Journal of Universal Computer Science, vol. 27, no. 7 (2021), 734-754
submitted: 15/5/2020, accepted: 26/5/2021, appeared: 28/7/2021 CC BY-ND 4.0

Assembling the Web of Things and Microservices for the Management of Cyber-Physical Systems

Manel Mena
(University of Almería, Almería, Spain
https://orcid.org/0000-0003-1084-8489, manel.mena@ual.es)

Javier Criado
(University of Almería, Almería, Spain
https://orcid.org/0000-0002-8035-5260, javi.criado@ual.es)

Luis Iribarne
(University of Almería, Almería, Spain
https://orcid.org/0000-0003-1815-4721, luis.iribarne@ual.es)

Antonio Corral
(University of Almería, Almería, Spain
https://orcid.org/0000-0002-0069-4642, acorral@ual.es)

Abstract: Cyber-Physical Systems (CPS) and Internet of Things (IoT) devices are handled by numerous different protocols. The management and connection to those devices tend to create usability and integrability issues. This brings about the need for a solution capable of facilitating the communication between different platforms and devices. The Web of Things (WoT) describes interfaces and interaction patterns among things, thereby abstracting itself from the underlying protocols used to manage those things and their implementation strategies. This paper describes the concept of Digital Dice, an abstraction of IoT devices and CPS capable of leveraging the advantages of microservices architectures and inspired by the concept of Digital Twins. A Digital Dice is a servient system of the WoT domain that represents a device by the features of the device, hence different WoT description models result in different microservices related to the particular thing. The paper explores the definition of Digital Dices and the conversion between WoT Thing Description Models and Digital Dices and the architecture that sustains the system.

Keywords: Cyber-Physical Systems, IoT, Microservices, WoT, Digital Twins, Digital Dice
Categories: C.3, D.2.11, D.2.12, D.2.13, I.6.5, J.7
DOI: 10.3897/jucs.70325

1 Introduction

In the world of IoT devices, there are a number of issues that developers must face. First, the ecosystems of these devices use multiple protocols for interaction. This means that developers must understand all of these protocols. This issue is partially solved with the framework presented by the W3C, the Web of Things (WoT) [WoT 2021]. This framework offers the possibility of declaring *things* from the properties, actions or events that they handle. This is done through the use of the Thing Description (TD), which enables us to understand the capabilities of a device. However, the Thing Description just allows us to declare devices that work using IP protocols whereas there are a lot of *things* that are managed by other kinds of network protocols, even so, the Web of Things is a very robust framework that allows us to establish any kind of interaction (property, action or event).

When using IoT devices or cyber-physical systems, the second problem is that they tend to be devices specially designed for efficiency with limited computing capabilities. This can cause failed requests when many are sent at the same time, making the device unable respond properly.

In addition, there is also the need to virtualize these types of devices for carrying out tests without influencing our business processes [Shetty 2021]. To solve this last problem, researchers introduced the concept of virtual component or Digital Twin (DT) [Tuegel et al. 2011, Koulamas et al. 2018]. Digital Twins are virtual representations of physical elements, systems, or devices, which can be used for different purposes. However, the concept of DT focuses on a monolithic approximation for the management and representation of devices in the real world. It lacks an approximation in multiple levels that specifically addresses each facet of a device.

We developed the concept of Digital Dice (DD) to face the accounted issues [Mena et al. 2019]. Digital Dice is a servient software of the Web of Things based on microservices capable of handling the interaction and virtualization of IoT devices leveraging the advantages of microservices. The WoT defines a servient as "software stack that implements the WoT building blocks. A servient can host and expose Things and/or host Consumers that consume Things. Servients can support multiple Protocol Bindings to enable interaction with different IoT platforms" [WoT 2021]. Besides introducing the concept of Digital Dices, we explain how they are closely related to the Web of Things framework using the standard to leverage its advantages. Furthermore, we study the conversion between Things Description models and the different facets or microservices that our Digital Dice use.

The lack of specific ways to develop WoT servients from their Thing Descriptions counterparts was the primary motivator when we came up with this last idea. The purpose of this paper is introducing a mechanism to automatically create "thing components" or servients for the Web of Things, more specifically Digital Dices. This purpose introduced a series of challenges to overcome, as the different pieces of a Digital Dice add a complexity layer to the model-to-text (M2T) [Rose et al. 2012] transformation required to generate the necessary code to make Digital Dice work. One of those challenges is that Digital Dices are formed by different pieces (or microservices) depending on the *thing* they represent.

As a summary, the main contributions of this paper are the following:

(a) The definition of Digital Dice, a WoT servient based on microservices.

(b) The creation of a semi-automated way to transform Thing Description into Web of Things servients.

(c) The definition of a pipeline transformation process that give us a certain level of assurance in the creation of Digital Dices.

(d) The introduction of an in-deep real scenario to demonstrate the way that our semi-automated transformation works.

(e) The provision of a transformation tool so that the community can help test it and improve it.

This paper is structured as follows. Section 2 introduces some concepts of the Web of Things. Section 3 describes a running example of a particular scenario. Section 4 shows Digital Dice concept as a solution to solve the problems described above. In Section 5 we study the Thing Description and how to convert *thing description* models to Digital Dices. Section 6 presents an example scenario of conversion between some models of the Thing Description and the microservices of the Digital Dices. Section 7 reviews the related work found in the literature. Finally, Section 8 describes the advantages of Digital Dice and some future work.

2 Background and fundamentals

In the IoT domain, the Web of Things standard is focused on the integration of IoT devices into heterogeneous systems. Digital Dice is an implementation of the framework proposed by the Web of Things. There are two main concepts that we must understand before learning about digital dices. The first one is the Web of Things, as it is one of the main pillars of our proposal. The second one has to do with Service Oriented Architectures (SOA) and microservices, more specifically, the advantages that microservices architectures offer to the system.

2.1 Web of Things

The Web of Things [WoT 2021] is a framework established in 2007 to explore the future of the physical Web. The main aim of this framework was to build the ecosystem of the Internet of Things in a flexible, scalable, and open way using web technologies as its application layer. Thanks to the W3C, the evolution of the work spearheaded by Guinard *et al.* [Guinard et al. 2010] has become a standard to define IoT devices that use the Web as the underlying technology. It is important to notice that this standard is able to work just with devices that can be defined using web technologies as the physical communication layer with those devices, so things like Bluetooth devices or other standards like the ones based on the IEEE 802.15.4 specification [Kabalci 2019], or the one used by Zigbee devices [Gislason 2008] are not fully supported. In some cases, workarounds can be found for KNX or Zigbee devices, where through the use of IP gateways we have access to those devices using an IP address and the particular info that the gateway needs to interact with those devices.

The building blocks of the Web of Things can be depicted with four different pillars: (*i*) *Thing Description* provides a schema in a readable data format capable of describing network-facing interfaces and metadata of Things. (*ii*) *Binding Templates* provide a series of guidelines to define network interfaces for particular protocols and IoT ecosystems, *i.e.*, the protocol bindings. (*iii*) *Scripting API* enables the implementation of application logic using a JavaScript API simplifying the development and enabling portability across multiple devices. Finally, (*iv*) *Security and Privacy Guidelines* provide a document to establish a series of guidelines for the secure implementation and configuration of things.

This stack of the different building blocks of the Web of Things is what forms the architecture of the WoT. The concept of Digital Dice was born as a *servient* system of the Web of Things: "A *servient* system of the WoT is a software stack that implements its building blocks. A servient can host and expose Things and/or host Consumers that consume Things. Servients can support multiple Protocol Bindings to enable interaction with different IoT platforms" [WoT 2021]. Digital Dices always have the capability of exposing things. However, there is not always the need to consume a thing as the Web of

Things defines it, given that Digital Dices can sometimes overcome the need for a thing of the Web of Things to work with IP technologies or even creating completely virtual devices without requiring a physical device associated with it.

