interacts with.
3. The UMLShape with UMLLabel (Comment) represents the getter and setter function for Mashup internal variables or Mashup properties.
4. The UMLShape (CombinedFragment) containing a UMLLabel (CombinedFragment) with literal *strict* and two UMLShape (InteractionOperand). It represents a WoT interaction sequence consisting of one or more *receive* interactions that are followed by one or more *send* interactions as explained in Section 1.
5. The UMLShape (CombinedFragment) with literal *par* containing UMLLabel (CombinedFragment) and two or more UMLShape (InteractionOperand). It represents an interaction set with all interactions direction either *send* or *receive* and defines that the contained interactions can be executed in arbitrary order.
6. The UMLEdge (Message, synchronous call) with UMLLabel (Message) represents a WoT operation.
7. The UMLEdge (Message, reply) with UMLLabel (Message) *confirmation*, *response* or *output* represents a WoT operation reply.

3 All references refer to version 2.5.1 of the UML standard.

8. The UMLEdge (Message, asynchronous signal/call) with UMLLabel (Message) *data-pushed* represents asynchronous pushed data, due to an event subscription or property observation.
9. The UMLShape (ExecutionSpecification) representing the ability of a lifeline to initiate WoT operations.
10. The UMLShape (CombinedFragment) containing a UMLLabel (CombinedFragment) with literal *break* and one UMLShape (InteractionOperand) containing the UMLLabel (InteractionConstraint) *data-pushed*. It represents the execution of the *send* interactions on the reception of the first asynchronous pushed data. If it is not included in the *receive* interactions fragment, the *send* interactions are defined to be executed after receiving all data-pushes at least one.
11. The UMLShape (CombinedFragment) containing a UMLLabel (CombinedFragment) with literal *loop* and one UMLShape (InteractionOperand) containing a UMLLabel (InteractionConstraint). It represents the equivalent of a *loop* in programming. The loop can either be defined to execute x times, forever or every x milliseconds.
12. The UMLLabel (DurationConstraint)[4] represents a duration that the application logic execution has to pause.
13. The UMLShape (CombinedFragment) containing a UMLLabel (CombinedFragment) with literal *alt* and one or two UMLShape (InteractionOperand). Each UMLShape (InteractionOperand) containing an UMLLabel (InteractionConstraint). The UMLShape (CombinedFragment) represents the equivalent of an *if* statement in programming languages. Optionally, it can also include an *else* equivalent.

By the given references of UML standard elements for every element contained in the PlantUML subset, together with their meaning for WoT Sequence Diagrams, one can simply retrace the UML-conformity of these elements. With the usage of the defined PlantUML notation subset, we can thus assure that the generated Sequence Diagrams are always UML-conform.

3.2.2 Manual Creation

To create a Sequence Diagram manually one has to create a JSON file which contains the TDs of all Things that are involved in the Mashup one wants to represent. The content of this file has to be only an JSON Array, represented by brackets, which contains the TD-objects separated by commas.

Furthermore, a PlantUML file has to be created (file extension `.puml`), which contains one Sequence Diagram for every sequence of application logic elements, i.e., a function,

4 due to the used UML Sequence Diagram generator the additional UMLEdge (DurationConstraint) is omitted, and the constraint always applies to the previous and next element.

an action, a property or the logic to be executed on Mashup execution. An example of such a Sequence Diagram is A.3, where the PlantUML notation of the Sequence Diagram presented in Figure 6 is given.
Every diagram starts with `@startuml` and the name of the diagram, which has to be unique for every diagram inside the same PlantUML notation file. The second line contains the call from *gate* to *agent* lifeline `[->"Agent":` followed by the diagram type, i.e., *top, function, action* or *property* and the name of the Mashup, function, action or property, respectively. The beginning of one diagram is then completed with the activation of the *agent* lifeline (lines 1–3 in Listing A.3).
The application logic has to be added for every created diagram in the PlantUML notation file. Each application logic element can be of the types:

- The **note** represents a getter or setter function for a property or variable of the Mashup. An example for it are the Lines 4–7 in Listing A.3.
- The **reference** represents the call of another function or action inside the same Mashup and consequently also defined inside the same PlantUML notation file. The reference can be noted by `ref over "Agent"`, followed by *function* or *action*, a colon and the name of the reference target, ended by another line containing `end ref`.
- The **strict** fragment represents an Atomic Mashup, as explained in REF. The PlantUML notation of all possible interactions for receiving and sending are shown in Listing A.2.
- The **loop** is started by the key *loop* and either a periodical expression, such as `every 500ms`, or a number of iterations, e.g., `25x` or `forever`. The content of the loop is other application logic elements and it ends with the key *end*.
- The **wait** element is represented by three dots, the key *wait*, a duration in milliseconds and again three dots (line 30 in Listing A.3).
- The **alternative** represent the if statement in program languages. Its resulting execution is determined by the condition, which can be a Boolean variable/property, a comparison, or the key *not*, *allOf*, *oneOf* or *anyOf*, followed by more conditions (or one condition for *not*) separated by comma and surrounded by parenthesis. After the first line, which contains the key *alt* followed by this condition, the content of the if statement is noted as application logic elements. After the separation by an *else* statement, one can also note application logic to be executed instead if the condition is false (lines 31-47 in Listing A.3).

The application logic elements are written concatenated, only separated by a line break between each of them. One can also use indentation to structure the elements, since nesting them, e.g., for loops or alternatives, is allowed.

Then every diagram ends with the return of the initial call, from *agent* lifeline to *gate*, followed by the deactivation of the *agent* lifeline and the `@enduml` statement (lines 49–51 in Listing A.3).

Finally, one can validate the created WoT Sequence Diagram notation file against the EBNF grammar, which is part of our repository. This can be done by calling the main command line interface function we provide (an SD and code will additionally be created by calling it) with the paths to the folder which contains only the created Sequence Diagram[5] and the JSON file, which contains the TDs, as parameters, such as `npm start ./pathToFolder/SeqDs ./pathToFolder/TDs.json`.

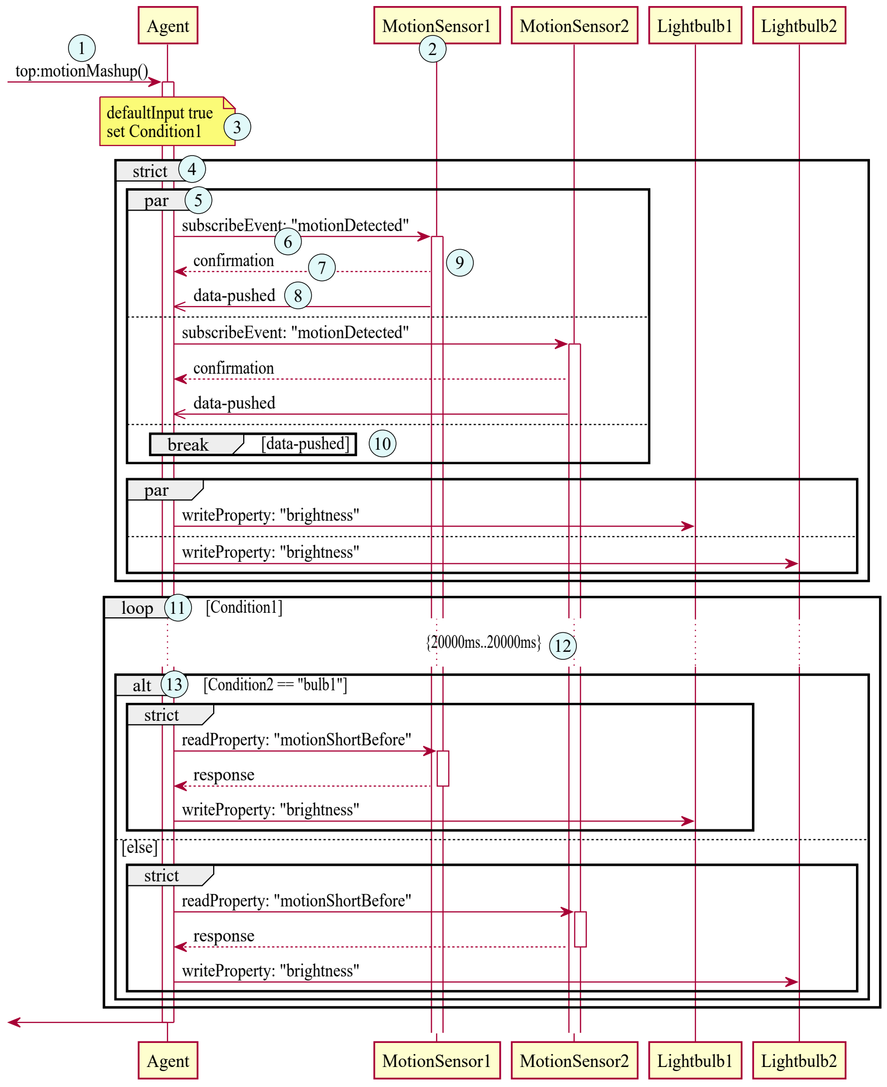

Figure 6: Example of an UML presentation of a Mashup, showing every UML element used in WoT context at least once. The represented application logic controls two lightbulbs by processing the received information of two motion sensors. The PlantUML notation used to generate this diagram is listed in the Appendix A.3.

