A component based approach for the Web of Things

Andreas Ruppen
Software Engineering Group
University of Fribourg
1700 Fribourg, Switzerland
andreas.ruppen@unifr.ch

Jacques Pasquier
Software Engineering Group
University of Fribourg
1700 Fribourg, Switzerland
jacques.pasquier@unifr.ch

Sonja Meyer
EMPA
Research
St. Gallen, Switzerland
sonja.meyer@empa.ch

Alexander Rüedlinger
Software Engineering Group
University of Fribourg
1700 Fribourg, Switzerland
alexander.rueedlinger@unifr.ch

ABSTRACT

Model Driven Architectures are the holy grail of software engineering. Instead of writing code, developers draw models from the client's specification, which are then compiled into executable code (skeletons). We have taken this principle and applied it to the WoT. With the help of a meta-model tailored for the WoT we are able to build models to simultaneously take care of the physical and virtual aspects of smart devices. These models can then automatically be turned into code skeletons. The emphasis in the meta-model and its associated tools is reusability. Following the software engineering principle of independent reusable and deployable components, the outcome of the meta-model compiler are WoT compliant components.

CCS Concepts

•**Computer systems organization** → *Client-server architectures;* •**Software and its engineering** → **Layered systems; Software design techniques;** *Publish-subscribe / event-based architectures;*

Keywords

meta-model, Web of Things, model driven architecture, software architecture, component based approach, software development approach

1 Introduction and Motivation

Looking at the fast growing number of available consumer grade smart devices, it becomes obvious that the IoT (Internet of Things) is a reality: Belkin has a whole palette of IoT smart devices[1], Philips sells their hue lightbulbs[2] and Koubachi their Plant Sensor[3], to cite only a few. With the spread of cheap hardware containing numerous sensors and actuators, new architectures and paradigms are needed to efficiently build, connect and manage smart devices. To keep them manageable and interoperable requires standards. The WoT (Web of Things) introduces such a standard by imposing the REST (Representational State Transfer) architectural style [6] for all smart devices.

Using REST as the architectural style has proven advantages over other approaches [15]. HTTP is a proven protocol, is stable with a large community and has great support in all operating systems and most programming languages. Furthermore, RESTful interfaces facilitate the ad-hoc creation of mashup applications [8, 5, 18].

Yet, the WoT is facing a "things crisis". There are no standards or guidelines on how to build new smart devices. In this paper, we present the vision of an extended WoT addressing some major shortcomings of the WoT. This vision is supported by a meta-model tailored for the extended WoT. Its outcome is a methodology and associated tools to guide and help developers through the task of creating new smart-devices. Embracing the meta-model and its associated tools will lead to deployable and reusable components for the xWoT (extended WoT) .

2 Related Work

Research shows that, although the WoT is a viable approach, some important aspects are missing. First, the WoT only takes into consideration physical smart devices (e.g. a HVAC device or an AED). Second, pushing information from servers back to clients is difficult to achieve. Third, there exist very few guidelines on how to build "good" smart devices. The research WoT community actively addresses the first two topics; however, it still largely ignores the third one, although things are about to change.

Classically, the WoT is all about smart devices with a RESTful interface [3]. The WoT treats smart devices like

WoT '15, October 26 2015, Seoul, Republic of Korea

ISBN 978-1-4503-4045-8/15/10…$15.00

DOI: http://dx.doi.org/10.1145/2834791.2834792

[1] http://belkinbusiness.com/internet-things
[2] http://www2.meethue.com/de-ch/
[3] http://www.koubachi.com/

first class citizens, but at the same time, ignores everything else. However, research has shown that there is some interest in leveraging algorithms and other virtual goods to the WoT. Mayer and Karam proposed a computational space where smart devices could link to algorithms compatible with their outputs [12]. Since smart devices were to link to these algorithms, they needed to be fully embedded in the WoT. Meyer and Ruppen went one step further and discussed how entire business processes can be embedded into the WoT [13]. On the other hand, smart devices can also be seen as workers in business processes [14]. Therefore, smart devices leverage their capabilities to business processes. The important point of these researches is that smart devices and other virtual goods should integrate seamlessly with each other [13].