As the bulk of Digital Dices is centered around the Thing Description, let us take a look at the parameters that the Thing Description requires. Figure 1 shows all the possible parameters required to define a *thing*. The parameters in bold are required, even though, in some cases, those parameters can be implicit. From here, we are going to focus our attention on the important aspects of the Thing Description.

Figure 1 shows the structure of a *thing*, which includes the set of properties, actions and events managed by the said *thing* (the three features are subclasses of `Interaction Affordance`). The interaction at the same time is formed by one or more `Forms` or ways to access the data. These `Forms` help us to define the methods that compose the DD generated by the TD. Figure 1 shows three fields in the `Form` class that are mandatory. Those three fields are used to create our DD. Table 1 contains the description of those fields and their possible types.

Sometimes, those fields are omitted in some instances of Thing Descriptions, that means that default values are assumed. The default values are as follows: *a*) `op` (operation) field has an array of string with the elements *readproperty* and *writeproperty*, if the feature is a `PropertyAffordance`. Futhermore, it will be an *invokeaction*, if it is an instance of `ActionAffordance`. *Subscribeevent* will be used as a default if it is an instance of `EventAffordance`. *b*) `contentType` field has as default value *application/json*.

The last important TD feature is `@type`. This parameter can be found in the *thing* or in different interactions affordances, and basically represents an object with semantic tags (or types). Thanks of being a semantic value, that particular parameter can be used for different purposes. In our case, it allows us to declare different features of a *thing* tailor-made for our Digital Dice, *e.g.*, if an interaction has a user interface or if a particular device uses a protocol not fully supported by the WoT. Further information about this topic is treated in Section 5.

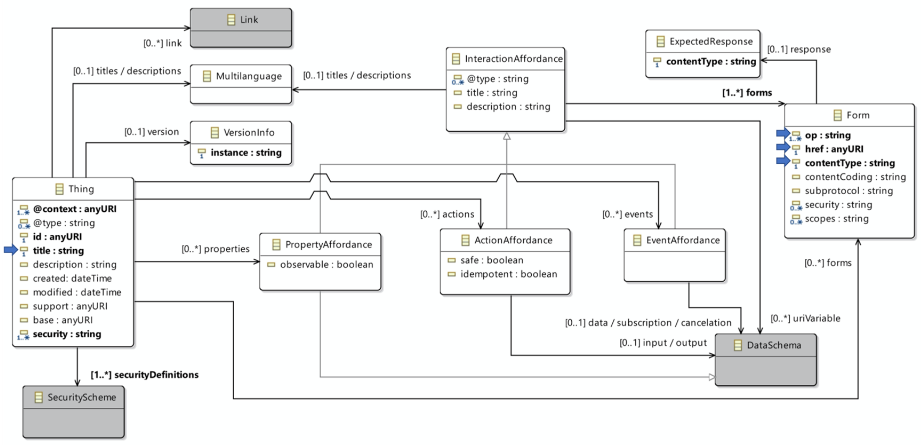


Figure 1: Thing Description schema

Field	Description	Type
op	Indicates the semantic intention of performing the operation(s) described by the form. For example, the Property interaction allows get and set operations. The protocol binding may contain a form for the get operation and a different form for the set operation. The op attribute indicates the form operation type and allows the client to select the correct form for the operation required. Op can be assigned one or more interaction verb(s) each representing a semantic intention of an operation.	string or Array of string (one of `readproperty`, `writeproperty`, `observeproperty`, `unobserveproperty`, `invokeaction`, `subscribeevent`, `unsubscribeevent`, `readallproperties`, `writeallproperties`, `readmultipleproperties` or `writemultipleproperties`)
href	Target IRI of a link or submission target of a form.	anyURI
contentType	Assign a content type based on a media type [IANA-MEDIA-TYPES] (*e.g.*, `` `text/plain' ``) and potential parameters (*e.g.*, `` `charset=utf-8' ``)	String

Table 1: Mandatory fields of the Thing Description

2.2 Service Oriented Architectures and microservices

As we defined in the previous section, the Web of Things is a framework that allows the definition of Cyber-Physical Systems or IoT devices. However, it is up to the developers to decide how they want to implement them. Digital Dice proposes to implement a software capable of hosting and consuming *things* using architectures based on microservices. This type of architectures follows the principles established by the Service-Oriented Architecture (SOA) paradigm, such as the fact that they are developed around business logic, propose standardized service contracts, must contain stateless services, or that they must ensure the discovery of services, among other principles. However, microservices bring a series of advantages by providing our proposal with better resilience and elasticity.

Microservices are services that meet a single functionality in our system, which mainly entails two advantages. The first one, their easy maintainability and testing, as they are less interdependent and more manageable units. The second one, the microservices improve the scalability of the solution, since the management and speed in the creation and destruction of replicas is easier.

3 Running example

A running example of a smart building is used to support the explanation of our approach. In this smart building, there is a few elements that need to be taken into account. Let us suppose the building contains a public parking system where each parking spot has a sensor. Furthermore, each house of the building has a series of connected devices, like the lights, temperature sensors, and Air Conditioner (AC) units, all of them smart devices using a different kind of protocols and datatypes.

Let us define the parameters and protocols of those devices. The parking subsystem is an HTTP enable system capable of controlling 100 spots in a parking lot, where we can get the status of all the parking spots individually or all together. All the lights are connected using KNX enable lights, which are managed by one KNX gateway. A temperature sensor controlled by a Zigbee gateway. Finally, an AC unit managed by an ESP8266 device controlled via a Web Socket server.

As developers, we need to understand how to work with the different devices individually, having to learn all of those protocols. In the example, all devices can be represented with a *thing description*.

4 Digital Dices

In this section, we will define the concept of Digital Dice as well as talk about the different parts that form it and the challenges and strengths that the use of these artifacts provides to our IoT ecosystem.

4.1 Definition

Digital Dices are virtual representations of *things*. A *digital dice* is quite similar to a *digital twin*, but there exist some differences. For instance, a digital twin is usually created for the virtualization of a device, whereas the main task of the Digital Dice is its management. Moreover, a digital dice is based on microservices, each one representing a different feature of a thing. Also, a digital dice aims to be agnostic to the protocols used by each IoT device. Digital Dices are compatible with the framework of the WoT, specifically the Thing Description model. Thanks to this, our digital dices can be compatible with different systems that make use of the standard, like Mozilla Web Thing framework [Mozilla 2021]. The concept of *dice* (multiple facets) is given by how the microservices that represent our devices are characterized:

(a) *Controller* (C). The controller handles the communication with the user, and it manages the orchestration with the rest of the facets or directly to the device. It acts as a gateway for the different facets that compose a DD.

(b) *Reflection* (R). This microservice acts as a replicator of the different properties of the underlying thing. This microservice tries to limit the number of requests performed directly to the device, replicating the different properties and events in a device and maintaining an open communication channel with the device, sending the values to the system database. This way, users of digital dices can request a particular property of a sensor or an actuator without accessing the represented device.

(c) *Data Handler* (DH). This facet handles the communication with the underlying database with two main purposes. On the one hand, it can log different requests performed by the user to trace possible problems originated in our Digital Dices. On the other hand, DH can act as a buffer for the IoT device to handle the requests. First, the requested data is recovered from the database before trying to get them from the physical device. If the data requested is newer than the time threshold configured by our Dice (by default 1 second), then this data will be sent as a response. In conjunction with the reflection, this microservice makes our Digital Dice improve the performance and the reliability of the management of a particular device because there are fewer communication channels directly open with the physical device. This makes our DH less prone to overload that communication channel.

(d) *Event Handler* (EH). This aspect processes the events generated in the IoT device. At the same time, it is intended to manage the connection with a future possible Complex Event Processing (CEP) subsystem [Angsuchotmetee and Chbeir 2016] to establish automatic interoperability between the devices.

