A Mapping Language for IoT Device Descriptions

1st Fabian Burzlaff
Institute for Enterprise Systems
University of Mannheim
Mannheim, Germany
burzlaff@es.uni-mannheim.de

2nd Maurice Ackel
Lohrtalweg 100
74821 Mosbach
Mosbach, Germany
maurice.ackel@t-online.de

3rd Christian Bartelt
Institute for Enterprise Systems
University of Mannheim
Mannheim, Germany
bartelt@es.uni-mannheim.de

***Abstract*—Component models for IoT devices regain popularity. As more and more devices must be semantically connected within IoT platforms, digital abstractions for these devices are needed. For this purpose, textual device descriptions which encapsulate device-specific characteristics are a suitable candidate. Such component descriptions formally describe a device's information model as well as the offered functionality in a standardized way. However, smart IoT platforms mainly solve user goals by composing various IoT devices in a suitable manner. Current IoT descriptions, such as Eclipse Vorto do not address this need at all. In this paper, we introduce a formal mapping language that allows to capture functional interaction semantics already during device integration time. Our evaluation shows that only few mapping elements are needed to define functional mappings between operations as well as to capture the underlying communication pattern.**

***Index Terms*—Internet of Things, Formal Mapping Language, Semantic Interoperability**

I. Introduction

Context: Currently, "Things" start to get increasingly smart as they get equipped with micro controllers and networking hardware. As a consequence, the so-called *Internet of Things* (IoT) emerged over the last years. The definition of this term still varies and reaches from "a galaxy of solutions somehow related to the world of intercommunicating smart objects" [1] to "a network that connects uniquely identifiable 'Things' to the Internet" [2]. Overall, IoT specifies networks of devices which communicate with each other and is a well-known topic in IoT research communities [3], [4].

Problem: Due to development of IoT, the well-known question whether a provided interface fits a required interfaces regains importance progressively. The underlying matching problem has already been extensively researched in communities such as component-based software development [5]. In order to automate component composition, current scientific approaches rely on the assumption that a common framework-language as well as top-down standardized ontologies are used by all parties involved within the IoT integration domain [6], [7]. However, this is naive to be assumed as standardization activities are too slow for technology innovation cycles which are getting faster and faster [8]. At the moment, this results in programming semantically identical software adapters that utilize syntactically different interfaces over and over again.

This work was supported by the BMVI research projects "VanAssist" and "xDataToGo" (Support Code: 16AVF2139F and 19F2048D)

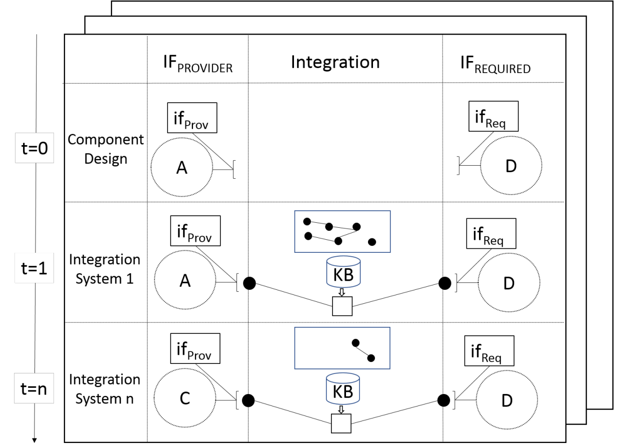

Fig. 1: Knowledge-driven Architecture Composition

Need: As a consequence, integration knowledge between provided and required interfaces is only persistent within software adapters and is currently not being captured, shared and reused [9]. Therefore, a novel engineering method called "Knowledge-driven Architecture Composition" (KDAC) for bottom-up IoT device integration was introduced [10]. The novelty of this method lies within the incremental formalization of semantic integration knowledge for IoT devices in addition to programming iterative software adapters. Over time, shared semantic integration knowledge can be utilized as a basis to generate adapters for the underlying IoT architecture [11].