5 The folder can also contain several diagrams that share the same Things.

3.3 System Description

In the following paragraphs, we present the WoT System Description representation and motivate its usage. Furthermore, we explain the syntactical and semantical definition of this Mashup representation and show how an instance of it can be validated and manually created.

3.3.1 Representation and Validation

The SD representation of a Mashup, which we propose, is based on a TD with additional keywords. To be able to describe a Mashup instead of a single Thing, the SD allows to describe one or more Things the Mashup controller interacts with. Furthermore, application logic can be represented, consisting of executing interactions and programming structures e.g. *loops*, *if*-statements or *wait* commands. The SD representation has the advantages:

- It enables one to represent a model of a Mashup that can be used as input for automatic code generation.
- The representation can express functionality and data that is for internal use or externally accessible via WoT Interaction Affordances, which are simple to access for other Things.
- It is framework independent by building on open standards: TD, JavaScript Object Notation (JSON), JSON-Linked Data (JSON-LD) and JSON Schema.

The keywords which are used in the SD in addition to the keywords of a TD are listed in the Appendix A.4, with exception of the most important added keywords that are explained more specific:

The *Things* keyword defines a JSON object that has to be on the top level of the SD and its properties have the *title* key of a TD. The properties are also JSON objects themselves and include a fragment or the entire TD of a Thing. The *Things* child elements represent all Things that the Mashup can interact with. This structure allows required forms to interact with, to be referenced inside the SD. Therefore, every *forms* child element has to have not only a *href* key like in the TD, but also an *op* key as described in the TD specification. This property can have the value of one or more operations to indicate all semantic intentions for which the form is valid.

The *Path* keyword defines a JSON array and can be a property of an action-object, function-object, property-object or on the top level of the SD. Its functionality is similar to the one of the *Path* keyword described in [15], where the *Path* represents an array of executable interactions. The path elements we present

Type of item	Represents
interact	interaction sequence, called Atomic Mashup
wait	a time delay before further task execution
case	a conditional execution
loop	the repeating execution of another *path* array
get	getter function for Mashup variables, properties or default values
set	setter function for variables or properties
ref	reference to an action or a function

Table 3.1: Defined *Path* array item object types and their corresponding representation

can furthermore represent application logic and are intended to be executed: On action invocation for action-object, on function call for function-object, on property read for property-object and on Mashup code execution for the top level. The *Path* array item objects have to be one of the object types shown in Table 3.1.

The *Functions* and *Variables* keywords enhance the SD by representing internal structures that are, in contrast to *Properties* and *Actions* keywords, not intended to be externally accessible. Both keywords define JSON objects and have to be on the top level of the SD. Their properties are also JSON objects and represent Mashup internal functions and variables. An example for the *functions* keyword with one function can be found in the Appendix A.8.

The aforementioned vocabulary can thus describe a Mashup in a textual format where a JSON-LD representation, like TD, would be possible. Similar to the TD we annotate and specify the syntax with a JSON Schema[16], which is available in our repository. It allows to validate an SD instance and also gives guidance on how a JSON serialization of an SD should be. Therefore, in a JSON Schema constraints for the JSON document content, such as properties of an object and annotations, e.g., descriptions and titles can be defined. The Schema we define also includes references to the TD Schema[6] to ensure that every SD instance is also a valid TD instance[7] and to validate the TD fragments of *Things* properties.
In addition to the JSON Schema, we define a JSON-LD [17] context for the SDs in order to make semantic statements and links about every possible content of an SD. The JSON-LD instance contains machine-understandable expressions, e.g. that an SD belongs to and enhances the semantic context of a TD. Thus, every Mashup can be described in a machine-understandable way by representing it with an SD, which contains

6 https://github.com/w3c/wot-thing-description/blob/master/validation/td-json-schema-validation.json
7 In our proof of work we replace the string-format *iri* with *uri* due to the limitations of the used JSON Schema Validator Ajv (`https://ajv.js.org/`).

a link to the JSON-LD context in our repository. The semantic foundations of the TD that we enhance are presented in [18].
With the aforementioned explanations and the JSON Schema and JSON-LD context in our repository, one can simply retrace the definition of the SD Mashup representation and validate a given SD instance against this definition.

3.3.2 Manual Creation

The creation of a System Description is manually possible by writing an JSON instance. In the following we will present the most systematic and error avoiding way of creating an SD, apart from the automatic creation with a Sequence Diagram as input.
One can start by opening the *manualSD* file in our repository. It includes a Mashup template JSON instance and also import statements for TypeScript definitions that describe the SD. These imports enable one to be supported by code editor features, such as automatic code completion[8].
Continue with adding the following information to the SD:

- The TDs of all Things involved in the Mashup that one wants to represent. They can be added by creating a child element named with the Things title[9] under the *things* property of the SD and paste the TD object in there.
- The application logic elements to represent the application logic of the Mashup. These elements can be added as content of the *path* property of different parent elements, as explained in the Section 3.3.1. The *path* keyword contains a JSON Array with the application elements listed in Table 3.1. The properties of each of these application logic elements, e.g. wait, loop or case, can be retrieved from the JSON Schema that defines the SD or, in a simplified version, from the TypeScript definitions in A.1. A wait-object is for example {`"wait": 500`}, which defines a pause of 500 milliseconds. The decision where application logic should be placed can be done with the following criteria:
  - The content of the *path* on the top-level can be used to execute the included logic immediately on Mashup code execution (without external trigger from a Consumer).
  - The content of the *path* of a function can be used to reuse the included application logic inside the Mashup but do not create Interaction Affordances for other Consumers.
  - The content of the *path* of an action can be used if the application logic should be externally accessible through Interaction Affordances hosted by

8 Part of the most modern code editors, for example called *IntelliSense* in Visual Studio Code (`https://code.visualstudio.com`).
9 If a title is the same for more than one Thing in the Mashup they can be renamed, for example by enumerating them.

the Mashup controller. If the logic should furthermore be executed upon Mashup code execution once, a reference object with the corresponding action as target inside the path on the top-level, can be added.

- For every variable and property that is used inside the application logic elements, e.g., in get, set or case elements, one needs to create a child object under the *variables* or *properties* keyword respectively. The keys of such a variable/property object are similar to the ones of a property in a TD. However, a property in the SD can furthermore have a *path* key that contains application logic which is executed on a *readProperty* operation on this property. A variable in the SD should not contain a *form* key, as it will be ignored since variables are defined for internal use only.

Finally, the *createSD* npm script, which can be run by executing `npm run createSD` in a console, can be called to write the created SD to a JSON file and validate it against the SD JSON Schema.

3.4 Equivalence Proof

To be able to benefit from the advantages of both presented representations, we prove their equality in this section. This is needed to ensure that the representations can be systematically converted into each other. By doing so, the following proof guarantees that conversions do not result in a loss of information. To the extent of our knowledge, a relationship between the standardized UML Sequence Diagram presentation and a semantically well-defined model for an IoT device system has not been established so far, which is further motivating the need for a proof.

3.4.1 Methodology

To prove the equality of the representations in the context of representing Mashups, we define a Mashup formally with a top-down approach. Additionally, we check the resulting definitions for consistency within both representations. Therefore, we compare the semantics of the representations, explained in the next paragraphs, with our Mashup definition.
For the Sequence Diagrams, the EBNF grammar and the PlantUML conversion define the space of possible UML elements as shown in Figure 6. From these UML elements formalisms can be concluded, and we follow the definitions in [19], where it is shown which UML element results in which semantic trace.
The SD can be validated to follow the Mashup formalism by checking the possible design space spanned by the JSON schema definitions, which is semantically specified with the given JSON-LD context and the annotations in the JSON Schema.

For the proof of equality, we focus on the semantic consistency with the Sequence Diagram representation, as it is an established standard for interaction representation that comes with defined semantic meanings, while the SD is based on a JSON-LD notation that explicitly allows to define semantics.
In the remainder of this section, we first define a Mashup formally and the order between single application logic elements (Equations 3.1–3.3 in Section 3.4.2). We continue with presenting the equality of the representations for every application logic element, e.g., loop, Atomic Mashup or conditional execution (Equations 3.4–3.17 in Section 3.4.2). Building up on these equations, we show that one can conclude the equality of the WoT System Description and WoT Sequence Diagrams with Equation 3.18 in Section 3.4.2.
The steps in the following proof and some definitions can be better understood after reading [19]. However, for scientific completeness, we still note all equations and hereby indicate that a detailed explanation of all definitions would require more space.

3.4.2 Mashup overview and application logic

The equality which is the simplest to show is that the TDs given with the diagram representation contain the same information relevant for building a Mashup as the TD fragments of the *Things* vocabulary term of the SD. The reason is that each *Things* child element is defined to equal the Mashup-relevant parts of one TD. Furthermore, a set of PlantUML Sequence Diagrams equal in total to the SD *Functions*, *Actions*, *Properties*, *Variables* and *Path* properties. The *Variables* and *Properties* without a *path* property can be deduced from the diagrams, where their name is represented with their first occurrence.
One Sequence Diagram, out of the set of diagrams representing the Mashup, equals exactly to one element in the *Properties*, *Functions* or *Actions* Property of the SD or the *Path* on the top level of the SD. Thus, the title of the SD equals the name of the sequence started with the *top* keyword and each function, action or property name equals one sequence diagram. The corresponding diagram starts with a message from *gate*, the event occurrence at the diagram border in the top left corner, to the *Agent* lifeline with *function*, *action* or *property* keyword followed by a colon and the name.
The content of these diagrams and the content of the *path* properties in the SDs are representing the application logic elements that have to be executed in the given order. Thus, one sequence of application logic elements execution e can be defined as:

$$e := e_1e_2...e_n|\forall_{i\in n} : e_i \in C, n \geq 1 \tag{3.1}$$

where e is defined as concatenation of n elements of the set of application logic elements C that consists of at least one element.
The application logic representation is ensured to be valid by the EBNF grammar we defined, allowing one or more application logic elements following each other that are

ordered by being concatenated in the diagrams. According to the UML specification, this results in weak sequencing, but we can assume the result to be strictly sequenced. The reason is that all Atomic Mashups and with them all event occurrences, except the application logic call and return as first and last messages, are encapsulated within a *strict* combined fragment and every interaction includes one event occurrence on the Mashup controller lifeline.