(e) *User Interface* (UI). The user interface establishes a frontend for each interaction controlled by our Digital Dice. The interactions can offer a UI that will be declared in the Thing Description model that supersedes our DD. Besides those individual interfaces for each interaction, we have a mechanism in our UI microservice to generate single UI mashup made up by all the individual UIs of each interaction. This mechanism offers the user a simple reusable interface to interact with the device.

(f) *Virtualizer* (V). This facet tries to replicate the behavior of the underlying device. If declared in the thing description, a user of DDs can request virtualized data instead of real data to try out their solutions.

Once described its structure, a Digital Dice can be described as a set of microservices {C, R, DH, EH, UI, V} where Controller (C), Data Handler (DH) and Reflection (R) are mandatory features in any DD instance, and the rest can appear depending on the Thing Description defined. These facets of the Digital Dice must be capable of managing the different aspects of the device represented. It is important to highlight that not all facets will be part of the digital dice, as one of the main advantages of our proposal is the modularity, as not all of the devices need to be represented by all the facets. Besides this fact, we enable the possibility of adding new facets like as, for example, a facet capable of offering a voice-activated interface or an open data manager capable of proactively publish data in Open Data platforms.

The connection of facets with IoT devices is one of the objectives that this proposal overcomes. In order to do that, a library capable of managing multiple protocols is required. Besides creating such libraries, we need to establish what microservices have a direct connection with the IoT device. To that end, we classified the microservices in two categories; (*a*) *hard related facets* for microservices that have a direct connection with the device (*i.e.*, Controller, Reflection and Event Handler); and (*b*) *soft related facets* for microservices that do not establish a direct connection with the device (*i.e.*, UI, Data Handler and Virtualizer).

The hard related facets use WoTnectivity[1], a library designed to manage different protocols but following a common use pattern to reduce its learning curve. Currently, this library is ready to work with three different IP protocols (KNX, WS, HTTP), but the idea is to add more protocols in the future.

4.2 Challenges and Strengths

One of the challenges is to establish a system architecture for Digital Dices. This architecture has to sustain multiple copies of the same microservices, allowing load balancing between them and detecting when a microservice is overloaded, starting a new replica of it to operate adequately. Figure 2 illustrates an example for the management of three Digital Dices with the different facets required to represent the devices proposed in the running example. The DD #1 has the four facets described, and it is responsible for handling an actuator to turn a switch on and off. The DD #2 only has two facets active because the interaction with the climatological information service is done through the Data Handler and Controller facets. The DD #3 has a duplicate Data Handler facet because the threshold number of accesses has been exceeded. The use of microservices and the modularity of our approach are some of the advantages of our proposal.

One of the advantages provided by the modularity of our solution is that we can replicate the different facets that compose our Digital Dice. This provides the ability to increase the maximum limit of requests received by our Digital Dices. In addition, we are able to scale only the facets that are most affected by said requests.

Another pillar of this proposal is the management of contextual information derived from IoT devices. For example, the geolocation of the device can affect the availability of the DD that represents it, as we can establish that only the users from a specific area can request information on our DD. All of this business logic can be implemented in the controller of the particular Digital Dice.

[1] WoTnectivity - https://github.com/acgtic211/wotnectivity

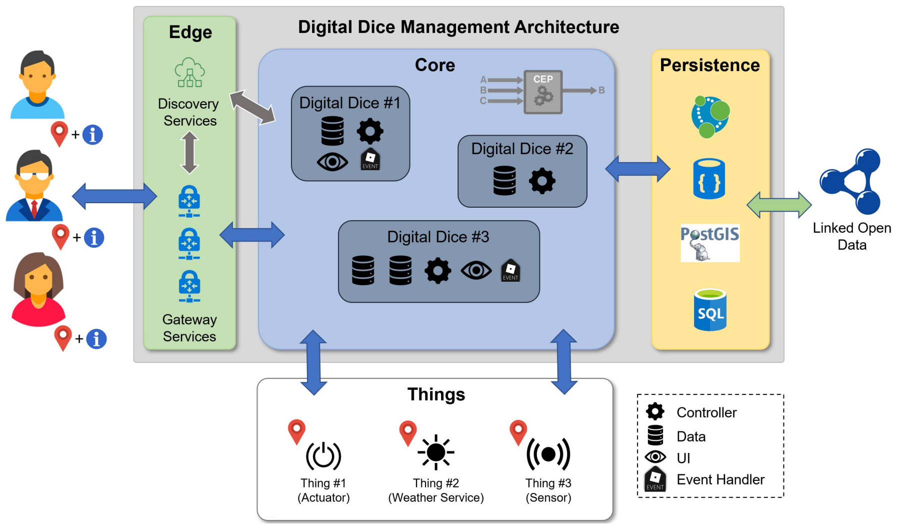


Figure 2: Digital Dice Architecture

Digital Dices establish communication with the users following the standards, mechanisms and technologies by the W3C, particularly in two ways, through a REST API and with Server Sent Events (SSEs). A Digital Dice offers to the user a simple REST API to interact with it. The addresses deployed follow the pattern `https://{ipaddress}:{port}/{thing.name}/{property|action|event}/{InteractionAffordance.title}` as it can be seen in the Figure 3 in the property `temp` (temperature). These addresses will be declared in the TD of the particular DD as `https://{ipaddress}:{port}/{thing.name}/`. Figure 3 shows the Thing Description of the Air Conditioner in the running example where it has two interactions, a property called temp that sets and reads the temperature set in the AC and an event called overload that warn the user of a problem with the internal compressor.

In some cases, properties and events can be exposed through SSEs. This is particularly useful when working with realtime data. As shown in Figure 3 the temperature property has the parameter subprotocol declared as SSE, this property declares that the temperature is exposed not as a simple resource but as an open channel to the user, *i.e.*, a keep-alive connection with the address. Thanks to the fact that Digital Dices follow the same communication patterns, no matter the underlying device, makes the adoption of them very easy for the developer because once the digital dice is deployed he/she can forget about the particular protocols used by the physical device.

5 Digital Dice Transformer

Digital Dices use the Thing Description of the WoT in two different ways, as shown in the Figure 4. The first one, being capable of exposing itself as a *thing* of the WoT, so other systems compatible with the framework can make use of it. This method requires a manual intervention from the developer, as he/she has to create the Digital Dice following the parameters established by the definition of the DD, as well as decide if each of the

```
{
"@context": "https://www.w3.org/2019/wot/td/v1",
"id": "urn:dev:ops:32473-ACH1-1234",
"title": "dd-ac-h1",
"properties": {
     "temp" : {
         "type": "string",
         "uriVariables": {"value": {"type": {"string"}}},
         "forms": [{"href": "http://h1.example.com/dd-ac-h1/property/temp{?value}",
                    "contentType": "Application/json;"},
                   {"href": "http://h1.example.com/dd-ac-h1/property/temp/sse",
                    "contentType": "Application/json;", "subprotocol": "sse"}]
     }
  },
"events": {
     "overload" : {
         "type": "string",
         "forms": [{"href": "http://h1.example.com/dd-ac-h1/event/overload"
                "contentType": "Application/json;",
                "subprotocol": "sse"}]
     }
  },
}
```

Figure 3: Thing Description of a Digital Dice instance

generated microservices requires some kind of business logic or needs to be taken into consideration not available in the Thing Description. The second use is that Digital Dices can consume *things* and expose them through the use of the thing description adding a series of advantages, like a common way to access to them via a well-formed API, or improving the performance through the use of microservices and limiting the number of connection that affects the represented *thing*.

In this last use case, the creation of the Digital Dice can be partially automatized applying model-to-text (M2T) transformation techniques [Rose et al. 2012]. As shown in Figure 4, the *thing* needs to have a TD associated with it. As part of the Digital Dice ecosystem, we developed the tool **TD2DD Transformer**[2]. This software converts a TD to the different microservices exposed by the Digital Dice that can represent it, as well as offering the necessary software infrastructure to the system so it can work.