Solution: In this paper, we will introduce a domain-specific mapping language and tool support that can be used to express mappings and communication patterns for Vorto device descriptions. This language is a first technical step towards operationalizing knowledge-driven architecture composition (see Figure 1) for broader application scenarios. Therefore, we will introduce and explain a suitable grammar. As a last contribution, we will evaluate the proposed language in a synthetic example to show that it is expressive enough for simple scenarios.

II. IoT Device Integration

At the moment, IoT solution developers must necessarily know the underlying information and service model of common IoT communication protocols (e.g. MQTT, COAP or OPC UA). Based on their domain experience, they can

usually map between standardized device descriptions (e.g. Eclipse Vorto) and the use case specific payload which is transmitted by a communication protocol. Nevertheless, standardized device descriptions do not necessarily result in less integration effort. This is due to the circumstance that current IoT systems (e.g. Apache iota or Eclipse Kura) are usually not able to realize a complex user goal by invoking only one service component. Instead, such user goals are fulfilled by composing multiple service's components and information elements in an ordered sequence. Hence, solving the matching and mapping problem between required and provided interface requires increasing integration effort if the amount of required devices increases.

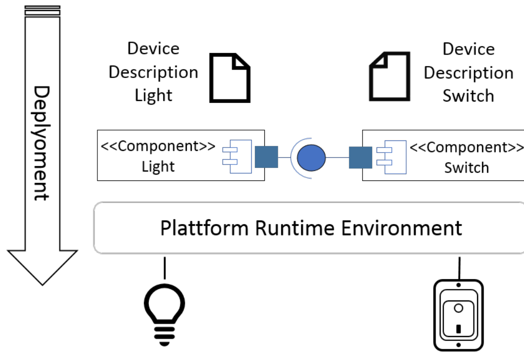

Fig. 2: Exemplary IoT Use Case

Let's consider the example depicted in Figure 2. In this case, a dimmable light bulb (left side) should be connected to a switch without dimming functionality (right side). The goal of the integration process is to toggle the light from on to off and vice versa whenever the switch is toggled. To perform this integration task, first, a software engineer needs to analyze the functionalities and the interfaces of both devices. The offered functionalities in our example can be found in the so-called *function blocks* where *configuration*, *status*, *events* and *operations* of the *Switch* (see Listing 1) and the *Light* (see Listing 2) are defined.

```
[...]
description "A simple boolean switch"
category Generic
functionblock SwitchFunctionality {
  configuration {
    defaultState as boolean "Default state of the
    switch"
  }
  status {
    currentState as boolean with {readable: true}
    "Current state of the switch"
  }
  [...]
  events {
    stateChanged {
      oldState as boolean with {readable: true} "Old
     state of the switch"
      newState as boolean with {readable: true} "New
     state of the switch"
    }
  }
  operations {
    set(state as boolean) "Sets the state of the
    switch"
    toggle() "Toggles the state of the switch"
  }
}
```

Listing 1: Vorto function block of the SwitchFunctionality

Furthermore, several other factors need to work together. This means, that 1) a *stateChanged event* should be fired as soon as the wall switch is toggled. Then 2) this **boolean** *stateChanged event*, originating from the switch must be translated to the *setValue (....)* operation offered by the dimmable light bulb. This results in an operation invocation with the *MAX* value of "1.0" to turn on the light. The required integration knowledge can then be deployed to an IoT platform in the form of a software adapter (e.g. transformation rule). Nonetheless, current component models for IoT architectures do not allow for the explicit definition of such knowledge.

```
[...]
description "A double-precision dimmer"
category Generic
functionblock DimmerFunctionality {
  configuration {
    optional defaultValue as double <MIN 0.0, MAX
    1.0>
    "The default value in percent"
  }
  status {
    currentValue as double with {readable: true}
    <MIN 0.0, MAX 1.0> "The current value in percent
    "
  }
  [...]
  operations {
    setValue(value as double<MIN 0.0, MAX 1.0>) "
    Sets the value in percent"
  }
}
```

Listing 2: Vorto function block of the DimmerFunctionality

III. An Architecture for Formalizing IoT Integration Knowledge

In this work, the main objective of a mapping language – for the purpose of device integration – is to formalize knowledge that is required to generate an adapter between integration endpoints (see System Architecture in Figure 3). Furthermore, mappings should serve both purposes, a mean to document knowledge in an explicit way and to be machine-readable so that existing mappings can be automatically recommended to the application designer.