By defining two substrings u and v of one execution sequence e, we show that the order of execution is according to Equation 3.1:

$$\begin{aligned} u &= e_1, e_2, ..., e_i \\ v &= e_{i+1}, e_{i+2}, ..., e_n \\ \text{strict}(u, v) &= e_1 e_2 ... e_i e_{i+1} ... e_n | \forall i \in \{1 \leq i \leq n-1\} \end{aligned} \tag{3.2}$$

Equation 3.2 proves that the order of the execution elements equals the definition. The strict function in the equation is defined with X and Y, which represent two not further constrained sets for the remainder of the paper:

$$\text{strict}(x, y) := xy \tag{3.3}$$

$$\text{with } x \in X, y \in Y$$

In the SD, the application logic elements are represented by the *path* property defining a JSON-Array with at least one item, which is specified to be ordered by the JSON-LD `"@container": "list"` property-value pair.

Each element of the application logic has to be either a getter function, setter function, reference, Atomic Mashup, loop, pause execution or a conditional execution. This is ensured by the syntax rules of both representations.

The preceding equations and explanations show that Sequence Diagrams and SDs are generally equally structured, can contain the same application logic elements and their execution order, of these elements, is equal to the one we define for a Mashup in Equation 3.1.

Application logic elements

We will show the equality of representation of every application logic element, a WoT Mashup can contain such as a loop, in the following paragraphs. This is necessary to prove the equality of both representations, which enables combining their advantages. The term *combined fragment*, which is used commonly this section, refers to the UML element that consists of a box and a literal, e.g., *par*, *loop* or *alt*. This element is illustrated in Figure 6, e.g., Labels 5, 11 and 13, and contains further UML-elements, which are ordered according to the definition of the literal of the fragment.

The **loop** element of the application logic has to represent the contained application logic and information about how to repeat the execution of this logic. The loop can either be defined to repeat the execution by a given number of times, or to loop infinite times with or without a given duration interval per loop cycle. The resulting execution order of a loop with m repetitions and the containing application logic e with "n" elements is defined as:

$$\text{loop}(m, e) := e_{11}e_{12}...e_{1n}e_{21}e_{22}...e_{mn} \,|e = e_{11}e_{12}...e_{1n} \tag{3.4}$$

$$\text{with } e_{1j} = e_{ij} \,|\, \forall i \in \{1, 2, ..., m\}, \forall j \in \{1, 2, ..., n\}$$

In the proposed UML representation, the use of the *loop* combined fragment that can be represented with the function *loop fragment* (lf)[10] ensures the correct execution order:

$$\text{lf}(e, p, q) := \text{lf}(e, p, q, 0) \tag{3.5}$$

$$\text{lf}(e, p, q, i) := \begin{cases} \{\epsilon\} & |i \geq q \\ \text{strict}(e, \text{lf}(e, p, q, i+1) & |\text{else} \end{cases} \tag{3.6}$$

where p is the minimum number of executions and q is the maximum number of executions. Since we define that $p = i + 1$ and $q = m$, lf results in the same execution sequence as *loop*, which is defined in Equation 3.4:

$$\Rightarrow \text{lf}(e, p, q) = \text{loop}(m, e) \tag{3.7}$$

The loop information is given inside the UML constraint of the combined fragment, noted with brackets, or in the *loop* object for the SD. For both representations, the syntax ensures that the requirements for application logic representation hold for the application logic contained in the loop and both numbers are defined to be naturals.

The **getter** and **setter functions** both represent whether the target is a property or variable and its name. The setter function additionally has to contain information about the value that the target will be set to, this can be an explicitly noted value or the reference to another variable or property. The **reference** command in the application logic is similarly structured, with the difference that only another application logic sequence, represented by an action or a function, is a valid target. All these requirements are ensured by the syntax of the representations.

The **conditional execution** cond (Equation 3.8), similar to an *if* statement in programming languages, defines the execution of one application logic sequence if a certain condition is true. Furthermore, it allows to optionally define a second application logic to be executed if the condition is not fulfilled, similar to an *else* statement:

$$\text{cond}(e, f) := e \cup f \cup \epsilon \tag{3.8}$$

$$\text{with } e \in C, f \in \{C \cup \{\epsilon\}\}$$

10 We omit the definition of the lf function for $p \geq i < q$, since we define $p = i + 1$.

In the Sequence Diagrams, the conditional execution is represented with the alternative combined fragment alt. This results in exactly one of the application logic lists being executed and can be defined with two sets X and Y:

$$\text{alt}(x, y) = \{x\} \cup \{y\} \tag{3.9}$$

$$\text{with } x \in X, y \in Y$$

In the SD, a mandatory content and optionally an else-content that are specified as application logic themselves represent the possible execution logic sequences.
For both representations, the condition that determines which application logic is to be executed is defined by the syntax. It can consist of a variable or property being a Boolean value or given in addition with a value or other variable/property to compare it with. Additionally, the terms *allOf*, *oneOf*, *anyOf* and *not* are defined in the representations and semantically equal the JSON Schema terms as defined in [20]. In the SD, they are specified as properties, containing further condition elements. In the diagrams, they are represented by the keyword followed by braces, which allow to nest them and contain the further condition expressions.
The **pause execution** command is defined by a natural number that specifies the time to pause in milliseconds. In UML Sequence Diagrams, a duration constraint with the same value for minimum and maximum duration represents this command. The SD JSON Schema ensures that a number equal or bigger one that is a multiple of 1.0 is used to specify the pause time.
The **Atomic Mashup**, which is introduced in the beginning of Section 3.1, refer to the application logic element that contains interactions with Things involved in the Mashup. To constrain this element to useful sequences and be able to represent asynchronous reception of data, we define the Atomic Mashup to consist of an unordered sequence of receive interactions followed by an unordered sequence of send interactions. The send interactions can be executed on reception of either the first receive interaction reply, or the last receive interaction reply.
The Atomic Mashups atom are defined by all invoked receiving interactions, all invoked sending interactions and the information whether the sending interactions should be executed on receiving the first or last pushed data. This information determines when to execute the sending interactions, based on the reception of asynchronous data pushes resulting from a subscription of an event or observation of a property in the receiving interactions.
The resulting execution sequence depends on the Boolean break definition b, which can

be true or false:

$$\operatorname{atom}(r,s,b) := \begin{cases} rs & \text{for} \quad b = \{t\} \\ \operatorname{pre}(r)s & \text{for} \quad b = \{f\} \end{cases} \tag{3.10}$$
$$\text{with } r \in R^n, s \in S^m, b \in \{t \cup f\}$$

where R is the set of receiving interactions and S the set of sending interactions. The resulting application logic sequence consists of n receiving and m sending interactions. The prefix function pre is defined recursively with the sets X and Y as:

$$\operatorname{pre}(z) := \{x(\operatorname{pre}(y))\} \cup \{x\} \tag{3.11}$$

$$\text{with } z = xy, x \in X, y \in Y$$

In the Sequence Diagrams, these atomic Mashups are represented by an enclosing strict combined fragment (Equation 3.3) and two parallel merge fragments par (Equation 3.13) that contain the receiving and sending interactions. Thus, one can show:

$$\begin{aligned} \operatorname{strict}(\operatorname{par}(r), \operatorname{par}(s)) &= \operatorname{par}(r)\operatorname{par}(s) \\ &= rs \end{aligned} \tag{3.12}$$

$$\begin{aligned} \operatorname{par}(x) &:= (x_1 \sqcup\!\sqcup \operatorname{par}(x_2, ..., x_v)) \\ &= x \end{aligned} \tag{3.13}$$

where $x \in X^v, r \in R^n, s \in S^m$, n number of receiving interactions, m number of sending interactions and the shuffling operator $\sqcup\!\sqcup$ is defined according to [21] as:

$$\begin{aligned} &(ax \sqcup\!\sqcup by) := a(x \sqcup\!\sqcup by) \cup b(ax \sqcup\!\sqcup y) \\ &x \in X, y \in Y, a \in A, b \in B \end{aligned} \tag{3.14}$$

$$\Rightarrow x \sqcup\!\sqcup y = \{xy\} \cup \{yx\} \,|x \in X, y \in Y \tag{3.15}$$

One can use Equation 3.15 since every parameter of the shuffle function is only one word. In the equations, A, B, X and Y are defined as sets. The information to send interactions on the first data push reception is represented by adding the break fragment brk to the parallel merge fragment that contains the receiving interactions. Thus, with the previous definitions we can define the resulting execution order of the entire Atomic Mashup for this case as:

$$\begin{aligned} \operatorname{brk}(\operatorname{par}(r), \operatorname{par}(s)) &= \operatorname{strict}(\operatorname{pre}(\operatorname{par}(r)), \operatorname{par}(s)) \\ &= \operatorname{pre}(r)s \end{aligned} \tag{3.16}$$

where $r \in R^n, s \in S^m$, n number of receiving interactions, m number of sending interactions and the break fragment:

$$\text{brk}(x, y) = \text{strict}(\text{pre}(x), y) \tag{3.17}$$

$$\text{with } x \in X^v, y \in Y^w$$

In the SD, the property *breakOnDataPushed* defines whether to send on the first or last receive of asynchronously pushed data. The receiving and sending interactions are both defined as JSON Arrays and with context `"@container":"set"` specifying them as unordered.