[2] TD2DD Transformer - https://github.com/acgtic211/td2dd-transformer

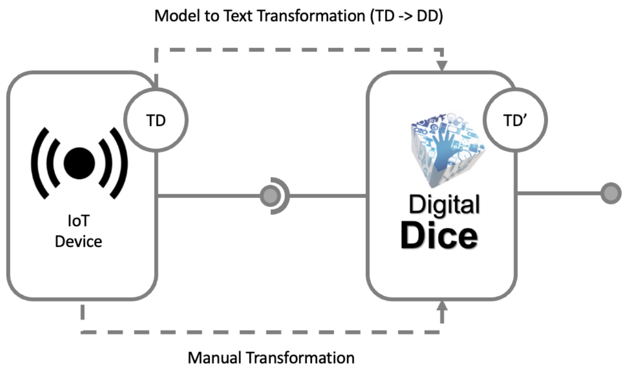


Figure 4: Thing Description use cases in Digital Dices

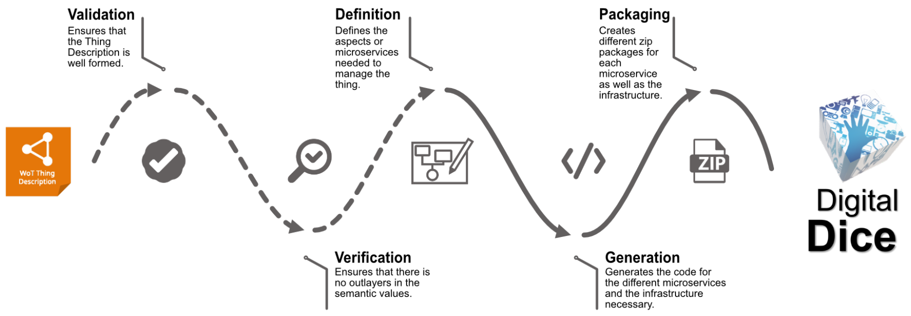

WoT Thing Description
Validation
Ensures that the Thing Description is well formed.
Verification
Ensures that there is no outlayers in the semantic values.
Definition
Defines the aspects or microservices needed to manage the thing.
Generation
Generates the code for the different microservices and the infrastructure necessary.
Packaging
Creates different zip packages for each microservice as well as the infrastructure.
ZIP
Digital Dice

Figure 5: TD2DD transformation workflow

For that purpose, the transformer converts a Thing Description into a Digital Dice following a series of steps designed to improve its reliability. Furthermore, the process needs to avoid the user intervention, only requiring his/her participation when choosing the TD that he/she wants to convert and downloading the parts generated. Figure 5 shows the workflow that the transformation process follows to generate the Digital Dice and its infrastructure. The workflow executes five different steps: validation, verification, definition, generation and packaging.

The Table 2 shows the mapping of each step in the source code. In it, we can see how each step has a series of methods that define its functionality. The validation is managed by the (validator.service.ts), meanwhile, the other steps are managed by the (dd-builder.service.ts). The following subsections describe each step of this workflow.

5.1 Validation and Verification

The first step is executed while we are introducing the Thing Description in the transformer. The software has a *textarea* where we introduce the thing description that we want to transform in a Digital Dice. Once the user introduces the TD in the *textarea,*

Step	File	Methods
Validation	`validator.service.ts`	`validate(data):boolean`

Verification	`dd-builder.service.ts`	`verify(td):string[]`
		`librariesNeed()`

Definition	`dd-builder.service.ts`	`servicesNeeded(td)`

Generation	`dd-builder.service.ts`	`private buildController()`
		`private buildDataHandler()`
		`private buildEventHandler()`
		`private buildUi()`
		`private buildReflection()`

Packaging	`dd-builder.service.ts`	`zipController()`
		`zipDataHandler()`
		`zipEventHandler()`
		`zipUi()`
		`zipReflection()`
		`zipInfrastructure()`

Table 2: Source code mapping for the transformation steps

```
1   validate(data):boolean {
2       try{
3           var jsonData = JSON.parse(data);
4       }catch(e){
5           return false;
6       }
7       var ajv = Ajv({ allErrors: true });
8       apply(ajv);
9       var valid = ajv.validate(this.schema, jsonData);
10      if (!valid) console.log(ajv.errors);
11      return valid;
12  }
```

Listing 1: Validation code

the system checks if the user is complying with the marked directives found in the JSON schema of the Thing Description[3]. If not, an attention mark will be shown in the top of the box and the transformation button will not be available to the user, to avoid starting the conversion process. Listing 1 shows the source code that complies with the aforementioned validation process. Between lines 2–6, the method checks if the input TD is well-formed. Once it is known that the JSON is correct, the code between lines 7–12 is the one that checks the TD schema. One of the advantages of validating the code with the official TD schema is that the code generated from the transformation cannot be incorrect if the source model is correct and assumes that the transformation rules are correct.

Once the thing description complies with the parameters established by the schema, the verification step begins. This step starts the processing of the TD taking a look into the semantic values of the TD, more specifically the `@type` parameters. This process checks if the values are compliant with those managed by the Digital Dice. These parameters alter the DD generated in the definition and generation steps. It is important to notice that the `@type` values can be declared in two different levels inside the TD, at the thing level and at the interaction level. The values of the `@type` that can alter the behavior in the construction of the Digital Dice are explained as follows.

First, the property value *ui*. When this property is found at a *thing* level, a global user interface is needed with all of the different interactions that have the parameter `@type` with the value *ui*. If this parameter is found at interaction level, the system needs to expose a user interface for that particular interaction. A Digital Dice can have multiple `InteractionAffordances` with *ui* exposed, but it is not mandatory to expose a global user interface.

Another value that can alter the behavior of a DD is `@type` with the value *virtual*. At the thing level, this value indicates that all the `InteractionAffordances` have to be virtualized inside the digital dice. In this case, when a user interacts with the digital dice, he/she will interact with a completely virtual device and not with the physical one. If we find this property value at the interaction level, it indicates that the particular interaction has to be virtualized. Unlike the *ui* value, if the *virtual* value is found at thing level, it has an impact over the whole device, indicating that the device managed by the DD is completely virtual.

Digital Dices include in some cases other contextual parameter for `@type` to clarify technologies not totally compatible with the WoT (*e.g.*, KNX). KNX devices cannot be represented by a Thing Description, as they do not use web technologies by themselves. So in order to declare a thing description of a KNX device, we need a proper middleware software [Ngu et al. 2016] and a device known as KNX IP gateway. The actual Thing

[3] TD JSON Schema for validation - https://cutt.ly/7kzePnP

Description of the lights of the running example is a mock Thing Description compatible with our Digital Dice ecosystem. Furthermore, this TD uses special data schemas to work with the particular Schneider KNX IP Gateway that needs to use especial parameters to establish communication with the lights. To define this kind of devices we use the value *knx* for `@type` of the device, this value warns the DDs of the different considerations that they have to implement in order to manage these devices.

5.2 Definition

The definition stage is the cornerstone of the whole transformation process as it defines the different microservices where the model-to-text transformation takes place. To do so, first, the provided transformer tool, after checking sure that the Thing Description is well-formed through the first two stages (Verification and Validation), automatically discovers what microservices will be needed to represent a particular *thing*. Figure 6 shows the microservices generated by the conversion of the TD to DD and how certain values of the Thing Description generate different configurations for the Digital Dice. Different properties, actions or events will generate different configurations of microservices. The interactions will always generate at least the three different microservices, *Controller*, *Reflection* and *Data Handler*. An *Event Handler* microservice will be generated when the TD contains events.