A mandatory information of a mapping is a reference to all involved interfaces. In addition, the mapping should be uniquely identifiable (i.e. by an ID). The mapping behavior is defined by the state change, from which source interface element data is queried and to which destination interface element data is passed. Hence, a mapping language must offer the possibility to define the direction of an information flow, as well as the role (i.e. source or destination) an interface plays in such a logical or functional mapping.

Given these basic requirements, additional ones may come along in special cases or because of non-functional requirements. Such additional requirements could be complex

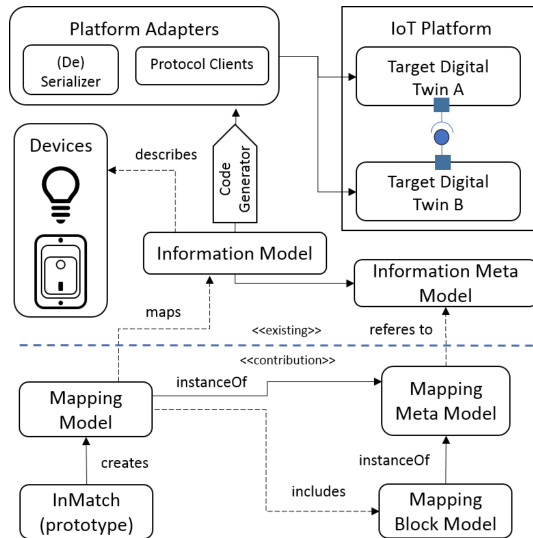

Fig. 3: High-level System Architecture

control structures or simple call-return models when acting in environments that include events. Moreover, additional descriptions of mapping elements could make them easier to understand. However, we will only outline the trigger-action interaction style in this work.

Two main challenges for IoT device integration are interface matching and mapping. Given only their formal device descriptions, the aforementioned language including appropriate tool support should facilitate an easy way to integrate two or more IoT devices within a platform. Therefore, two concepts, the *Matcher* and the *Mapper*, were developed and implemented and will be further explained in the next section. The tool that instantiates both components is called "InMatch" (see Figure 3).

IV. Integrating Vorto IoT Device Descriptions

This section will describe how the concepts of Matcher and Mapper work. Thereby, the focus lies on the mapping language and its application.

A. Eclipse Vorto

Eclipse Vorto is an open source project, which offers a tool set that allows the creation and management of abstract information models for physical devices [12]. The main project target is to be completely technology-independent in order to increase the range of possible usage. In addition, the project provides an online repository where developers and vendors can share their information models to emphasize model reuse. Moreover, Vorto offers a variety of code generators and a code generator API with which any information model can be converted into a code skeleton. Furthermore, Vorto includes a domain-specific language (DSL) which defines the textual device description syntax.

While many DSLs are designed to be compiled or executed by an interpreter, the main objective of Vorto's DSL is to specify constructs which are translated to a specific programming language via a preprocessor (i.e. code generators). This way, once created, developers can use device descriptions on different destination systems by simply using a preexisting code generator or writing a new one.

B. Matcher Component of InMatch

The developed matching approach uses an abstraction layer and heuristics to analyze whether or not interface elements potentially match. From another viewpoint, the Matcher component is acting like a recommender-system that proposes matching interfaces to the user based on prior defined mappings.

In general, matching approaches rely on different information of interface definitions. This information could be semantic and syntactic. While semantic matching can enhance the matching process by using additional information on how an interface element behaves, Vorto only provides syntactic information. In addition, other heuristics, such as data type comparison and element name similarity measurement, were utilized and will be further outlined in the evaluation section. For this method to work, an interface definition at least needs to give information on element names, data types, return types and method signatures of operations.