Equivalence Reasoning

By considering that Sequence Diagrams and SDs are generally equally structured and can contain the same elements in the same order (Section 3.4.2), together with the equality in representing all of these elements (Section 3.4.2), one can conclude their equality. Thus, if M is defined as the set of possible Mashups, every $m \in M$ can be represented with an SD or Sequence Diagram:

$$\text{SystemDescription}(m) \Leftrightarrow \text{SequenceDiagram}(m) \quad \square \tag{3.18}$$

This concludes our proof that the WoT Sequence Diagram and the WoT System Description can represent Mashups equally, which ensures that their conversion into each other does not cause a loss of information.

3.5 Algorithms

Based on the proof of equality presented in Section 3.4, we present algorithms to convert the representations into each other and generate code from an SD.
We choose the SD as input for the code generation, because it is a valid TD describing the network-facing interfaces of the Mashup, which is required for creating a new Exposed Thing following the Scripting API standard. The Mashup that would be consumed by other Things would not require the SD logic and also the other Things might be more resource constrained. Thus, when exposing the Mashup's TD, the algorithm removes SD-specific vocabulary.

3.5.1 Representation conversions

The algorithms we present to convert the representations into each other are shown as an example in Algorithm 1, as they are both similarly structured.
The algorithm first parses the Sequence Diagram input (line 1), by calling a function that retrieves an internal representation of the Mashup logic, for every included diagram

Algorithm 1 An algorithm to convert a System Description to a Sequence Diagram by retrieving the application logic.

```
 1: procedure parseSequenceDiagram
 2:     clean inputPlantUmlNotation;
 3:     for diagram $\in$ inputPlantUmlNotation do
 4:         for line $\in$ diagram do
 5:             mashupLogic $\leftarrow$ uml_to_internal(line);
 6: procedure generateSystemDescription
 7:     SystemDescription $\leftarrow$ generate_SD_Template();
 8:     for logicArray $\in$ mashupLogic do
 9:         for element $\in$ logicArray do
10:             if element[form] not in AddedForms then
11:                 SystemDescription $\leftarrow$ element[form]
12:                 AddedForms $\leftarrow$ element[form]
13:             SystemDescription $\leftarrow$ add_SD_Path_Element(element)
```

Algorithm 2 Code generation algorithm that generates two code instances, one implementing a Mashup's application logic and another one to include required protocol bindings.

```
 1: procedure generateIndex
 2:     for protocol $\in$ (confExposeProtocol $\lor$ SdInteractionForms) do
 3:         index $\leftarrow$ include_WoT_API_protocol_binding(protocol);
 4: mashupLogic $\leftarrow$ parse_System_Description(inputSD);
 5: code $\leftarrow$ add_variables_handlers_and_classConstructor(inputSD);
 6: procedure generateMashupCode
 7:     while mashupLogic $! =$ null do
 8:         code $\leftarrow$ generate_code_for_logic(mashupLogic[0])
 9:         if mashupLogic[0] has logicContent then
10:             generateMashupCode(logicContent)
11:         mashupLogic.removeIndex(0)
```

line-wise (lines 3–4). Based on the retrieved Mashup application logic and a Mashup template (line 7), it generates the SD by adding the SD equivalents (line 13) and required forms (lines 10–12) for every application logic element in every application logic sequence (lines 8–9).

3.5.2 Automatic code generation

The automatic code generation presented in Algorithm 2 integrates all bindings for protocols required to interact with a Thing involved in the Mashup or configured to expose the Mashup's interfaces (lines 1–3). Furthermore, it generates executable code for every application logic element, represented by an internal tree-like structure, of the Mashup with the recursive executable procedure *GenerateMashupCode* (lines 6–11).

3.5.3 Implementation

All algorithms we present can be implemented in any Turing-complete programming language, but our publicly available[11] implementation is based on Node.js, JavaScript, TypeScript and the WoT Scripting API standard reference implementation.

3.6 Discussion

From the listed Mashup equations in Section 3.4, we reason that one can define the representations we propose, to be equal (Equation 3.18) for the context of representing systems of IoT devices. With the proposal of the representations and algorithms for WoT Mashups, we achieve to provide automatic code generation in combination with Mashup insights for the WoT. We also provide simple creation of Interaction Affordances for Mashups, as they are automatically generated. In addition, the SD representation simplifies the transmission and storage of Mashups and the Sequence Diagram representation simplifies the documentation and manual creation of Mashups.

11 The implementation is part of our repository.

4 Related Work

We are mentioning the WoT Scripting API standard as enabler for simpler development by providing an abstraction on an *interaction layer*. However, it is important to mention that there have been comparable approaches before, e.g., the Mozilla Web Thing API[1] or the Philips Hue API, but no open standard.

To conclude semantic traces from the UML Sequence Diagram specification elements, we follow the definitions in [19] as mentioned in Section 3.4.1. This is an important building block for our proof of equality. In the referenced work, it is shown how the UML specification can be formalized, starting with basic elements, such as *event occurrences* until rather complex to model elements, such as the parallel merge fragment that requires the introduction of an shuffle operator (Equations 3.13–3.15). Although, it is not modeling the complete UML specification, the formalism for every element used in WoT Sequence Diagrams is defined.

An approach for using MDE to represent the *consumedThing* interface, referring to the WoT Scripting API standard, of a single Thing is presented in [22]. The authors present a multi-layered model built in the Eclipse Modeling Framework [6], which allows automatic code generation for the described network-facing interfaces but comes with the disadvantage of tool-specificity.

The WoT-based Asset Description concept, which also uses a JSON-LD instance to represent a WoT Mashup, presented in [23] is partly similar to our SD proposal, but it is focused on automatic discovery and composition of simple systems. Therefore, a mechanism to discover and update available Things together with a graphical user interface for system composition is presented, but the created systems are restricted in their functionality. The Mashups described can only contain actions of Things or events triggered based on one condition, or automatically calculated property values. The systems cannot include application logic, i.e., loops and also cannot contain sequences

1 https://iot.mozilla.org/wot/

of interactions.
In [24] the WoTify platform for sharing WoT-conform implementations and TDs of IoT devices is presented and in [25] a similar platform, called WoT Store, for sharing WoT Mashups is proposed. For both approaches, our contributions would add value, where one can share Mashups represented with an SD and enable user insights with Sequence Diagrams. The code generation algorithm we proposed in Section 3.5.2 could generate code using an implementation fitting the user requirements not only for the WoT Scripting API reference implementation in Node.js, but also for example for implementations in other programming languages such as for Python with [26].
The WoT semantic functionality distance presented in [27], is a measure for the ability of two Things to fulfill the same functionality in a specific context, such as a Mashup scenario. It can help to identify which devices can collaborate or can be replaced by each other. This value could be computed between all Things of a Mashup and stored as property of the *Things* keyword child elements in the System Description. Storing the value in a Mashup representation would make the replacement of malfunctioned devices very simple for the Mashup controller since it could interact with the semantically closest other Thing in the Mashup instead, without having to be able to compute the value.
The aforementioned scenario demonstrates as an example that the annotation of Mashup representations, which is possible with the System Description we propose in this thesis, can improve the Mashup management by integrating other research approaches effectively. Other information about involved Things that could make sense to be stored in a Mashup representation are, e.g., Timing behavior, Security information or physical relations.

5 Evaluation

In order to evaluate our contribution and since there exists no state-of-the-art approach which we could use for a meaningful comparison, we present three case studies with different characteristics.

5.1 Common Device Setup

All Mashups consist of physical Things, which can be implemented with the information in our repository and are connected over a local network. The Mashup controller is hosted on a conventional laptop that is in the same local network. The whole setup is illustrated in Figure 7, which also shows a Consumer that is not required for every use case.

5.2 Evaluation Procedure

We evaluate whether the Mashup can be represented with the proposed representations and the conversion algorithms work as expected. The steps we follow each scenario are:

- Manually creating a Sequence Diagram representation.
- Automatically converting the Sequence Diagram into an SD.
- Automatically converting the resulting SD back to a diagram.
- Comparing the initial Sequence Diagram to the automatically generated one in the previous step.