As we stated in Section 2.1, in some cases, default values are considered if they do not appear in the TD. Furthermore, a *Virtualizer* or a *User Interface* microservice will be generated when the semantic parameter `@type` has the values *virtual* or *ui* respectively. In the Listing 2 we can see this fact, as lines 3 and 4 are two methods that analyze the `@type` of the Thing Description to see the communication libraries that the project will need, as well as see if the *ui* or *virtualizer* microservices are going to be needed. In line 5, the DD (Controller and Reflection) base microservices are shown together with an auxiliary service that only needs to be deployed once, no matter the number of DDs that our systems have. This service contains all the infrastructure needed to support the DD ecosystem. Furthermore, between lines 6–17, we define all the microservices needed in accordance with the Figure 6.

```
 1  servicesNeeded(td) {
 2      this.parsedTd = JSON.parse(td);
 3      var types = this.verify(this.parsedTd);
 4      this.librariesNeed();
 5      var services = { services: ["infrastructure", "controller", "
            ↪ reflection"] };
 6      types.forEach(element => {
 7        services.services.push(element);
 8      });
 9      if (this.parsedTd.events) {
10        services.services.push("eventHandler");
11      }
12      if (this.parsedTd.actions || this.parsedTd.properties) {
13        services.services.push("dataHandler");
14      }
15
16      return services;
17  }
```

Listing 2: Definition code

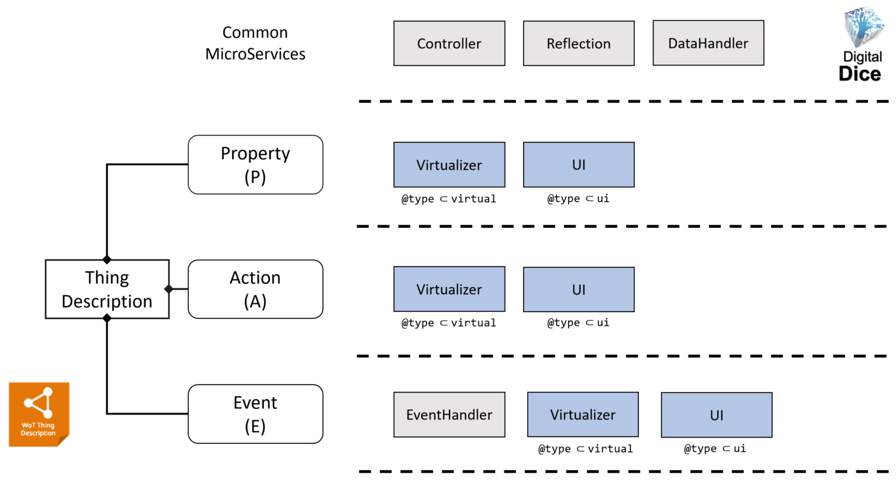


Figure 6: Definition of the microservices to manage a TD

5.3 Generation

Once defined the microservices, the system will generate the different files needed to create said microservices, as well as the infrastructure to support the Digital Dices. In this stage is where the bulk of the model-to-text transformation takes place, defining the classes, methods, and files necessaries to create a template of each microservice to help the end-user develop *thing* servient or component faster. However, before that, the software infrastructure that supports the Digital Dices must be generated.

The infrastructure to manage the Digital Dice ecosystem is based on three artifacts. The first one is a MongoDB[4] database. We choose this database because it is capable of horizontal scaling with sharding. The second is the use of a Discovery Service, more specifically, Spring Cloud Netflix Eureka[5]. We have developed the solution of Digital Dices in the Java programming language, and using Eureka brings about full compatibility with Server-Sent Events, full load balancing, circuit-breaker pattern to our system effortlessly. The system is capable of registering each facet or microservice of our Digital Dice and its replicas. Besides the service discovery, DD uses other service dedicated to act as a gateway for all the requests sent by the user, a Spring Cloud Gateway[6]. This artifact provides a library for building an API Gateway on top of Spring MVC. Spring Cloud Gateway aims to provide a simple way of routing APIs and takes care of a series of concerns such as security, monitoring/metrics, and resiliency. The advantage of using this Gateway is that it is completely compatible with Eureka and can automatically create routes for the services registered in the Discovery Service. These three artifacts and the rest of the microservices generated are managed by Dockers [Jaramillo et al. 2016], and Kubernetes [Hightower et al. 2017] since they provide different ways to scale and downscale when necessary automatically. These services have a common codebase no matter the Digital Dice created, so we have a fragment of static code to perform the generation step for the infrastructure.

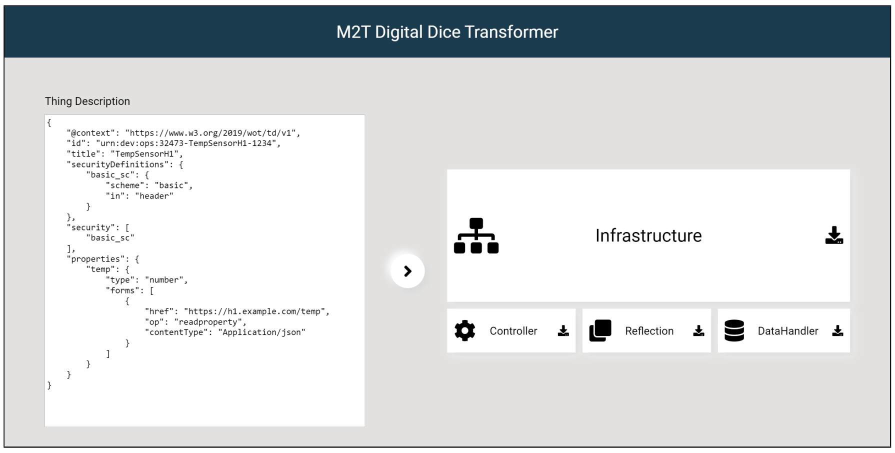
[4] Mongo Db - https://www.mongodb.com
[5] Spring Cloud Netflix - https://spring.io/projects/spring-cloud-netflix
[6] Spring Cloud Gateway - https://spring.io/projects/spring-cloud-gateway

Once the infrastructure is defined, the generation process builds the different microservices of the Digital Dices. The *Controller* generates a route and a method for each `InteractionAffordance`, two in the case that the particular interaction has an SSE behavior, or even more if the interaction has more than one `Form` to interact with the device. The *Controller* has other features like exposing the TD to manage the particular DD; the generation of user-generated events to extend the functionality of the system; or the orchestration of the request sent by the user to the other services acting as a gateway for the particular Digital Dice. The communication with the rest of the microservices is managed following reactive patterns. This helps us to improve the performance of the system using a minimal amount of resources. The *Reflection* requests the `PropertyAffordances` of a device *per* second and saves the data recovered into the database if it has changed since the last time requested. This microservice will do long polling every second if the physical device does not have any real-time data protocol implemented. The *Event Handler* propels the events declared on the device to the *Controller* and saves them into the database if needed, as well as handling the user-generated events sent by the *Controller*. The *Virtualizer* generates one method for each `InteractionAffordance` declared with the `@type` as *virtual*, the same with the *UI* but with the value *ui*.

Microservices are generated into source code that must be compiled. Each facet of the Digital Dice is an individual project, and every project has all the necessary source code files as well as the dependency files (`pom.xml`), the `application.properties` necessary for Spring and a `Controller.java` file that implements the behavior described above. If the user wants to modify the behavior or implement something different for the particular DD, he/she needs to modify the generated source code. In some cases, the user has to intervene to implement part of the methods as the transformer is a "partial" M2T transformation tool [Burgueno et al. 2019].

5.4 Packaging

The last step in the transformation process is the packaging of each microservice or facet. To do so, each project created in the generation step is scaffolded and compressed to a zip file. After that, the user can download and modify it. The Figure 7 shows the web



```
{
    "@context": "https://www.w3.org/2019/wot/td/v1",
    "id": "urn:dev:ops:32473-TempSensorH1-1234",
    "title": "TempSensorH1",
    "securityDefinitions": {
        "basic_sc": {
            "scheme": "basic",
            "in": "header"
        }
    },
    "security": [
        "basic_sc"
    ],
    "properties": {
        "temp": {
            "type": "number",
            "forms": [
                {
                    "href": "https://h1.example.com/temp",
                    "op": "readproperty",
                    "contentType": "Application/json"
                }
            ]
        }
    }
}
```

Figure 7: TD2DD Web Application User Interface

user interface of the TD2DD Transformer, in which appears the thing description of the temperature sensor of the running example, and all the packages generated with the microservices the digital dice needs.