By definition, in client server systems, the client initiates the exchange and the server responds to requests. No other type of communication is possible. Over the years, research and industry came up with a few architectures to circumvent this problem and create, at least apparently, server to client interactions. Although all of these approaches work, there are differences in their performances. The most basic approach is *polling* or *long polling*, where a client executes repetitive `GET` requests. Atom and RSS feeds rely on this approach. Webhooks [7] take a completely different approach. Instead of having the client requesting a resource on the server, the client provides an URL where the server can later contact him. Therefore, the roles are inverted; the client becoming the server and vice-versa. Another popular approach are Websockets [11]. Not only do they solve many problems occurring with other approaches, they perform very well. While other approaches like streaming exist, they only play a minor role today.

Software components have been around for a while. When the software crisis hit the industry, Ledbetter and Cox proposed relying upon reusable software components, since this was already a standard in the hardware industry [10]. They stated that software must be capable of withstanding changes. Therefore, a key concept is encapsulation and interfaces to define a software's specification. According to Szyperski [17] components are the solution to building robust and reusable software. Accordingly, components are meant for composition, which is one way of reusing software. One of the first approaches to structure the IoT was Haller's proposal, *The Things in the Internet of Things*, which laid the foundations for the European project IoT-A [1]. Being a compromise between the views and opinions of the different participants of the project, the resulting reference model is too generic to be of any real use. It contains, however, some interesting points. More recently, the w3c has started a WoT interest group with the aim of structuring the current IoT [2].

Additionally, much work has been achieved in different aspects related to the IoT and, more precisely, to the WoT. Mashup applications and editors have been researched extensively [5, 9]. The results of these efforts are different platforms providing more or less sophisticated mashup capabilities [4]. IoTaaS platforms offer places to link smart device hardware to a publicly available API. However, all these discussions completely ignore the smart device itself and its underlying structure. It is taken for granted that the smart device already exists.

3 Reusability for the WoT

Ledbetter and Cox [10] identified reusability as one of the main factors to avoid the software crisis back in the early 80's. Software components apply Ledbetter's vision in practice. They have a clearly defined interface over which they can communicate with other components, but their inner guts remain hidden. Furthermore, components are ready to use. To run, they need to be deployed in a container. Therefore, components can be seen as the atomic units of software. New software creates new components, but it can also rely on already existing components to implement the services provided. We think that the same concepts should be true for the WoT. Whereas extensive research has been done about how to combine smart devices, we propose to go one step back and to re-think how to build these building blocks of the WoT.

This vision supports the concept that each smart-device corresponds to a RESTful web service having a clearly defined interface. Together, they form a deployable and reusable component. These components have, at the same time, a physical side made of the sensors and actuators defining the capabilities of the smart device, and a virtual side transposing those physical attributes into the virtual world. If these components are well built and structured, they can be easily deployed, managed and reused by other applications built on top of smart-devices. Let us consider the example of a door. We can admit that a door can be opened or closed and that it can be locked or unlocked. Therefore, to build a smart door, the physical side needs 2 actuators, one for opening and closing the door and another for locking and unlocking it. Furthermore, the physical side contains two sensors, each reporting one of the two properties. These physical properties need to be translated into the virtual world. Additionally, the virtual world should offer a notification system, so that clients (humans and machines) can rely upon these notifications to take action. These two facets, the actuators and sensors and their virtual counterparts, bundled together already form a component. Relying on a component based approach allows such components to always look similar to the outer world, regardless how their inner structure is. According to Szyperski [17], a component also needs a container. In the case of the WoT and RESTful Web services, this container is simply the web.

Of course, we can build all sorts of such simple components: smart blinds, smart smoke alarms, smart HVAC etc. Each of these components is of interest on its own. However, they can also be grouped together to form new components. Rather than just a single smart door, imagine now that we would like to build a smart corridor, containing smart lights and smart doors. We could do so by creating a mashup application relying on the capabilities offered by the different smart devices composing a corridor. However, a better approach is to create a new component *smart corridor* containing the different *smart light bulb* and *smart door* components as children. Modeling the smart corridor as a new component rather than using a mashup application has several advantages: (1) Already existing and deployed components can be reused. (2) The smart corridor component inherits all the properties of a component. This includes a well defined event architecture. (3) Since it offers a RESTful interface, applications can build on top of this component. (4) Being a component, the smart corridor can be reused in other scenarios like a smart building. (5) Unlike mashup

applications, components are deployed and always on, even if no client is currently using it.

