

Enabling IoT Platform Interoperability Using a Systematic Development Approach by Example

Michael Schneider, Benjamin Hippchen, Sebastian Abeck
Research Group Cooperation & Management
Karlsruhe Institute of Technology (KIT)
Zirkel 2, 76131 Karlsruhe, Germany
(michael.schneider | benjamin.hippchen | abeck)@kit.edu

Michael Jacoby, Reinhard Herzog
Department Information Management and Production Control
Fraunhofer IOSB
Fraunhoferstr. 1, 76131 Karlsruhe, Germany
(michael.jacoby | reinhard.herzog)@iosb.fraunhofer.de

Abstract—Today, the IoT landscape consists of a large number of vertical IoT platforms that are rarely interconnected. To enable creation of applications across platforms and domain boundaries interoperability needs to be established between IoT platforms. In this paper we present how this task can be simplified by utilizing a systematic software development process based on behavior- and domain-driven development. This process is illustrated using an example that uses the open source IoT interoperability framework symbIoTe to connect two indoor navigation platforms. We show that developers can actually profit from this approach but existing IoT interoperability frameworks are still cumbersome to use.

Keywords—Internet of Things, IoT, interoperability, semantic interoperability, behavior-driven development, domain-driven design

I. INTRODUCTION

The landscape of Internet of Things (IoT) platforms is constantly growing. Most of these platforms are highly specialized on a specific domain or even an application. This results in the IoT landscape being fragmented into many vertical IoT silos that are rarely interconnected. However, the key feature of the IoT is to interconnect things regardless of the way they are physically connected which means that interoperability should not only be possible within platforms but also across platforms.

To enable interoperability across IoT platforms multiple levels of interoperability need to be addressed. The two most important ones are technical interoperability and semantic interoperability. Technical interoperability enables the exchange of data. This is well addressed by existing communication protocols and standards. Thus, it can be treated as more or less solved. However, besides exchanging data the platforms must also be able to understand it. The ability to understand exchanged data is referred to as semantic interoperability. Enabling semantic interoperability can be far more complex than technical interoperability and is still an active area of research.

In this paper, we present how the process of making two existing IoT platforms interoperable using the open source IoT interoperability framework symbIoTe can be simplified by applying the software development approaches behavior-driven development (BDD) and domain-driven design (DDD).

The remainder of this paper is structured as follows. Section II provides background information on BDD, DDD and

the used symbIoTe framework as well as related work on other IoT interoperability systems. Furthermore, the example used throughout the paper is introduced in this section. Section III and Section IV present how the software development approaches and processes are applied to the example in the analysis and the design phase. In Section V is shown how (semantic) interoperability is implemented for our example using the symbIoTe interoperability framework. The paper closes with conclusions in Section VI.

II. BACKGROUND & RELATED WORK

A. Software Development Approaches

The development process we apply in our case study is based on behavior-driven development (BDD) and domain-driven design (DDD). Each of the two approaches covers a different aspect of the application. One first step to merge both approaches is given in [1]. The authors classify them into a widely accepted software engineering approach from Brüggel et al. [2].

BDD could be seen as a further development of test-driven development (TDD) [3]. TDD tries to determine the correctness of an application with executable acceptance tests, which are written in the chosen programming language [4]. This is the first difference to BDD, which specifies the acceptance tests in the business readable, domain specific language Gherkin [3]. Gherkin uses the common speech and predefined keywords for defining its features. This allows practitioners of BDD to create features directly with the customer, as they are “understood by everyone” [3]. Furthermore, the philosophy of BDD is progressing the application from the outside to the inside [3]. The most visible functionality is first specified as a feature and then implemented directly. During the implementation, new functions are discovered, defined as features and implemented. North characterizes this approach as “code-by-example” [5].

According to Evans, the main reason why applications do not meet the customer’s expectations and needs is the lack of the developers knowledge of the customer’s domain [6]. To address this problem, Evans introduced the DDD approach with various patterns and principals in his book “Domain-Driven Design: Tackling Complexity in the Heart of Software”. With DDD, the customer’s domain is analysed

and the results, the so-called domain knowledge, are stated in a domain model. By establishing a “ubiquitous language”, the development team and the customer speak the same language. An essential principal of DDD is that the source code of the application reflects the structure of the domain model.