6 Conversion example

Let us see now what would happen if we had to develop an app capable of managing the devices shown in the running example. First, we have to learn how every device works and how to communicate with the devices using their respective technologies. This process requires a lot of time, even only considering four types of devices. If the system was governed by a Digital Dice ecosystem, the time to develop any application related to the use of the devices would be minimal, as the developer would just need to know how to make simple HTTP requests following a well-formed API.

Being the conversion between Thing Descriptions and Digital Dices the main contribution of this paper, this section will show the conversion steps for the lights and the air conditioner of the running example. The KNX protocol governs the lights, and the air conditioner works with Web Socket. First, let us see the particularities found in both Thing Descriptions[7]. The lights got more than one `Form` to access to the `PropertyAffordance` *status*, one of those with a *sse* subprotocol. That means that it is necessary to leave a connection open with the device to get the value of the property. Both the property *status* and the action *toogle* require independent *UI*, but in this case not a global one. The air conditioner has a more simple model. The only thing that the system has to take into account in the next steps is the fact that the subprotocol has the value *ws* indicating our Digital Dice that it has to maintain a subscription via web sockets to get the data for said device.

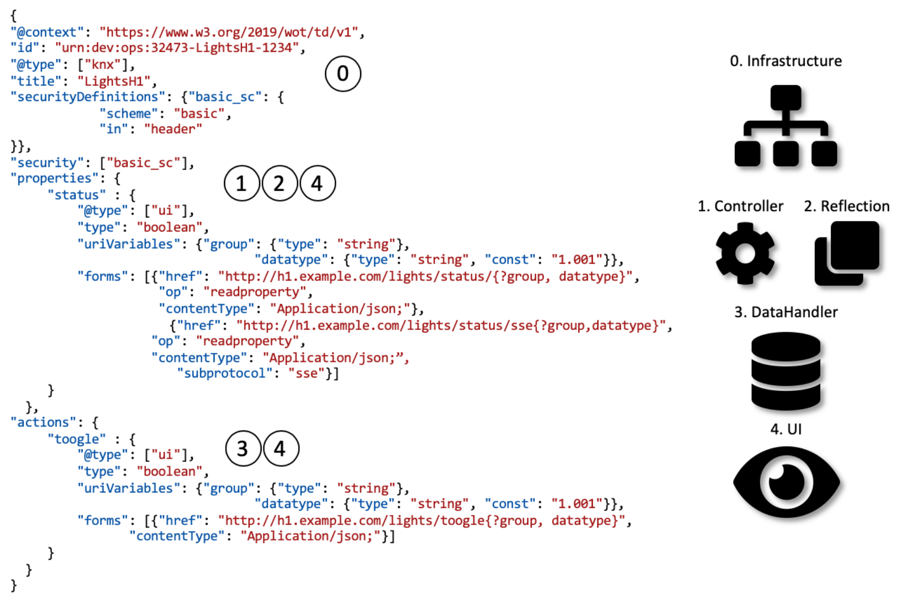
The system handles the **Validation** of the Thing Descriptions that we want to convert into a Digital Dice. By default, the two thing descriptions are well formed as they are compliant with the parameters established in the JSON schema of the TD. Once the thing descriptions are validated, we progress to the **Verification** step. Like in the validation step, the semantic values of the TDs are known to the Digital Dices so the system has no problem of verifying.

In the **Definition** stage, we see the differences between both conversions. Figure 8 shows what microservices are generated from the Thing Description that represents the KNX lights of the running example showing what fragments of the TD affect the generation of the different microservices.

The figure depicts how every Thing Description generates an infrastructure and seeing than the TD has `InteractionAffordances` (properties, actions or events) means that the digital dice that represent the device will have at least a *Controller* and a *Reflection*. The device has an action, this means that the DD will have a *DataHandler*. If we take a closer look to the figure, we can see that the semantic `@type` has a value of *ui* meaning that the digital dice will have a UI microservice to show two interfaces: one for the property and the other one for the action. In the case of the Air Conditioner (AC) the differences strives in the fact that the Digital Dice that represents this device will have an *EventHandler* and no *UI*.

In the **Generation** step, the transformation tool builds all necessary files for the Digital Dice. For instance, the files generated in the *Controller* microservice for the AC, the system will generate the dependency file for the Spring project, the `pom.xml`, and

[7] Thing descriptions of running example - https://cutt.ly/Kkze1qM



```
{
"@context": "https://www.w3.org/2019/wot/td/v1",
"id": "urn:dev:ops:32473-LightsH1-1234",
"@type": ["knx"],
"title": "LightsH1",
"securityDefinitions": {"basic_sc": {
            "scheme": "basic",
            "in": "header"
}},
"security": ["basic_sc"],
"properties": {
     "status" : {
         "@type": ["ui"],
         "type": "boolean",
         "uriVariables": {"group": {"type": "string"},
                                "datatype": {"type": "string", "const": "1.001"}},
         "forms": [{"href": "http://h1.example.com/lights/status/{?group, datatype}",
                    "op": "readproperty",
                    "contentType": "Application/json;"},
                     {"href": "http://h1.example.com/lights/status/sse{?group,datatype}",
                   "op": "readproperty",
                   "contentType": "Application/json;”,
                      "subprotocol": "sse"}]
     }
   },
"actions": {
     "toogle" : {
         "@type": ["ui"],
         "type": "boolean",
         "uriVariables": {"group": {"type": "string"},
                                "datatype": {"type": "string", "const": "1.001"}},
         "forms": [{"href": "http://h1.example.com/lights/toogle{?group, datatype}",
                "contentType": "Application/json;"}]
     }
   }
}
```

Figure 8: TD2DD Example - Definition

the `application.properties` file with configuration parameters for the microservice. After that, the system generates a series of auxiliary files that helps the *Controller* communicate with the other microservices.

Listing 3 shows the code generated for the Controller microservice, more precisely, the main file for that microservice. The code uses a series of properties (lines 3 and 5). First, the `env` variable (line 3) accomodates the configuration of the microservice, with parameters such as: the Thing Description of the represented device and others like ports, the Discovery Service address, some parameters related to the database connection, among others parameters configured in the `application.properties` file. Next, the `dicoveryClient` (line 5) provides access to the discovery client to allow the Controller microservice the discovering of other microservice addresses when needed. The `dataHandler` property (line 7) implemets a Java interface that shows all the method available, in this case, on the *DataHandler* microservice. The next property `WebClient` is used to establish communication with the other microservices. Furthermore, we can see three methods generated (lines 10–12, lines 14–18 and lines 20–24) following the specification defined in Section 5. It is important to notice that, due extension constraints, we cannot show all the code generated, but it is available in project code repository. The method `returnTD()` (lines 10–12) returns the thing description managed by the digital dice. The method `getProperty` (lines 14–18), request the *DataHandler* the particular property that the user wants, and once the controller gets the response it sends it back to the user. The method `getPropertySSE` (lines 20–24) performs the same that the second one, but with a *sse* behaviour. We can see in the original Thing Description (Figure 3) for this example that the user has two different ways to access the temperature data.