Furthermore, we evaluate the automatic code generation algorithm by creating code from the generated SD and executing it on a Mashup controller. During the execution we check whether the physical interactions observable on the Things, which are triggered via

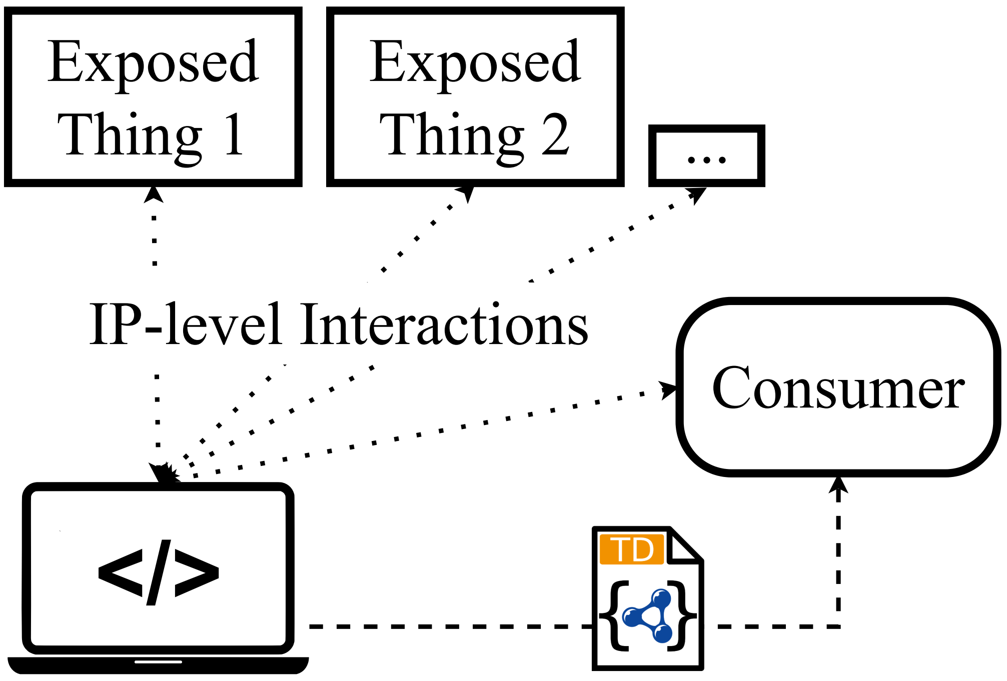

Figure 7: Abstract illustration of the common device setup we use to evaluate our approach, where the Mashup controller is hosted by a laptop. Depending on the case study, there is a different number of Exposed Things involved, which is listed in Table 5.1, and a consumer is required or not.

requests of the Mashup controller, follow the application logic described in the Sequence Diagram.

5.3 Case Studies

The case studies we use to evaluate our approach are characterized by the following scenarios and requirements:

- **Case Study 1: Simple Scenario**
  A Mashup involving an LED-strip and a push button, where the LEDs should be turned on for ten seconds on every push of the button. Here, the Mashup does not require a Consumer.
- **Case Study 2: W3C Reference Smart Home Scenario**
  A smart home scenario as introduced in the Second W3C workshop on the WoT[1]. The setup involves an LED-strip and three ZigBee lamps with a ZigBee gateway attached to the local network. The Mashup controller exposes the two actions, named *coming home* and *leaving*, that hierarchically turn all lights on or off. In this scenario, the Consumer of the Mashup is a Mozilla WebThings Gateway[2],

1 https://github.com/w3c/wot/tree/master/workshop/ws2/Demos
2 https://iot.mozilla.org/gateway/

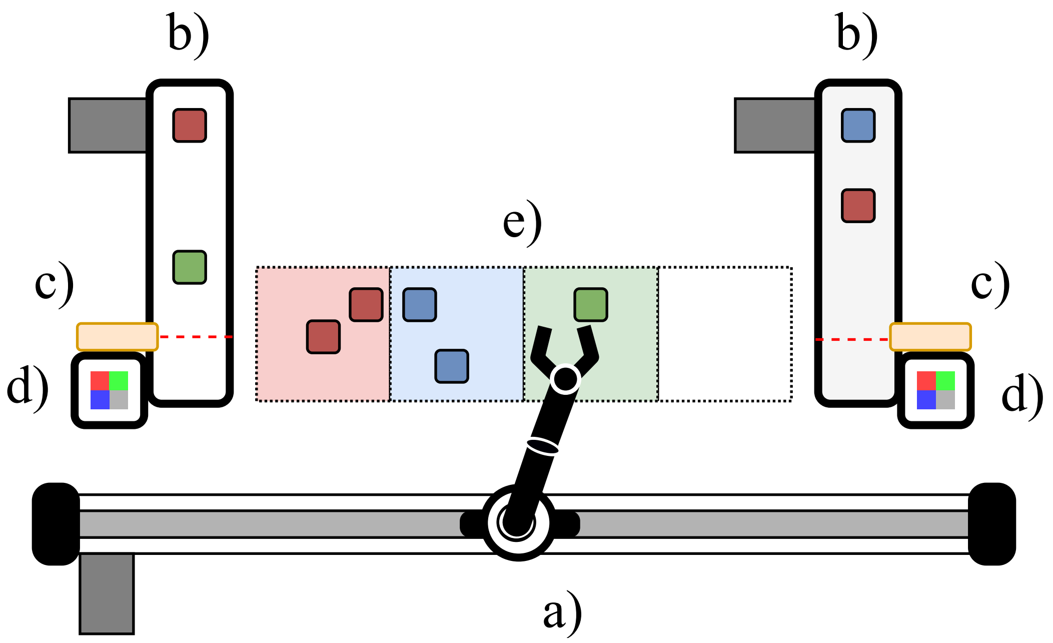

Figure 8: Top view of industrial scenario's setup, with: Robot arm on an actuated slider (a), conveyor belts (b) with objects on them, infrared sensors (c), color sensors (d) and an area to drop the items divided by color (e).

which is connected to the same local network and could integrate services such as voice assistants.

- **Case Study 3: Industrial Scenario**
  An industrial environment scenario involving two conveyor belts, a robot arm on an actuated slider that is able to pick objects from both conveyor belts, two infrared sensors mounted on the belts and two color sensors. The hardware setup is illustrated in Figure 8 and the application logic can be explained as:

  1. Both conveyor belts are started.
  2. On detection of an object by one of the infrared sensors, the corresponding belt is stopped.
  3. The robot arm grabs the object of the stopped belt and presents it to the color sensor.
  4. The robot arm moves and drops the object to a position depending on the color value read by the color sensor.
  5. The conveyor belt is started again. If the other conveyor belt's sensor had detected an object during the execution of the steps 2–5, it is stopped immediately and the steps 3–5 are immediately executed again.
  6. The application logic continues with step 2.

  In this case study, the Mashup does not require a Consumer.

Case Study	Number of devices	Generated lines of code [4]	Application logic	Consumer
1	2	154	8	no
2	4	156	2	yes
3	7	365	33	no

Table 5.1: Metrics characterizing the evaluation results of the different case study scenarios.

5.4 Evaluation Results

We evaluated the approach with the aforementioned setup and procedure for each of the three case studies and listed the resulting scenario metrics in Table 5.1. The application logic column in the table lists the number of application logic elements, e.g., loop, Atomic Mashup or pause command. Furthermore, we present an extract of the manually created Sequence Diagram input and the generated JavaScript code, TypeScript code and SDs in the Appendix (A.5, A.6, A.7 and A.8). We found that it is possible to represent the Mashup of each case study with both representations and convert them into each other, which shows that the Equation 3.18 holds. More specifically, the conversion algorithms generate PlantUML notations that are equal to the input with the exception of design choices inside the notation such as spaces, empty lines or the order of unordered contents. Furthermore, the code generation algorithm produces working Mashup logic implementations and no manual adaptions are necessary to achieve the expected Thing interaction behavior. Nevertheless, the generated code is still readable and could be changed by humans. All resources of the evaluation can be accessed[3] by the reader, to enable him/her to inspect, retrace or reproduce our evaluation results.

3 Our repository includes the complete evaluation inputs, i.e., manually written Sequence Diagrams and the results, i.e., automatically generated SDs and code, since putting them in the paper would require more pages.

4 The SD and TD objects and protocol binding inclusions are excluded from the lines of code count while empty lines are included, since the generated code is intended to be human-readable.

6 Conclusion

We have proposed two representations for Mashups in the Web of Things (WoT), namely WoT Sequence Diagram and WoT System Description. These representation formats have unique advantages respectively. The Sequence Diagram is especially giving an insight into a system's application logic, which can be better understood than for example program code, also for complex logic. The System Description in comparison provides a machine-understandable open format for saving and transmitting Mashup representations. By stating their syntactical and semantical foundations, we have shown how both representations are defined and how they can be validated using the resources we present. By providing code generation and conversion algorithms, we have demonstrated in three case studies that the representations and the management of Mashups in the WoT are improved, by combining the advantages of both representations and introducing automatic code generation for Mashup application logic. Our contribution thus establishes a groundwork for further improvements in managing WoT Mashups in a systematic way.

6.1 Outlook

To improve the creation process of the WoT Mashups described in this work, a graphical user interface can simplify the manual creation of Sequence Diagrams and System Descriptions. This could be taken another step further by using design space exploration techniques to simplify the user interactions by ordering and restricting the possible options according to the potential benefits that these Mashups could provide.
A future extension of the WoT System Description presented in this work, could be the emission of Mashup specific events. This can integrate Mashup application logic in the concept of subscription and notification. Thus, it could be for example possible to emit an event called *security alert* if two motion sensors involved in the same Mashup

recognize a movement in a short period of time, in order to make the alert more failure resistant. The emission of events can be integrated as own application logic element.