An adequate semantic model is needed to match different interface elements. Thus, it is important to know which type of interaction (i.e. providing data, receiving data) is supported by an operation. Hence, the developed abstraction layer uses different sets to cluster the elements of an interface according to the way they can be accessed or the way they behave:

readset Contains all elements from which data can be retrieved. In this context, it does not matter whether data is the result of a method call or may be accessed directly as an attribute.

writeset All elements to which data can be provided. Again, independent of the way it is provided.

triggeringset Consists of all elements that are throwable in some sense. This means, that one could listen to them just like to an event and get notified when it is fired.

triggerableset Holds all elements that can be called by an action. An example for this is a method call.

Within these four sets, the first two of these sets adhere to functional interface matching approaches where mappings ensure a correct information exchange between two components. In contrast, the last two sets represent a communication style (i.e. trigger-action) which then uses functional mappings to express the data flow. The underlying idea of building an additional abstraction layer is to add a semantic understanding to an interface. The different sets can already provide information on which elements need to be considered when searching for a potential match.

When working with modeling languages that offer predefined types, data type information can be used to assess whether some information is exchangeable in the first place. Therefore, when testing if readable and writable elements match, the data types of the writable and readable element

must be inspected. Especially data types from the readable element are required to be accepted by the writable element. This analysis can either be done by checking for a perfect match of data types or by testing whether data types are *transformable* into each other. This means, that with a set of transformation operations provided data types can be converted to required data types.

C. Mapper Component of InMatch

The Mapper is used to create mappings between IoT devices and to automatically specify them in the developed mapping language. The mapping language is defined by an ANTLR grammar which makes it possible to create a textual definition of *Mappings* and *MappingBlocks* in an unambiguous way. Furthermore, it allows mapping descriptions to be parsed by a computer program (e.g. to automatically generate an adapter from them).

A Mapping is a composition of MappingBlocks. It is designed to contain all MappingBlocks that are required between certain information models in one particular integration process (cf. Listing 3). This means that for a specific integration, only one Mapping between a certain pair of information models (and thus devices) should exist.

A Mapping contains basic information, like an ID, a namespace and a description, as well as a list of references to the involved information models. Each used information model is given a unique ID, so that it can be referenced in the MappingBlock section. The MappingBlock section contains a list of references to contained MappingBlocks. Each reference also specifies the direction of the MappingBlock, i.e. which information model is the source and which information model is the destination.

```
mapping:
  'id' id
  'namespace' namespace
  'description' description
  'infoModelReferences' '{'
    modelReferenceProperty*
  '}'
  'mappingBlocks' '{'
    mappingBlockReferenceProperty*
  '}'
;
modelReferenceProperty: 'using' modelReference 'as'
    id;
mappingBlockReferenceProperty: 'from' id 'to' id ':'
     qualifiedName|mappingBlock;
```

Listing 3: Grammar for Mapping

1) MappingBlocks: The idea of a MappingBlock is to define the way elements of different interfaces need to interact with each other in order to provide **one** particular functionality or service (see Listing 4). Hence, MappingBlocks map function blocks to each other, regardless of which devices are equipped with the specific function block. Similar to function blocks, MappingBlocks should only define one integration at a time on an atomic level to emphasize their reuse among different projects or device description pairs.

MappingBlocks hold information on which function blocks act as inputs and which as outputs. Furthermore, metadata and identifiers are contained to ensure references within Mappings.

```
mappingBlock:
  'id' id
  'namespace' namespace
  'description' description
  'sources' '{'
    sources
  '}'
  'destinations' '{'
    destinations
  '}'
  (flowBlock | logicBlock)
;
sources : qualifiedName*;
destinations : qualifiedName*;
```

Listing 4: Grammar for MappingBlock

The information on which interface elements should be combined needs to be specified as well. During integration, this depends on the desired functionality and the component interaction pattern. Hence, a mapping language should enable the definition of different interaction patterns. In this work, two types of interactions were implemented. One is an information exchange between function blocks and the other is a trigger-action pattern.