The `Controller.java` contains other methods to change the temperature with a POST request to the particular address or the used to manage the events. In addition to

```
public class Controller {
@Autowired
private Environment env;
@Autowired
private DiscoveryClient discoveryClient;
@Autowired
private IDataHandler dataHandler;
private WebClient wc;
@RequestMapping(method = RequestMethod.GET, path="/", produces="application/json")
public ResponseEntity returnTD() {
        return new ResponseEntity(env.getProperty("td"), HttpStatus.OK);
}
@RequestMapping(method = RequestMethod.GET, path = "/property/{property}")
public Mono getProperty(@PathVariable("property") String propertyName) {
        this.wc = WebClient.create(this.discoveryClient.getInstances("ac-h1-DATAHANDLER").get(0).getUri
              ↪ ().toString());
        Mono<PropertyData> res = webc.get().uri("property/{prop}", propertyName).retrieve().bodyToMono(
              ↪ PropertyData.class);
        return res;
}
@RequestMapping(method = RequestMethod.GET, path = "/property/{property}/sse", produces = MediaType.
      ↪ TEXT_EVENT_STREAM_VALUE)
public Flux getPropertySSE(@PathVariable("propertyName") String property) {
        this.wc = WebClient.create(this.discoveryClient.getInstances("ac-h1-DATAHANDLER").get(0).getUri
              ↪ ().toString());
        Flux<PropertyData> res = webClient.get().uri("/property/{prop}/sse", propertyName).retrieve().
              ↪ bodyToFlux(PropertyData.class);
        return res;
}
...
}
```

Listing 3: Controller code (AC House1)

this microservice, we have to take into account that there are others. Apart from this microservice, the transformation process generates more code. In this paper, we have only focused on a piece of the code to explain some details of the transformation process of Things Descriptions into Digital Dices. As we just advanced, the complete transformation code is available at the code repository. Doing the transformation manually for each of the devices can become a very tedious process, hence the need for a code generator that offers the possibility of the rapid deployment of a Digital Dice ecosystem.

Once all the microservices have been generated, in the **Packaging** stage, a zip file is generated for each project. Finally, the system returns the project generated to the user, who can then download, set, and modify the provided files and run the Digital Dice.

7 Related work and discussion

The idea of linking the Web with IoT devices is not new. Authors like Guinard *et al.* [Guinard et al. 2010, Guinard et al. 2011, Guinard et al. 2016] are working actively in making this a fact. They propose a Web of Things architecture and best practices based on RESTful principles. Our Digital Dice try to go a step further leveraging the latest trends in the Web Services architecture, the use of microservices as a building block of our solution.

As we explain through the article, the Digital Dices are closely related to the Web of Things [WoT 2021], being this a reference framework that seeks to counter the gap found in the IoT world. The idea of using the Web of Things comes from the need of making our Digital Dice concept compatible with other system and software like Mozilla WebThings [Mozilla 2021]. Mozilla offers a unifying application layer, linking together multiple underlying IoT protocols using existing web technologies. The Mozilla WebThings is an open-source implementation of the WoT that offers two primary software artifacts: (a) *WebThings Gateway*, a web-based user interface to monitor and control smart home devices; and (b) *WebThings Framework*, a reusable software component to help developers build their own web things, which directly expose the Web Thing API. Moreover, as our Digital Dice is a solution compatible with the Web of Things, nothing stops a user of the WebThings Gateway from using a Digital Dice as a virtual Smart Device in it.

An approach more closely related to our proposal is that one presented in [Khanda et al. 2017]. In this work, the authors propose a solution based on Jolie to manage a prototype platform supporting multiple concurrent applications for smart buildings. This proposal uses a sensor network and a distributed microservices architecture. The solution has the caveat of being focused on a specific domain, thus not giving a broad solution to the management of IoT devices or Cyber-Physical Systems.

The proposals given by [Sun et al. 2017] and [Vresk and Čavrak 2016] offer a general solution to the use of microservices in a non-domain specific approach. The solution proposed by [Sun et al. 2017] makes use of eight different microservices to separate aspects of an IoT centric systems, such as security, events, devices, etc. However, from our point of view, this solution does not really take advantage of the power of microservices, because all those microservices have a high amount of complexity, difficulting the replication and maintainability of the system. In contrast, the Digital Dice Architecture is a more fine-grained solution with less complex microservices helping to enhance the capabilities of microservices. Therefore, one of our advantages is that our system is able to replicate each aspect of a particular *thing* individually as soon as they are needed, as our proposal handles each aspect of a device as a microservice.

Niflheim approach [Small et al. 2017] proposes an end-to-end middleware for a cross-tier IoT infrastructure that provides modular microservice-based orchestration of applications to enable efficient use of IoT infrastructure resources. This solution provides a framework way to divide the resources of a cloud infrastructure in a 3-tiered infrastructure pool; (a) *Cloud tier*, to support pools of Open-Stack hosted cloud VMs; (b) *Fog tier*, for Docker containers; (c) *Mist tier*, for CerberOS powered IoT devices. We see Niflheim as a complementary system since it could provide a reliable architecture for the deployment of our Digital Dices.

The IoT-A project [Bauer et al. 2013] is an Architectural Reference Model (ARM) that establishes a common understanding of the IoT domain and provides to IoT system developers a common technical foundation and a set of guidelines for deriving a concrete IoT system architecture. This ARM is taken into account in both the WoT and the Digital Dice architectures to establish the communication between different IoT devices.

Aside from Digital Dices, in this paper, we introduced the model-to-text transformation process used to generate the code of said Digital Dices. Code generation for IoT systems has been another topic that we have explored in the literature to complete our proposal on Digital Dices. We found the work in [Steinmetz et al. 2017], where the authors propose a tool that facilitates the creation of Internet of Things systems using ontologies and, through this model, generates code. Another work in code generation for IoT systems is presented in [Sharaf et al. 2019], in which the authors provide a model-driven code generation framework (CAPSml) to generate ThingML models [Harrand et al. 2016], and it offers an open-source tool capable of code generation.

Both of these approaches offer a broader and personalized way of generating code for IoT devices using different modelling languages. Our approach offers a way to directly use the metamodel of the Web of Things - Thing Descriptions to generate Digital Dices as its only purpose. That is the main advantage of our approach, as the Digital Dice Transformer has only one particular purpose does not need a modelling language to define rules, so we do not need to change the rules of the conversion nor the stages mentioned in Section 5, making our approach less prone to failure. However, this issue could be just the main disadvantage of our proposal since there is no way to customize the current code generation easily. Our solution is an ad-hoc proposal where the changes in the code generation involve some changes in the base code of the transformer application. On the contrary, our transformer allows us rapidly generate Digital Dices. In the WoT domain, there are not many, if at all, code generators that leverage the well-known framework, hence the novelty of our approach.

Once discussed the related work about WoT servients and their applications, it is important to highlight the principal milestones reached to come up with the idea of Digital Dices and the transformation tool. Those milestones were the following:

(a) Thinking how to improve and define interoperability and integration of IoT ecosystems.

(b) Choosing the Web of Things as a robust and recognized metamodel to declare IoT devices.

(c) Improving WoT servients adding an architecture helping us to leverage the advantages of microservices (Digital Dices).

(d) The need to quickly generate at least template code for those servients for a rapid deploy of a DD ecosystem.

(e) Have flexibility in the transformation process and assure that the artifacts generated follow a concrete schema using the Thing Description as a base.

8 Conclusions and Future Work

This proposal aims to improve interoperability, integration, and management between both real and virtual IoT systems and devices. To do this, the functionality of devices is abstracted to a set of microservices making use of WoT standards. Microservice architectures help us to establish operational mechanisms that permit better use of resources and facilitate maintainability. This paper offers a solution to convert external IoT devices described as a WoT Thing Description into Digital Dices. At the same time, Digital Dices expose a Thing Description so they can be used by external systems seamlessly. An example scenario in the smart building domain has been used to understand transformation process.

The use of Digital Dices offers a series of advantages. Firstly, due to being a software abstraction help us define a common communication language no matter which device our Digital Dice is representing, which leads to the use of a typical pattern to connect to features defined by said device. Secondly, the internal behaviour of our Digital Dice lessens the number of requests received by the device . What provokes that in some cases is unnecessary to connect with the device to recover a property value of the said device. Other advantage is that given the fact that our system is based on microservices, it allows us to replicate only the microservices of the system that receive more requests. This gives our system flexibility, thus is always trying to minimize the use of resources. Finally, the compatibility of our solution with the WoT definition Schema offers other systems like the Mozilla IoT Gateway the possibility of using the schema defined by our Digital Dice to interact with.