B. symbIoTe

symbIoTe (symbiosis of smart objects across IoT environments) [7] is an H2020 EU project and provides an open source interoperability framework for collaboration and federation of IoT platforms. It provides interoperability on four so-called *interoperability levels* (L1-L4): syntactic and semantic interoperability (L1), platform federations (L2), dynamic smart spaces (L3) and roaming devices (L4). In our example we are going to make the platforms only L1-compliant as L1 already covers our needs (syntactic and semantic interoperability). Figure 1 shows a high-level

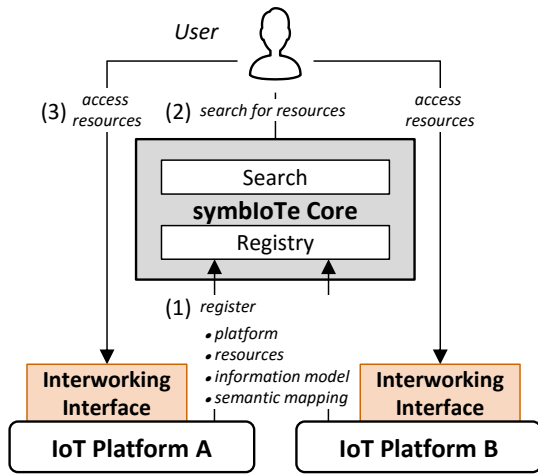


Fig. 1. High-level overview of symbIoTe L1 functionality.

overview of symbIoTe’s L1 functionality. The two major building blocks are the symbIoTe Core (depicted in grey) and the Interworking Interface (depicted in orange). The Core acts as an IoT search engine where platforms can register their resources (1) and users (meaning users of symbIoTe, e.g. mobile apps, websites, IoT platforms, backend services, etc. rather than end-users) can search for these resources (2). The resource access (3) is done directly between user and platform so that platforms can maintain complete control over their data.

The Interworking Interface defines a uniform API for resource access, authentication and authorization across platforms and has to be implemented by each platform. In the following, we present how syntactic and semantic interoperability are achieved in symbIoTe.

Syntactic interoperability is about exchanging data. In symbIoTe, syntactic interoperability is achieved by requiring all platforms to implement the Interworking Interface as a common API for resource access.

Semantic interoperability is about understanding the exchanged data. Three different types of semantic interoperability

can be identified: (semantic) interoperability by chance, by standardization and by mapping. Interoperability by chance means that an interoperability framework allows the usage of any information model and the platforms are only interoperable if they, by chance, use the same one. This would be comparable to an API where each parameter and return value is defined as a set of key-value-pairs but the possible values for the keys are not defined. In this case, everybody could use the API but nobody would know exactly how to use it to be interoperable to other implementations. Interoperability by standardization means that there is exactly one agreed-upon or standardized information model that all platforms must use. This is currently the default for API definitions. Almost all existing interoperability frameworks and systems support either one or both of these types of semantic interoperability. However, in the highly diverse and dynamic environment of IoT this only works within some scenarios but not as a general way to enable interoperability between any IoT platform.

The most powerful type of semantic interoperability is called interoperability by mapping and allows different platforms to use different information models. Semantic interoperability between the platforms is established only by semantic mappings between their information models. However, this comes at the price that the mappings have to be defined and data has to be translated between the different models.

According to [8], there are multiple possible approaches on how to achieve semantic interoperability. symbIoTe uses the approach called *Core Information Model with Extensions*, which supports all three types of semantic interoperability.

In symbIoTe, each platform can describe the resources it offers using its own information model called Platform-Specific Information Model (PIM). However, this model cannot be chosen completely freely by the platform but must be an extension of the Core Information Model (CIM) that is defined by symbIoTe shared between all platforms. The CIM is designed to be as explicit as needed to ensure a minimal level of out-of-the-box interoperability, but at the same time to be as minimalistic and abstract as possible giving platforms the freedom to adapt it to their needs. For example, the CIM defines abstract classes like Sensor, Actuator, Service or Location with minimal properties like id and name. When a platform needs a more sophisticated class hierarchy or additional properties it can define them in its PIM.