A

Appendix

A.1 System Description TypeScript Definitions

```typescript
type pathEl = pathWait | pathLoop | pathCase | pathInteract | pathSet |
    ↪ pathGet | pathRef

interface pathWait {
    wait: number
}

interface pathLoop {
    loop: {
        type: "logical" | "timed",
        defaultInput: number | true,
        path: pathEl[]
    }
}

interface pathCase {
    case: {
        if: ifWord,
        then: {
            path: pathEl[]
        },
        else: {
            path?: pathEl[]
        }
    }
}
type ofWord = {allOf: ifWord[]} | {oneOf: ifWord[]} | {anyOf: ifWord[]}
type ifWord = ofWord |
                {not: ifWord} |
                {get: { $ref: string}, output?: typeOutput}
```

```

type pathInteract = {
    receive: pathInteractReceive[],
    send: pathInteractSend[],
    breakOnDataPushed?: boolean
}

type pathInteractReceive = {
        form: {
            $ref: string
        },
        set?: {$ref: string},
        op: "subscribeevent" | "observeproperty" | "readproperty" | "
    ↪ invokeaction"
}

type pathInteractSend = {
        form: {
            $ref: string
        },
        get?: {$ref: string},
        defaultInput?: any,
        op: "writeproperty"| "invokeaction"
}

type pathRef = {
    $ref: string
}

type pathSet = {
    set: {$ref: string},
    defaultInput?: typeDefaultInput,
    get?: {$ref: string}
}

type pathGet = {
    get: {$ref: string}
}

type typeDefaultInput = boolean | number | string
type typeOutput = number | string | {$ref: string}
```

Listing A.1: Extract of the System Description TypeScript definitions. It defines the structure of the *pathEl* element, which represents a single application logic element.

A.2 PlantUML Notation - Interaction Sequences

```
@startuml templateSubscribe
[->"Agent": top:templateSubscribe()
activate "Agent"

group strict
    par
        "Agent" -> "Thing" : readProperty: "state"
        activate "Thing"
        "Thing" --> "Agent" : response
        deactivate "Thing"
    else
        "Agent" -> "Thing" : invokeAction: "int-string"
        activate "Thing"
        "Thing" --> "Agent" : output
        deactivate "Thing"
    else
        "Agent" -> "Thing" : subscribeEvent: "maintenance"
        activate "Thing"
        "Thing" --> "Agent" : confirmation
        "Thing" ->> "Agent" : data-pushed
    else
        "Agent" -> "Thing" : observeProperty: "state"
        activate "Thing"
        "Thing" --> "Agent" : confirmation
        "Thing" ->> "Agent" : data-pushed
    end
    par
        "Agent" -> "Thing" : writeProperty: "string"
    else
        "Agent" -> "Thing" : invokeAction: "stopBelt"
    end
end

[<-"Agent"
deactivate "Agent"
@enduml
```

Listing A.2: PlantUML notation which lists every possible receiving operation, i.e., *readProperty*, *invokeAction* with output, *observeProperty* and *subscribeEvent*, followed by every possible sending operation, i.e., *invokeAction* and *writeProperty*, as one interaction sequence.

A.3 PlantUML Notation - Possible Elements

```
@startuml used-elements-overview
[->"Agent":top:motionMashup()
activate "Agent"
note over "Agent"
  defaultInput true
  set Condition1
end note
group strict
  par
    "Agent" -> "MotionSensor1" : subscribeEvent: "motionDetected"
    activate "MotionSensor1"
    "MotionSensor1" --> "Agent" : confirmation
    "MotionSensor1" ->> "Agent" : data-pushed
  else
    "Agent" -> "MotionSensor2" : subscribeEvent: "motionDetected"
    activate "MotionSensor2"
    "MotionSensor2" --> "Agent" : confirmation
    "MotionSensor2" ->> "Agent" : data-pushed
  else
    break data-pushed
    end
  end
  par
    "Agent" -> "Lightbulb1" : writeProperty: "brightness"
  else
    "Agent" -> "Lightbulb2" : writeProperty: "brightness"
  end
end
loop Condition1
  ... {20000ms..20000ms} ...
  alt Condition2 == "bulb1"
    group strict
        "Agent" -> "MotionSensor1" : readProperty: "motionShortBefore"
        activate "MotionSensor1"
        "MotionSensor1" --> "Agent" : response
        deactivate "MotionSensor1"
        "Agent" -> "Lightbulb1" : writeProperty: "brightness"
    end
  else else
    group strict
        "Agent" -> "MotionSensor2" : readProperty: "motionShortBefore"
        activate "MotionSensor2"
        "MotionSensor2" --> "Agent" : response
        deactivate "MotionSensor2"
        "Agent" -> "Lightbulb2" : writeProperty: "brightness"
    end
  end
end
[<-Agent
deactivate "Agent"
@enduml
```

Listing A.3: PlantUML notation instance that defines every graphical UML element used in WoT Sequence Diagrams at least once, which is presented in Figure 6.

A.4 System Description - Keywords

Keyword	Usage
loop	Loop element of a path array.
defaultInput	Input to setter functions, loop count/period, case comparison or default value of variables.
$ref	Reference to a variable, property, function or action. Is chosen because of the known meaning in JavaScript Object Notation (JSON) Schema.
wait	Pause execution command of a path array.
interact	Atomic Mashup element of a path array, contains *send*, *receive* and *breakOnDataPushed*.
send	Array of sending interactions, part of an Atomic Mashup.
receive	Array of receiving interactions, part of an Atomic Mashup.
breakOn-DataPushed	Defines when to execute sending interactions in an Atomic Mashup.
set	Setter function in a path array or as part of a receiving interaction.
get	Getter function in a path array or as part of another element which requires a variable as input, e.g., *send*, *loop*, *case*.
case	Conditional execution (if statement) parent keyword, element of a path array. Contains *if*, *else* (optionally) and *then*.
if	Contains the condition of a *case* element.
then	Contains the application logic to be executed if the condition of a *case* is true.
else	Contains the application logic to be executed if the condition of a *case* is false.
oneOf, allOf, anyOf	Can be used as child elements of *if* and have to contain arrays of further conditions, which are computed with the respective Boolean operator.
not	Can be used as child element of *if* and has to contain another condition, which is inverted.
sync	Optionally determines whether a loop should be executed synchronously or asynchronously.
type	Type of a loop, can have the value *timed* or *logical*.
isUpdated-OnDemand	Can optionally be used to determine when the value of a path, which is child element of a Mashup's property, is computed. The computation can happen periodically or only on request.
defaultOutput	Can optionally be used to determine the output of an action if it has no other value to return.

Table A.1: All keywords, with exception of the most important ones that are explained in Section 3.3, of the System Description (SD) enhancement in comparison to the Thing Description (TD).

A.5 Use Case Scenario 1 - Code Extract

```
private func_SubsPress() {
    return new Promise < any > (async (resolve, reject) => {
        // ### path: ###
        // -- interaction sequence --
        this.consumed_things["SenseHat"].subscribeEvent("joystickPress"
    ↪ , async autoGenReceive5 => {
            this.maproisLightOn = autoGenReceive5
            this.data_pushes[0][0] = true
            if (Object.keys(this.data_pushes[0]).every(el => {
                    return (this.data_pushes[0][el] === true)
                })) {
                console.log("data push allOf: " + autoGenReceive5)
                const autoGenWrite3 = this.consumed_things["
    ↪ DotStarRGBLEDstrip"].invokeAction("random")
                await autoGenWrite3
            }

              if (Object.keys(this.data_pushes[0]).every(el => {
                    return (this.data_pushes[0][el] === true)
                })) {
                Object.keys(this.data_pushes[0]).forEach(el => {
                    this.data_pushes[0][el] = false
                })
            }
        })

        // -- end intrct seq --

        // ### end path ###
        resolve()
    })
}
```

Listing A.4: The function *SubsPress*, taken from the generated TypeScript code of the first use case scenario (simple use case).

A.6 Use Case Scenario 1 - Index

```javascript
WotMashup = require("./ma1").WotMashup

const TD_DIRECTORY = ""

Servient = require("@node-wot/core").Servient

HttpServer = require("@node-wot/binding-http").HttpServer


HttpClientFactory = require("@node-wot/binding-http").HttpClientFactory


const httpServer = new HttpServer({port: 8080})


const servient = new Servient()

servient.addServer(httpServer)


servient.addClientFactory(new HttpClientFactory())


servient.start().then( WoT => {
    wotMashup = new WotMashup(WoT, TD_DIRECTORY) // you can change the wotDevice (wotMashup) to something that makes more sense
})
```

Listing A.5: Index.js file of the generated code of the first use case scenario (simple use case), which handles the required protocol binding imports.