To represent these different types, two specific realizations of a MappingBlock (i.e. the *FlowBlock* and the *LogicBlock* as depicted in Listing 5) were created:

FlowBlock A FlowBlock defines a data flow between two interface elements. Besides the common information contained in the general MappingBlock, the FlowBlock must specify a list of readable input and writable output elements as well as a rule on which and how information should be exchanged. Furthermore, JavaScript is used to define the information transformation rules. This is a design choice as it is expressive enough (e.g. containing conditionals and loops) and provides native support for working with objects and methods. Overall, applicability among various use cases increases. Moreover, many JavaScript engines for integrating predefined script exist and function composition is a first-class citizen.

LogicBlock In comparison to the FlowBlock which does not specify any interaction pattern intrinsically, the LogicBlock represents a trigger-action pattern. This means that it explicitly defines when the action (also specified in JavaScript code) is performed. A LogicBlock has a set of triggering inputs which specify when an action is executed. In addition, a set of writable **or** triggerable outputs are specified as a LogicBlock which is not limited to an information exchange only.

The difference between Flow- and LogicBlocks can be best illustrated by imagining how corresponding adapters would look like. A FlowBlock can be represented by a simple method. This method executes JavaScript code. In contrast, a LogicBlock would be represented as a listener which is added to each trigger. Hence, it is required to define the specific, use

case dependent interaction pattern of a FlowBlock when the adapter is deployed.

```
flowBlock:
  'flowblock:'
  'inputs' '{'
    inputs
  '}'
  'outputs' '{'
    outputs
  '}'
  'code' code
;
inputs: qualifiedName*;
outputs: qualifiedName*;
logicBlock:
  'logicblock:'
  'trigger' '{'
    trigger
  '}'
  'outputs' '{'
    outputs
  '}'
  'action' code
;
trigger: qualifiedName*;
code: '{' (code |.*) '}'
    ;
```

Listing 5: Grammar for Flow- and LogicBlock

D. Evaluation

In order to evaluate the presented language a synthetic integration case was performed by using the implemented tools. This example should show how the semantic analysis of involved interfaces, the interface matching, the low-level mapping (MappingBlocks) and the high-level mapping (Mappings) can be realized by using the InMatch-Protoype (see Figure 4). For the case study, our motivating example (see Figure 2) serves as a basis. Hence, the integration task is still to toggle the state of a light depending on a switch.

The first integration step is to retrieve or create Vorto representations of all IoT devices involved. As a consequence, the Vorto repository is queried for device descriptions or function blocks which could be reused (not shown in the system architecture).

```
using de.ma.fb.SwitchFunctionality; 1.0.0
writeset {
  de.ma.fb.SwitchFunctionality.operation.set
}
readset {
  de.ma.fb.SwitchFunctionality.status.currentState
}
triggerableset {
  de.ma.fb.SwitchFunctionality.operation.set
  de.ma.fb.SwitchFunctionality.operation.toggle
}
triggeringset {
  de.ma.fb.SwitchFunctionality.event.stateChanged
}
```

Listing 6: Semantic abstraction of SwitchFunctionality

As a next step, the semantic abstraction needs to be performed. The goal of this process is to identify how different elements of interfaces can interact with each other. This process can be either done automatically or manually.

```
using de.ma.fb.DimmerFunctionality; 1.0.0
writeset {
  de.ma.fb.DimmerFunctionality.operation.setValue
}
readset {
  de.ma.fb.DimmerFunctionality.status.currentValue
}
triggerableset {
  de.ma.fb.DimmerFunctionality.operation.setValue
}
triggeringset {}
```

Listing 7: Semantic abstraction of DimmerFunctionality

Having done that, interface elements which need to be mapped to enable the desired service must be identified. Without the support of the Matcher and an explicit abstraction, this step had to be done manually for each integration case. This means that an IoT solution developer needed to scan all interface elements of the switch to find the correct event that is triggered when the switch is toggled and all elements of the light to find a way for (de-)activation. With the help of the Matcher, this process can be done tool-supported. Given the function block abstraction, the Matcher creates a list of MappingBlock recommendations from which an developer can select the needed one based on prior defined integration cases (i.e. formalizing semantic integration knowledge incrementally).