This approach supports all three types of semantic interoperability. Interoperability by standardization is supported as all platform share the CIM as smallest denominator ensuring some minimal interoperability between all platforms out-of-the-box (i.e. without having to define semantic mappings). Furthermore, platforms can agree to use the same (standardized) PIM which makes them interoperable without doing anything further not only regarding the CIM but the whole shared PIM. If they use the same PIM by chance, e.g. because it is best practice or de-facto standard in their domain, they also gain full semantic interoperability in terms of that PIM. Additionally, semantic interoperability by mapping will also be supported by symbIoTe. For this to work, symbIoTe needs

formally defined mapping definitions describing how two PIMs relate to each other.

These mappings are used in multiple ways within symbIoTe. First, a platform can store mappings between its PIM and other PIMs in the symbIoTe Core. This way, they are available to the Core, which will utilize them to do query re-writing which allows finding resources across different mapped PIMs. Additionally, the symbIoTe client library will utilize them for data translation, allowing to access data from another platform in a (PIM-) transparent way. This means, that one platform can access another platform as if it would use the same PIM as long as there is mapping between both PIMs registered in the Core.

At the time of writing, semantic mapping functionality in symbIoTe is still under development. Details on the development status of semantic mapping and the definition of the mappings are presented in Section V.

C. Other IoT interoperability frameworks

There are a lot of research groups and standards addressing the problem of cross-platform interoperability in the IoT context. These are for example the IRTF Thing-to-Thing Resource Group (T2TRG), W3C Web of Things Working Group (WoT), SensorThingsAPI from the Open Geospatial Consortium (OGC), iot-schema.org, oneM2M and many more. However, these activities are more fundamental whereas we will focus on more hands-on projects that are either available right now or are expected to be available soon.

Therefore, in this section we will present three projects, which are part of the IoT-European Platforms Initiative (IoT-EPI), all addressing the problem of IoT platform interoperability. IoT-EPI is a European initiative bringing together seven EU-funded research and innovation projects (including symbIoTe) in the area of IoT platform development.

BIG IoT

Main objective of the BIG IoT project [9] (**B**ridging the **I**nteroperability **G**ap of the **I**o**T**) is to create an open marketplace for IoT platforms and services. The marketplace concept of BIG IoT is very similar to the symbIoTe Core and also the whole processing of making a platform (L1-) compliant to the system as well as registering and searching resources. BIG IoT uses the W3C WoT ThingDescription to semantically describe resources and allows the usage of any information model. However, they do not provide any support to enable interoperability between platforms using different information models.

bIoTope

The bIoTope project [10] (**B**uilding an **I**o**T** **O**pen **I**nnovation **E**cosystem **F**or **C**onected **S**mart **O**bjects) addresses interoperability with a Systems-of-Systems approach. Everything (e.g. apps, devices, platforms, gateways, non-IoT application and services) has to have a wrapper that exposes the resource's data through the Open Message Interface (O-MI) using the Open Data Format (O-DF). Just like BIG

IoT, bIoTope allows the use of any information model but does not provide any tool support for establishing semantic interoperability.

INTER-IoT

The INTER-IoT project [11] (**I**nteroperability of heterogeneous **I**o**T** platforms) addresses interoperability on five levels; device, network, middleware, application and data, and semantics. Besides symbIoTe, INTER-IoT is the only project in the IoT-EPI explicitly addressing semantic mapping and providing a tool set to make use of these mappings. However, in INTER-IoT, semantic mapping is not done directly between two different platform-specific information models but always from a platform-specific information model to a common, shared model [12].

D. Introduction of the Example

Figure 2 shows a schematic representation of the components and their deployment of the running example. The initial position is: there are two existing IoT platforms, Platform Campus A and Platform Campus B, one deployed at the KIT, the other at Fraunhofer IOSB premises and both providing a mobile app supporting their offered services. The goal is to enable interoperability between the platforms so that users of Campus A can use their existing application (e.g. for searching a room) for the same intention when visiting Campus B. In this case, the application of Campus A requires data (including IoT data, such as the current location of the user) of Campus B. However, as the two platforms may use different information models (called PIMs in symbIoTe) the exchanged data might not be understood by the application. Therefore, the data needs to be provided in a way so that the corresponding application can process the data. A transformation of the information model (like schematic mapping) is needed. This transformation is done by a component of the adapter, as shown in Figure 2. To establish the connection to symbIoTe, Campus A and Campus B have to implement the symbIoTe adapter. For accessing resources from Campus B, the resource access proxy (RAP) of Campus B must be known. This adapter offers the required data from Campus B. Therefore, the adapter registers the platform and provides the information model in the so-called resource description format (RDF) [13] to symbIoTe Core. Afterwards, the resources from Campus B can be searched and found in the Core and are accessible via the interworking interface. The Interworking Interface is REST-based [14]. The backend of Campus B provides the needed data through a plug-in that communicates with the RAP. The data that Campus A requested from Campus B is received by the symbIoTe client (which is part of the adapter) of Campus A. In this stage, the received data is semantically mapped for the needs of the backend that Campus A provides for their application. On the left side, Campus A has to implement the necessary adapter for enabling symbIoTe. On the right side, Campus B also needs a connection to symbIoTe and has to implement an adapter as well. One challenge of the semantic mapping is that the domains of Campus A and Campus B are