The disadvantages Digital Dices are mainly inherent to microservices. First, the architectural complexity that microservices usually require. It is easier to develop a monolithic application than software based on microservices. Furthermore, this kind of software requires outside gateways, discovery, and other auxiliary software to orchestrate the communication inside our system. Besides this fact, in some circumstances, DD can be slower to respond than a direct connection to IoT devices but usually more reliable.

Besides the definition of Digital Dices, in this paper, we provide a transformation tool to generate boilerplate code for Digital Dices from Thing Descriptions rapidly. This helps the end-users to create Web of Things components faster than doing it manually, providing a shorter time to deploy virtual representations of things.

In this transformation tool, the source models are validated against their own schema (Thing Description Schema). The code generated from the transformation cannot generate incorrect code if the source model is correct, assuming that the transformation rules are correct. However, it must be acknowledged that no empirical study has been carried out on this statement. As a future consideration, we will try to provide this transformation process with mathematical notation and MDE foundations such as those proposed by the authors in [Vallecillo et al. 2012, Guerra et al. 2013].

As future work, we are working on comparing different IoT and WoT middleware systems with our solution, performance and reliability-wise. An extension of the WoT thing description is also in the works. This extension aims to try to establish a way to improve machine-to-machine communication through the use of complex events. With this, we want to establish a series of scenarios where a Digital Dice can react to the changes produced by others. This extension has to maintain the core of the Thing Description as it is, so our system is still totally compatible with others, but adding the necessary features to be able to work together with other devices without human intervention.

Acknowledgements

This work is funded by the Spanish MINECO project (ref. TIN2017-83964-R) and CEIMAR consortium (ref. CEIJ-C01.2). Manel Mena has been funded by a grant from the Spanish Government FPU17/02010.

References

[Angsuchotmetee and Chbeir 2016] Angsuchotmetee, C., Chbeir, R.: A survey on complex event definition languages in multimedia sensor networks. In *Proceedings of the 8th Int. Conf. on Management of Digital EcoSystems*, pp. 99–108. ACM, 2016.

[Bauer et al. 2013] Bauer, M. et al.: Internet of Things – Architecture IoT-A Deliverable D1.5 – Final architectural reference model for the IoT v3.0, 2013.

[Burgueno et al. 2019] Burgueño, L., Cabot, J., Gérard, S.: The Future of Model Transformation Languages: An Open Community. *Journal of Object Technology*, 18(3), 2019.

[Gislason 2008] Gislason, D.: Zigbee Wireless Networking. Oxford: Newnes, 2008.

[Guerra et al. 2013] Guerra, E., de Lara, J., Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Automated verification of model transformations based on visual contracts. Autom. Softw. Eng. 20(1), pp. 5–-46, 2013.

[Guinard et al. 2010] Guinard, D., Trifa, V., Wilde, E. et al.: A resource oriented architecture for the Web of Things. In *2010 Internet of Things (IOT)*, pp. 1–8, 2010.

[Guinard et al. 2011] Guinard, D., Trifa, V., Mattern, F., Wilde, E.: From the Internet of Things to the Web of Things: Resource-oriented architecture and best practices. In *Architecting the Internet of Things*, pp. 97–129. Springer, 2011.

[Guinard et al. 2016] Guinard, D., Trifa, V.: *Building the Web of Things: with examples in node.js and raspberry pi*. Manning Publications Co., 2016.

[Harrand et al. 2016] Harrand, N., Fleurey, F., Morin, B., Husa, K.: ThingML: a language and code generation framework for heterogeneous targets In *ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pp. 125–135, 2016.

[Hightower et al. 2017] Hightower, K., Burns, B., Beda, J.: Kubernetes: up and running: dive into the future of infrastructure.*O'Reilly Media*, Inc. 2017

[Jaramillo et al. 2016] Jaramillo, D., Nguyen, D. V., Smart, R.: Leveraging microservices architecture by using Docker technology. In *SoutheastCon*, Norfolk, VA, pp. 1–5 2016.

[Kabalci 2019] Kabalci, Y.: IEEE 802.15. 4 technologies for smart grids. *Smart Grids and Their Communication Systems*, pp. 531–550. Springer, Singapore, 2019.

[Khanda et al. 2017] Khanda, K., Salikhov, D., Gusmanov, K., Mazzara, M., Mavridis, N.: Microservice-based IoT for smart buildings. In *31st International Conference on Advanced Information Networking and Applications Workshops*, pp. 302–308. IEEE, 2017.

[Koulamas et al. 2018] Koulamas, C., Kalogeras, A.: Cyber-physical Systems and Digital Twins in the Industrial Internet of Things. *Computer*, 51(11), 95–98, 2018.

[Mena et al. 2019] Mena, M., Criado, J., Iribarne, L., Corral, A.: Digital Dices: Towards the Integration of Cyber-Physical Systems merging the Web of Things and Microservices. In *9th Int. Conf. on Model and Data Eng. (MEDI'2019)*, pp. 195-205, Springer, 2019.

[Mozilla 2021] Mozilla IoT WebThings. https://iot.mozilla.org/. Acc.: 2021-01-07.

[Ngu et al. 2016] Ngu, A., Gutierrez, M., Metsis, V., Nepal, S., Sheng, Q.: IoT middleware: A survey on issues and enabling technologies. *IEEE Internet Things J*, 4(1), 1–20, 2016.

[Rose et al. 2012] Rose, L. M., Matragkas, N., Kolovos, D. S., Paige, R. F.: A feature model for model-to-text transformation languages. In *4th International Workshop on Modeling in Software Engineering*. IEEE Press, pp. 57–63, 2012.

[Sharaf et al. 2019] Sharaf, M., Abusair, M. et al.: Modeling and code generation framework for iot. In *International Conference on System Analysis and Modeling*, pp. 99–115, 2019.

[Shetty 2021] Shetty S.: How to Use Digital Twins in Your IoT Strategy. https://gtnr.it/2FFU4al. Acc.: 2021-01-24.

[Small et al. 2017] Small, N., Akkermans, S., Joosen, W., Hughes, D.: Niflheim: An end-to-end middleware for applications on a multi-tier IoT infrastructure. In *IEEE 16th Int. Symposium on Network Computing and Applications (NCA)*, pp. 1–8. IEEE, 2017.

[Steinmetz et al. 2017] Steinmetz, C., Schroeder, G. et al.: Ontology-driven IoT code generation for FIWARE. In *IEEE 15th Int. Conf. on Industrial Informatics (INDIN)*, pp. 38–43, 2017.

[Sun et al. 2017] Long S., Yan L., Memon, R. H.: An open IoT framework based on microservices architecture. *China Communications*, 14(2), 154–162, 2017.

[Tuegel et al. 2011] Tuegel, E., Ingraffea, A., Eason, T., Spottswood M.: Reengineering aircraft structural life prediction using a digital twin. *International Journal of Aerospace Engineering*. vol. 2011, Article ID 154798, 14 pages, 2011.

[Vallecillo et al. 2012] Vallecillo, A., Gogolla, M., Burgueño, L., Wimmer, M., Hamann, L.: Formal Specification and Testing of Model Transformations. In *Int. School on Formal Methods for the Design of Computer, Communication and Software Systems* pp. 399–437. Springer, 2012.

[Vresk and Čavrak 2016] Vresk, T., Čavrak, I.: Architecture of an interoperable IoT platform based on microservices. In *39th International Convention on Information and Communication Technology, Electronics and Microelectronics*, pp. 1196–1201, 2016.

[WoT 2021] W3C: Web of Things. https://www.w3.org/WoT/. Acc.: 2021-01-05.