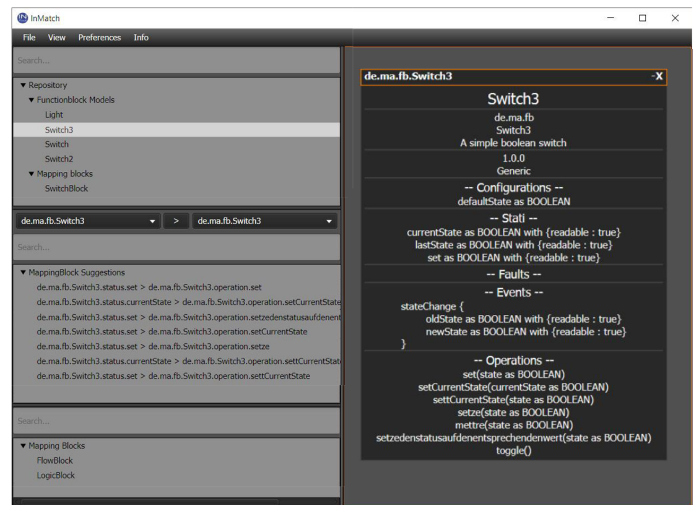

Fig. 4: InMatch Prototype utilized as tool-support for Mapping Language

In Figure 4, the "MappingBlock Suggestions" is realized. This component is essential for reusing incrementally defined integration knowledge. Here, the following comparison features are implemented for readset $\leftrightarrow$ writeset and triggerset $\leftrightarrow$ triggerableset: data types, data type transformations and language transformation (e.g. english to german) of signature elements. It is important to note that all suggestions are based on prior defined working integration cases and are not based on a formal standard.

```
id Switch2Dimmer
namespace de.ma.mappingblocks
description "Sets a dimmer value according to a
    switch."
sources {
  de.ma.fb.SwitchFunctionality
}
destinations {
  de.ma.fb.DimmerFunctionality
}
logicblock:
trigger {
  de.ma.fb.SwitchFunctionality.event.stateChanged
}
outputs {
  de.ma.fb.DimmerFunctionality.operation.setValue
}
action {
```

```
 var t = de.ma.fb.SwitchFunctionality.event.
    stateChanged;
 if(t.newState) {
   de.ma.fb.DimmerFunctionality.operation.setValue
     (1.0);
 } else {
   de.ma.fb.DimmerFunctionality.operation.setValue
     (0.0);
 }
}
```

Listing 8: Switch- to DimmerFunctionality MappingBlock

The next integration step is the creation of a mapping between the light's DimmerFunctionality and the switch's SwitchFunctionality. Therefore, a mapping between these function blocks needs to be specified. Vorto itself does not offer any mapping language which supports the mapping between function blocks or information models. Hence, an integrator would need to specify the mapping directly in a platform-dependent way (e.g. by programming an adapter for Amazon AWS). By using the presented mapping language, the mapping definition can be formalized by creating a MappingBlock in a platform- and technology-independent way. The result of this step can be found in Listing 8.

The last integration step is to use the mapping between function blocks to create the mapping between information models. Therefore, a mapping needs to be created and formalized. In the Mapping, the involved information models as well as the required MappingBlocks are combined to define how the IoT devices need to interact. The resulting Mapping can be seen in Listing 9.

```
id Switch2Light
namespace de.ma.mappings
description "A mapping which connects a switch to a
    light so that the light is toggled by the switch
    "
infoModelReferences {
  using de.ma.informationmodels.Switch;1.0.0 as
    Switch
  using de.ma.informationmodels.Light;1.0.0 as Light
}
mappingBlocks {
  from Switch to Light: de.ma.mappingblocks.
    Switch2Dimmer
}
```

Listing 9: Switch to Light Mapping

Limitations: Although the presented case study does not seem to improve the overall integration process in the given example a lot, the proposed language is a first technical step towards facilitating the presented, bottom-up integration methods based on incomplete integration knowledge [10] [13]. This means that with a high number of already integrated function blocks the described integration process can be automated based use-case specific semantics. The resulting efficiency gains are mostly due to reusing previously created MappingBlocks instead of creating new software adapters for every integration task. However, the proposed mapping language currently assumes that types per programming language are similar (e.g. 1 is always true and 0 is always false). Furthermore, the proposed language may not scale for 1:n or even n:m mappings as they can be proposed by the matcher component but must be eventually confirmed by the system integrator.