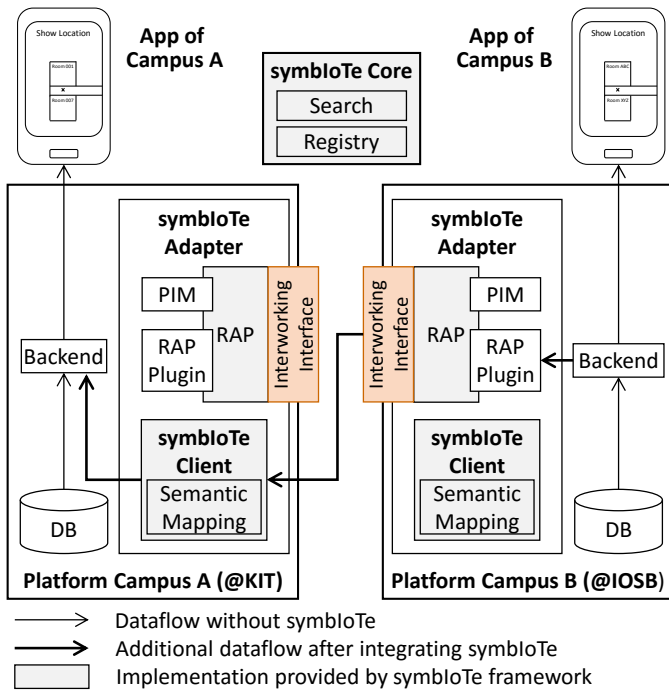


Fig. 2. Schematic representation of components and their deployment for the example use case including the data flow with and without symbIoTe.

not the same. Campus A provides a beacon-based navigation service while Campus B offers a room reservation service. The consequence is that the information models are not the same. However, the models have several overlaps that can be utilized for the application, e.g. both are determining the current location.

III. ANALYSIS PHASE: FEATURE DESCRIPTION

As a member of Campus A I want to use my application to show my current location on Campus B. This is an important requirement in terms of interoperability the desired application has to fulfil. In order to create a cross-domain application, it is necessary to discuss the need of the interoperability as a requirement. To address this need behavior-driven development is used for analysing and specifying the requirements. Therefore, the requirements are specified as features. This is the first part of the systematic development process. Figure 3 shows the feature that describes the desired goal of the application. A BDD template is used to specify

1. Feature: Show my location on Campus B
 2. As a member of Campus A
 3. I want to use my well-known application
 4. In order to determine my current location on Campus B
-
5. Scenario: Show my location on Campus B
 6. Given I am at Campus B
 7. And a beacon from Campus B is available
 8. When I open the "Current Location" page
 9. Then my current location on Campus B should be displayed

Fig. 3. Campus Interoperability Feature

the feature shown in Figure 3. It consists of two parts; a feature description part [15] describing the business value of the feature and a scenario description part [3], which is written in natural language. An advantage of the scenarios is that developers can easily understand and discuss the specified features with the stakeholders. The scenarios describe how the application works. In addition, scenarios describe the acceptance criteria of a feature that allows the automated testing of the requirement. In order to test the feature shown in Figure 3, the step definitions have to be implemented. Before the written tests can pass, the functionality has to be implemented. To be able to test the feature shown in Figure 3, there must be a connection to symbIoTe. Line 7 of the scenario given in Figure 3 makes it clear that a beacon from Campus B must be available. This step delivers an important hint. Without a connection to symbIoTe, the test cannot pass because the needed data cannot be acquired from Campus B. Thus, the systematic development approach delivers important clues for the developers in terms of interoperability and the developer knows what needs to be implemented. Features also contain scenarios that cover errors that may occur. Therefore, the features provide important clues for the developers in order to allow cross-domain communication. To test a feature even further, each feature contains more than one scenario. The set of scenarios of a feature should also contain scenarios that describe what happens in an error situation.