A.7 Use Case Scenario 1 - Sequence Diagrams

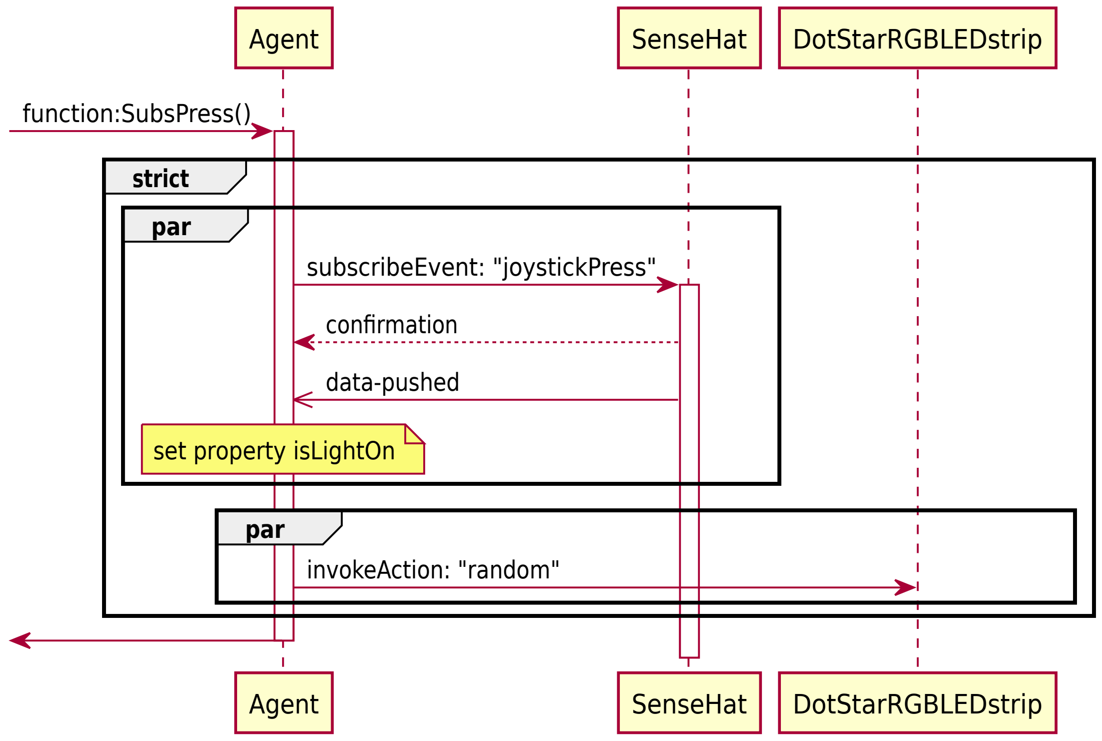

Figure 9: Sequence Diagram presentation of the manually created function *subsPress* of the application logic for the Mashup that implements the first use case scenario. The generated code for this function is shown in Listing A.5

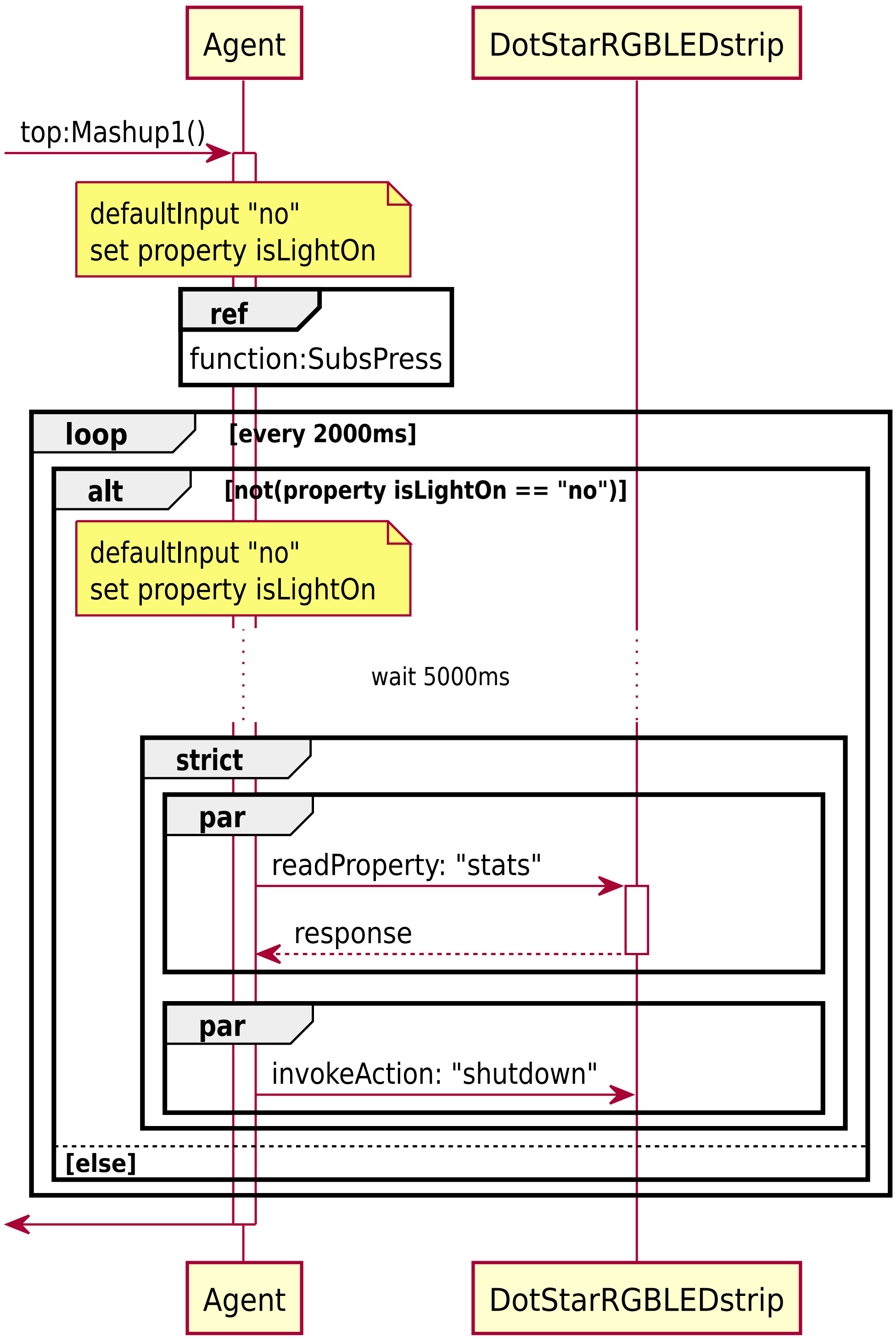

Figure 10: Sequence Diagram presentation of the application logic which is immediately executed on code execution of the Mashup that implements the first use case scenario.

A.8 Use Case Scenario 1 - System Description

```json
"functions": {
  "SubsPress": {
    "forms": [
      {
        "href": "example://functions/SubsPress"
      }
    ],
    "path": [
      {
        "receive": [
          {
            "form": {
              "$ref": "#SenseHat/events/joystickPress/forms/0"
            },
            "op": "subscribeevent",
            "set": {
              "$ref": "#/properties/isLightOn"
            }
          }
        ],
        "send": [
          {
            "form": {
              "$ref": "#DotStarRGBLEDstrip/actions/random/forms/0"
            },
            "op": "invokeaction"
          }
        ],
        "breakOnDataPushed": false
      }
    ]
  }
}
```

Listing A.6: Extract of the SD generated in the use case scenario 1 of the Evaluation. The extract shows the *functions* key and its content, which is only one function named *SubsPress*. This function is implemented with the TypeScript code in Listing A.5 and its application logic is presented by the Sequence Diagram in Figure 9.

A.9 Sequence Diagram Grammar

```
mashup ::= diagram+
diagram ::= header content footer
header ::= ’@startuml’ S Title L ’[’ ’->’ ’"Agent"’ ’:’ S Diatype ’:’
    ↪ Title ’()’ L ’activate’ S ’"Agent"’ L
footer ::= ’[<-’ ’"Agent"’ L ’deactivate’ S ’"Agent"’ L ’@enduml’ L?
content ::= (loop
           | wait
           | condition
           | interaction
           | getset
           | ref)+
loop ::= ’loop’ S ( ’every’ S Nr ’ms’ | ’forever’ | Nr ’x’ ) L content
    ↪ ’end’ L
wait ::= ’...’ S ’wait’ S Nr ’ms’ S ’...’ L
condition ::= ’alt’ S comparison L content ’else else’ L content? ’end’
    ↪  L
comparison ::= (’not(’ S? comparison S? ’)’)
             | (’allOf(’ S? comparison (S? ’,’ S? comparison)* S? ’)’)
             | (’oneOf(’ S? comparison (S? ’,’ S? comparison)* S? ’)’)
             | (’anyOf(’ S? comparison (S? ’,’ S? comparison)* S? ’)’)
             | ((’variable’ | ’property’) S VarName (S ’==’ S ( ’true’
    ↪ | ’false’ | ’"’ Char* ’"’ | Nr | (’variable’ | ’property’) S
    ↪ VarName ))? )
interaction ::= ’group’ S ’strict’ L interactionRecCont
    ↪ interactionSendCont ’end’ L
interactionRecCont ::= ’par’ L (interactionReceive (’else’ L)?)* (’
    ↪ break’ S ’data-pushed’ L ’end’ L)? ’end’ L
interactionSendCont ::= ’par’ L (interactionSend (’else’ L)?)* ’end’ L
interactionReceive ::= getset? interactionPre ( receiveSubs |
    ↪ receiveInv | receiveObs | receiveRead ) getset?
interactionSend ::= getset? interactionPre ( sendWrite | sendInv )
    ↪ sendPost getset?
interactionPre ::= ’"Agent"’ S ’->’ S interactionTo S ’:’ S
receiveRead ::= ’readProperty:’ receiveMiddle readResponse L deactTo L
receiveSubs ::= ’subscribeEvent:’ receiveSubsObsPost
receiveObs ::= ’observeProperty:’ receiveSubsObsPost
receiveInv ::= ’invokeAction:’ receiveMiddle invConfirmation L deactTo
    ↪ L
receiveSubsObsPost ::= receiveMiddle subsObsConfirmation L subsObsData
    ↪ L
receiveMiddle ::= S interactionName L actTo L
sendWrite ::= ’writeProperty:’
sendInv ::= ’invokeAction:’
sendPost ::= S interactionName L
readResponse ::= interactionTo S ’-->’ S ’"Agent"’ S ’:’ S ’response’
subsObsConfirmation ::= interactionTo S '-->' S '"Agent"' S ':' S 'confirmation'
subsObsData ::= interactionTo S '->' '>' S '"Agent"' S ':' S 'data-pushed'
invConfirmation ::= interactionTo S '-' '->' S '"Agent"' S ':' S 'output'
actTo ::= 'activate' S interactionTo
deactTo ::= 'deactivate' S interactionTo
interactionTo ::=  '"' Ntitle '"'
interactionName ::= '"' Ntitle '"'
safetitle ::= '"' Title '"'
getset ::= 'note' S 'over' S ('"')? 'Agent' ('"')? L
           (((('get'|'set') S ('variable' | 'property') S VarName ) | ('defaultInput' S ('true' | 'false' | '"' Char* '"' | Nr | '{' Nchar* '}' ))) L)+
           'end' S 'note' L
ref ::= 'ref' S 'over' S ('"')? 'Agent' ('"')? L
        ('function'|'action') ':' VarName L
        'end' S 'ref' L

<?TOKENS?>

L ::= S? (#x000A
    | #x000D #x000A?)+ S?
S          ::= [#x0020#x0009]+
Title      ::= [a-zA-Z] [a-zA-Z0-9]+
Ntitle ::= [a-zA-Z] ([a-zA-Z0-9] | '-' | '_')+
Nchar ::= (Char | '"') - '}'
Diatype   ::= ('top'|'action'|'property'|'function')
VarName   ::= ([a-zA-Z] [a-zA-Z0-9]*) - ('false' | 'true')
Char      ::= [https://www.w3.org/TR/REC-xml/#NT-Char] - '"'
Nr        ::= [0-9]*
```