V. Conclusion and Future Work

In this paper a novel mapping language for IoT device descriptions has been introduced. By using a mapping language functional integration knowledge can be defined explicitly in a formal way. In order to reuse already captured integration knowledge, semantic abstractions were defined. In addition, logical as well as functional MappingBlocks have been introduced, which describe the interaction between function block elements. Our case study shows, that a mapping language with a moderate expression complexity is applicable for simple smart home scenarios. It can be seen that the proposed mapping language only requires a moderate formalization effort as existing mappings can be captured and reused incrementally. In the future, we plan to replace Vorto device descriptions with a formal component model that suits a broader range of IoT scenarios [8].

References

[1] M. Bauer, M. Boussard, N. Bui, F. Carrez, C. Jardak (SIEMENS), J. De Loof (ALUBE), C. Magerkurth (SAP), S. Meissner, A. Nettsträter (FhG IML), A. Olivereau, M. Thoma (SAP), W. Joachim, J. Stefa (CS-D/SUni), and A. Salinas, *Internet of Things - Architecture IoT-A Deliverable D1.5 - Final architectural reference model for the IoT v3.0*, 07 2013.

[2] R. Minerva, A. Biru, and D. Rotondi, "Towards a definition of the internet of things (iot)," *IEEE Internet Things*, pp. 1–86, 2015.

[3] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[4] P. Barnaghi, W. Wang, C. Henson, and K. Taylor, "Semantics for the internet of things: early progress and back to the future," *International Journal on Semantic Web and Information Systems (IJSWIS)*, vol. 8, no. 1, pp. 1–21, 2012.

[5] T. Vale, I. Crnkovic, E. S. De Almeida, P. A. d. M. S. Neto, Y. C. Cavalcanti, and S. R. de Lemos Meira, "Twenty-eight years of component-based software engineering," *Journal of Systems and Software*, vol. 111, pp. 128–148, 2016.

[6] S. Evdokimov, B. Fabian, S. Kunz, and N. Schoenemann, "Comparison of discovery service architectures for the internet of things," in *2010 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*, pp. 237–244, IEEE, 2010.

[7] I. Szilagyi and P. Wira, "Ontologies and semantic web for the internet of things-a survey," in *IECON 2016-42nd Annual Conference of the IEEE Industrial Electronics Society*, pp. 6949–6954, IEEE, 2016.

[8] C. Brooks, C. Jerad, H. Kim, E. A. Lee, M. Lohstroh, V. Nouvellet, B. Osyk, and M. Weber, "A component architecture for the internet of things," *Proceedings of the IEEE*, 2018.

[9] Z. Li, P. Liang, and P. Avgeriou, "Application of knowledge-based approaches in software architecture: A systematic mapping study," *Information and Software technology*, vol. 55, no. 5, pp. 777–794, 2013.

[10] F. Burzlaff and C. Bartelt, "Knowledge-driven architecture composition: Case-based formalization of integration knowledge to enable automated component coupling," in *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*, pp. 108–111, IEEE, 2017.

[11] H. Muccini and M. T. Moghaddam, "IoT Architectural Styles," in *Software Architecture* (C. E. Cuesta, D. Garlan, and J. Prez, eds.), Lecture Notes in Computer Science, pp. 68–85, Springer International Publishing, 2018.

[12] "https://www.eclipse.org/vorto/."

[13] F. Burzlaff and C. Bartelt, "I4. 0-device integration: A qualitative analysis of methods and technologies utilized by system integrators: Implications for enginering future industrial internet of things system," in *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pp. 27–34, IEEE, 2018.