In addition to the features which concern the cross-domain integration, there is also a need for features that describe the functionality of the application. These features are specified according to the features concerning interoperability. For example, the navigation application needs to determine the current position of a user and the location of the user inside a building should be displayed. The terms used in the feature description, provide important domain knowledge and have an influence on the domain model. Therefore, the specified features function as input for the modelling phase.

IV. DESIGN PHASE: MODELLING AND INTEGRATION OF THE MODELS

To enable semantic interoperability each platform needs to provide a formally defined model of the domain. In case of Campus A where we applied BDD and DDD, we automatically get this model as part of the systematic development process. For Campus B, the model was created manually based on existing class diagrams. In this section, we first introduce the two different models and then analyse their differences and how they can be aligned. Please note that the models have been simplified to better illustrate the example.

A. Model of Campus A

The creation of the domain model from Campus A is based on the features. A feature specifies parts of the business logic from the viewpoint of a user. In addition, each feature specifies parts of the application logic of the software system. A part of the business logic is the domain logic, which is application agnostic. Therefore, a feature contains relevant information

about the domain, which is relevant for understanding the domain. The approach to understand the domain and the functionality related to the domain is achieved by reading every line of the feature and its scenarios and identifying the presumably relevant terms. Each relevant term becomes a part of the ubiquitous language, an important concept of DDD [6]. With each analysed feature, additional and relevant terms are identified. For example, the feature shown in Figure 3 led to the terms location and building. In addition, the relationships between the terms can be derived from the feature, e.g. a beacon is at a location. Further features, knowledge crunching, and more insight extend the domain model to the resulting model, as shown in Figure 4.

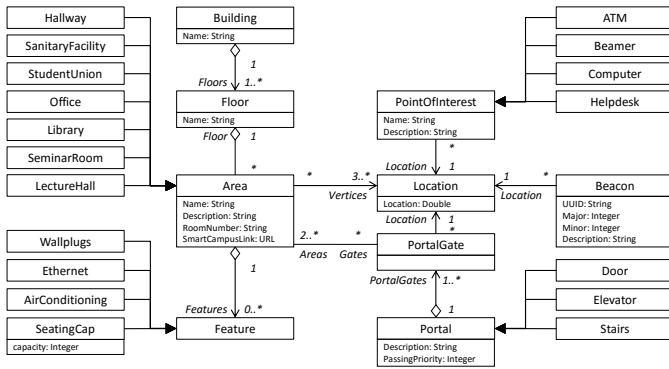


Fig. 4. Domain Model of Campus A

The excerpt from the domain model of the navigation platform describes the relations between the domain objects. The main concept of the model is the *Area*. Three or more *Locations* define the vertices of an *Area*. Each *Floor* is divided into several *Areas* while a *Floor* is part of a *Building*. An *Area* can be a specific type like *Hallway* or *Office*. For navigation purposes, each *Area* has at least one *PortalGate* at a specific *Location*. Two or more *PortalGates* are connected by a *Portal*. To determine the position of a user, beacons are placed at a specific *Location* in an *Area*. For this purpose, it is necessary to distinguish the beacons. Therefore, each beacon has a universally unique identifier (UUID) which is displayed as attribute in the model.

B. Model of Campus B

Figure 5 depicts the domain model of Campus B using the EduCampus platform. The main concepts are *BleBeacons* that are attached to a *Thing* and *BeaconDetection* which represents events generated when a *User* was close to a beacon at a certain date and time. Users can also create *Reservations* for a room and time interval, including *Catering Requests*. A *Thing* can be either a *Room* or a *MoveableThing*, e.g. inventory. A *Room* has the properties *capacity* and *roomNo*, can have multiple *Features* like a projector or a whiteboard and can contain multiple *Workspaces*.