Listing A.7: The whole EBNF grammar that defines the valid language to note a WoT Sequence Diagram.

Bibliography

[1] Google Ireland Limited, "Google Trends — Query: IoT, from 2010-02-05 to 2020-02-05," 2020. Available: https://trends.google.com/trends/explore?date=2010-02-05%202020-02-05&q=IoT (Accessed 2020-02-05). cited on p. 1

[2] Gartner UK Limited, "Gartner Research Hype Cycle for Emerging Technologies, 2018," 2018. Available: https://www.gartner.com/en/documents/3885468/hype-cycle-for-emerging-technologies-2018 (Accessed 2020-02-09). cited on p. 1

[3] D. Guinard and V. Trifa, "Towards the Web of Things: Web Mashups for Embedded Devices," in *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences), Madrid, Spain*, vol. 15, p. 8, 2009. cited on p. 1, 5, 8

[4] M. Kovatsch, R. Matsukura, M. Lagally, T. Kawaguchi, K. Toumura, and K. Kajimoto, "Web of Things (WoT) Architecture - W3C Candidate Recommendation 30 January 2020," 2020. Available: https://www.w3.org/TR/2020/PR-wot-architecture-20200130/ (Accessed 2020-02-09). cited on p. 1, 5

[5] S. Käbisch, T. Kamiya, M. McCool, V. Charpenay, and M. Kovatsch, "Web of Things (WoT) Thing Description - W3C Proposed Recommendation 30 January 2020," 2020. Available: https://www.w3.org/TR/2020/PR-wot-thing-description-20200130/ (Accessed 2020-02-09). cited on p. 1

[6] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse Modeling Framework*. Pearson Education, 2008. Further information: https://www.eclipse.org/modeling/emf/ (Accessed 2020-04-01). cited on p. 3, 29

[7] "Abstract Architecture of W3C WoT." Available: https://w3c.github.io/wot-architecture/images/architecture/overview.svg (Accessed 2020-04-01). cited on p. 5

[8] M. Koster and E. Korkan, "Web of Things (WoT) Binding Templates - W3C Working Group Note 30 January 2020," 2020. Available: https://www.w3.org/TR/2020/NOTE-wot-binding-templates-20200130/ (Accessed 2020-03-09). cited on p. 7

[9] Z. Kis, D. Peintner, J. Hund, and K. Nimura, "Web of Things (WoT) Scripting API - W3C Working Draft 28 October 2019," 2019. Available: https://www.w3.org/TR/2019/WD-wot-scripting-api-20191028/ (Accessed 2020-02-05). cited on p. 7

[10] D. Guinard, V. Trifa, T. Pham, and O. Liechti, "Towards Physical Mashups in the Web of Things," in *2009 Sixth International Conference on Networked Sensing Systems (INSS)*, pp. 1–4, IEEE, 2009. cited on p. 8

[11] A. Roques, "PlantUML: Open-Source Tool that Uses Simple Textual Descriptions to Draw Uml Diagrams," 2015. Available: https://plantuml.com/ (Accessed 2020-03-24). cited on p. 11

[12] D. D. McCracken and E. D. Reilly, "Backus-Naur Form (BNF)," in *Encyclopedia of Computer Science*, pp. 129–131, John Wiley and Sons Ltd., 2003. cited on p. 11

[13] J. Robie, M. Dyck, and J. Spiegel, "XQuery 3.1: An XML Query Language. W3C Recommendation 21 March 2017," 2017. Available: https://www.w3.org/TR/2017/REC-xquery-31-20170321/ (Accessed 2020-03-16). cited on p. 11

[14] S. Cook, C. Bock, P. Rivett, T. Rutt, E. Seidewitz, B. Selic, and D. Tolbert, "Unified Modeling Language (UML) Version 2.5.1," standard, Object Management Group (OMG), 2017. Available: https://www.omg.org/spec/UML/2.5.1/ (Accessed 2020-03-16). cited on p. 13

[15] E. Korkan, S. Käbisch, M. Kovatsch, and S. Steinhorst, "Safe Interoperability for Web of Things Devices and Systems," in *Languages, Design Methods, and Tools for Electronic System Design*, pp. 47–69, Springer, 2020. cited on p. 17

[16] A. Wright and H. Andrews, "JSON Schema: A media Type for Describing JSON Documents," in *IETF, Internet-Draft draft-handrews-json-schema-O1*, 2018. cited on p. 18

[17] M. Sporny, D. Longley, G. Kellog, M. Landthaler, P.-A. Champin, and N. Lindström, "JSON-LD 1.1 A JSON-based Serialization for Linked Data - W3C Candidate Recommendation 05 March 2020," 2020. Available: https://www.w3.org/TR/2020/CR-json-ld11-20200305/ (Accessed 2020-03-10). cited on p. 18

[18] V. Charpenay, S. Käbisch, and H. Kosch, "Introducing Thing Descriptions and Interactions: An Ontology for the Web of Things.," in *SR+ SWIT@ ISWC*, pp. 55–66, 2016. cited on p. 19

[19] H. Störrle, "Semantics of Interactions in UML 2.0," in *IEEE Symposium on Human Centric Computing Languages and Environments, 2003. Proceedings. 2003*, pp. 129–136, IEEE, 2003. cited on p. 20, 21, 29

[20] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of JSON Schema," in *Proc. of the 25th International Conference on World Wide Web*, pp. 263–273, 2016. cited on p. 24

[21] A. Mateescu, G. Rozenberg, and A. Salomaa, "Shuffle on trajectories: Syntactic constraints," *Theoretical Computer Science*, vol. 197, no. 1-2, pp. 1–56, 1998. cited on p. 25

[22] M. Iglesias-Urkia, A. Gómez, D. Casado-Mansilla, and A. Urbieta, "Enabling easy Web of Things compatible device generation using a Model-Driven Engineering approach," in *Proc. of the 9th International Conference on the Internet of Things*, pp. 1–8, 2019. cited on p. 29

[23] K.-H. Le, S. K. Datta, C. Bonnet, and F. Hamon, "WoT-AD: A Descriptive Language for Group of Things in Massive IoT," in *2019 IEEE 5th World Forum on Internet of Things (WF-IoT)*, pp. 257–262, IEEE, 2019. cited on p. 29

[24] E. Korkan, H. B. Hassine, V. E. Schlott, S. Käbisch, and S. Steinhorst, "WoTify: A platform to bring Web of Things to your devices," *arXiv preprint arXiv:1909.03296*, 2019. cited on p. 30

[25] L. Sciullo, C. Aguzzi, M. Di Felice, and T. S. Cinotti, "WoT Store: Enabling Things and Applications Discovery for the W3C Web of Things," in *2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, pp. 1–8, IEEE, 2019. cited on p. 30

[26] A. G. Mangas and F. J. S. Alonso, "WOTPY: A framework for web of things applications," *Computer Communications*, vol. 147, pp. 235–251, 2019. cited on p. 30

[27] M. I. Robles, B. Silverajan, and N. C. Narendra, "Web of Things Semantic Functionality Distance," *2019 26th International Conference on Telecommunications, ICT 2019*, pp. 260–264, 2019. cited on p. 30