C. Differences and Integration of the models

Although both models cover more or less the same domain, they have quite different views of it. This is caused by different

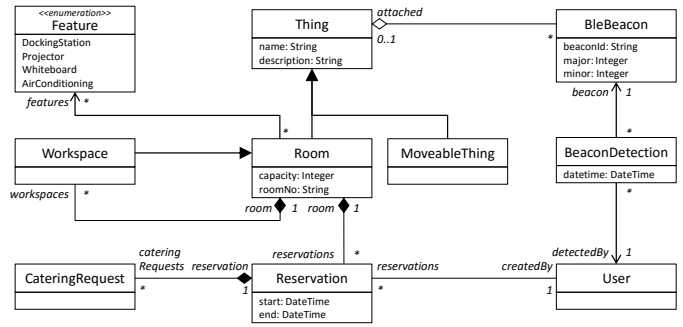


Fig. 5. Domain Model of Campus B.

needs of the existing applications resulting in different levels of detail of the model and by design decisions like using ray tracing or nearest-neighbour for location. However, they both provide information about areas/rooms and the functionality to identify the position of a user based on the beacons in range. To share this functionality between the platforms, both expose a service called *getPosition* taking some beacon information as input and returning the position. However, they use the corresponding classes of their domain model to describe the input and return type. Therefore, the service definition for Campus A is $getPosition(Beacon[]) \rightarrow Area$ and for Campus B $getPosition(BleBeacon[]) \rightarrow Room$. This reduces the problem of semantic interoperability to mapping the concepts *Area* and *Room* as well as *Beacon* and *BleBeacon*. This mapping has to include all the properties and references of these concepts that are present in both models.

Mapping *Beacon* and *BleBeacon* is straightforward by renaming the shared properties ($UUID \leftrightarrow beaconId$, $Major \leftrightarrow major$, $Minor \leftrightarrow minor$) and dropping the optional property *Description*. To map *Area* and *Room* we start again with renaming the shared properties ($Name \leftrightarrow name$, $Description \leftrightarrow description$ and $RoomNumber \leftrightarrow roomNo$). The rest of the mapping, covering features of an area/room and the seating capacity, is more complex as it involves relations between objects. If an *Area* has a *Feature* of type *SeatingCap* we map its property *capacity* to the property *capacity* of *Room* and the other way round. Further, we map the only shared type of feature from an instance in the model of Campus A to the corresponding enumeration value in the model of Campus B.

V. IMPLEMENTATION PHASE: CURRENT STATE

To become symbIoTe L1-compliant a platform has to implement the Interworking Interface. To ease the development process, symbIoTe already provides a microservice-based Java implementation of the Interworking Interface that only needs to be slightly adapted to the platform¹. The PIM needed for registration of the platform to the Core can be extracted easily from the domain model, which we created through the design phase of the systematic development process. Creating an RDF/OLW2-based ontology of the PIM, which is required by symbIoTe, is now only a matter of minutes.

¹<https://github.com/symbiote-h2020/SymbioteCloud>

To enable interoperability across the two different PIMs we need to define a semantic mapping. In symbIoTe, semantic mappings are defined using a proprietary domain specific language. They consist of an optional set of prefix declarations, an optional set of transformation definitions (allowing to define transformation functions in JavaScript) and some mapping rules. Figure 6 shows an example mapping definition with some prefix definitions and a single rule. Currently, these mappings have to be defined manually by a domain expert. However, to ease usability, they should be (partially) auto-generated in the future based on existing algorithms for ontology matching.

```

1. BASE <http://iosb.fraunhofer.de/ilt/ontologies/educampus#>
2. PREFIX kit: <http://cm.kit.edu/SmartCampus/DomainModel#>
3. PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

4. RULE
5.   CONDITION
6.     CLASS :Room
7.       :name TYPE xsd:string
8.     AND :description TYPE xsd:string
9.     AND :roomNo TYPE xsd:string
10.    AND :capacity TYPE xsd:integer
11.  PRODUCTION
12.    CLASS kit:Area
13.      kit:Name VALUE REFERENCE :name
14.    AND kit:Description VALUE REFERENCE :description
15.    AND kit:RoomNumber VALUE REFERENCE :roomNo
16.    AND kit:hasFeature TYPE CLASS kit:SeatingCap
17.      kit:capacity VALUE REFERENCE :capacity

```

Fig. 6. Example semantic mapping definition.

It is to notice, that semantic mapping is still an active research area. Nevertheless, symbIoTe is trying to progress the state-of-the-art by providing a proof-of-concept implementation covering query translation and data transformation based on mappings. At the time of writing, the semantic mapping functionality in symbIoTe is still under development². The basic functionality – mapping language including parser, query re-writing and data transformation – is working but not yet integrated into the Core and the symbIoTe client library.

It is planned to release a finished version in July 2018.

VI. CONCLUSION

In this paper, we presented a real-life example of how to establish interoperability between two IoT platforms dealing with BLE-based indoor localization. We used symbIoTe as an open source IoT interoperability framework and showed that using a systematic software development based on BDD and DDD is helpful in this process. This is because semantic interoperability is a very tough problem (besides when standardization is used) and BDD and DDD implicitly output a domain model, which is needed to achieve semantic interoperability. This is a major advantage of using this kind of systematic software development process in such scenarios because a formally defined information model is always needed when trying to establish semantic interoperability and surprisingly very few IoT platforms already provide this. Furthermore, lots

of IoT platform providers and owners seem to have trouble when being asked to create such a model.

IoT interoperability is a growing area of research and will even grow faster in the future as the number of internet-connected devices keeps increasing. Even though it is currently addressed by multiple research projects as described in Section II-C, further research and standardization is needed to provide easy-to-use interoperability libraries and frameworks to be used by non-expert programmers.

ACKNOWLEDGMENT

This work is supported by the H2020 symbIoTe project, which has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 688156.

REFERENCES

- [1] B. Hippchen, P. Giessler, R. Steinegger, M. Schneider, and S. Abeck, “Designing microservice-based applications by using a domain-driven design approach,” in *International Journal on Advances in Software*. IARIA, 2017, pp. 432–445.
- [2] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering Using UML, Patterns and Java-(Required)*. Prentice Hall, 2004.
- [3] M. Wynne, A. Hellesoy, and S. Tooke, *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2017.
- [4] K. Beck, *Test-Driven Development: By Example*. Addison-Wesley Professional, 2003.
- [5] D. North. (2009) BDD & DDD. QCon London 2009. URL: <https://www.infoq.com/presentations/bdd-and-ddd> [retrieved: 2017.11.30].
- [6] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004.
- [7] S. Sourso, I. P. Žarko, P. Zwickl, I. Gojmerac, G. Bianchi, and G. Carozzo, “Towards the cross-domain interoperability of iot platforms,” in *2016 European Conference on Networks and Communications*, 2016.
- [8] M. Jacoby, A. Antonić, K. Kreiner, R. Łapacz, and J. Pielorz, “Semantic interoperability as key to iot platform federation,” in *International Workshop on Interoperability and Open-Source Solutions*. Springer, 2016, pp. 3–19.
- [9] A. Bröring, S. Schmid, C.-K. Schindhelm, A. Khelil, S. Käbisch, D. Kramer, D. Le Phuoc, J. Mitic, D. Anicic, and E. Teniente, “Enabling iot ecosystems through platform interoperability,” *IEEE software*, vol. 34, no. 1, pp. 54–61, 2017.
- [10] bIoTope project. [Online]. Available: <http://www.biotope-project.eu/>
- [11] G. Fortino, C. Savaglio, C. E. Palau, J. S. de Puga, M. Ganzha, M. Paprzycki, M. Montesinos, A. Liotta, and M. Llop, “Towards multi-layer interoperability of heterogeneous iot platforms: The inter-iot approach,” in *Integration, Interconnection, and Interoperability of IoT Systems*. Springer, 2018, pp. 199–232.
- [12] M. Ganzha, M. Paprzycki, W. Pawłowski, P. Szmaja, and K. Wasielewska, “Towards semantic interoperability between internet of things platforms,” in *Integration, Interconnection, and Interoperability of IoT Systems*. Springer, 2018, pp. 103–127.
- [13] O. Lassila, R. R. Swick *et al.*, “Resource description framework (rdf) model and syntax specification,” *W3C*, 1998.
- [14] R. T. Fielding, “Rest: architectural styles and the design of network-based software architectures,” *Doctoral dissertation, University of California*, 2000.
- [15] D. North *et al.*, “What’s in a story,” 2016. [Online]. Available: <https://dannorth.net/whats-in-a-story/>

²<https://github.com/symbiote-h2020/SemanticMapping/tree/develop>