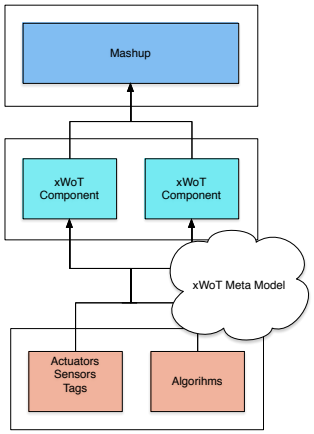

Figure 1: A three layered approach for reusable components

3.1 Building the Right Components

Before diving into the details of our meta-model, its scope of application needs to be defined. In any software, algorithms play a major role. In the WoT, they can occur at different levels: a mashup application can sometimes be seen as an algorithm taking different (sensor) inputs and producing a result. However, algorithms can also be of interest at the smart device layer. A unit conversion algorithm deployed as component with a RESTful interface allows sensors to deliver a large palette of data format. Deploying algorithms as components has not only the advantage of reusability but also allows for a seamless integration of time consuming algorithms [16] within the WoT. For these reasons we state that an algorithm should be treated in the same way as any other smart devices. As long as it offers a clear RESTful interface, it is treated as a first class-citizen. On a larger scale, this vision applies to all kinds of purely virtual goods. From a client's perspective, it is impossible to tell whether a result actually comes from a smart device, i.e. from a sensor, or is a computed one. Therefore we define the xWoT as a natural extension of the classic WoT encompassing - at the same time - physical and virtual goods. This point of view also covers RESTful APIs like the ones offered by Facebook and Twitter, which would be outside the scope of classic WoT applications.

To come back to the previous example, we can imagine a new xWoT component taking as input the different sensor values from the smart light bulbs and returning as output whether there is light in the corridor or not. The advantage of offering this simple algorithm as a new component, is its reusability: each corridor can now have its own instance of this algorithm. Additionally, this aggregation service might also be useful for other devices, as long as their inputs are compatible.

Popular mashup editors like IFTTT[4] show that events play a major role. Applications built on top of the xWoT can get the necessary information by two means: (1) they can actively fetch the data when needed or (2) they can listen to events. Whereas the first approach is fully supported by every RESTful API, the second approach requires some engineering and there is no standard for how to push events to clients. Instead there are a handful of competing and complementary approaches. To overcome this situation, each xWoT component integrates a well-defined pushing infrastructure so that applications can rely upon events. Rather than proposing yet another event architecture, xWoT components integrate a combination of the best available approaches: each resource capable of generating events contains a child resource responsible for the event architecture. This contains a WebSocket endpoint, useful in situations where events occur frequently, plus a Webhook for situations with less frequent events. Depending on the kind of event, each client can choose whether a WebSocket or a Webhook is more suitable. Additionally, upon subscribing, the user provides a machine-readable description of the kind of events he is interested in. Through this mechanism, a client can specify to be notified only if a given sensor or actuator is in some state. He can for example restrict the events generated by a smart thermometer to situations where the temperature is above $30°C$. This helps to keep the amount of notifications as low as possible.

The vision presented herein is particularly interesting if supported by the necessary tools and methodologies to automatically generate such xWoT components. This guarantees that new components meet the meta-model's requirement and stay composable. The reference architecture discussed in Section 2 is of particular interest. First, it acts as a starting point for our meta-model. By eliminating its shortcomings and limiting its application to the xWoT, we can define a simpler meta-model and also provide a model compiler, turning instances of the meta-model into almost ready to deploy xWoT components. Second, taking into account events and integrating a clearly defined pushing mechanism opens up the perspective for novel applications. xWoT components cannot only answer to *what* happens but also *when* something happens. These two questions are complementary and depending on the scenario, one or the other is of interest. Figure 1 introduces a three layered approach to create reusable and deployable xWoT components. It shows how the meta-model helps to shape xWoT components and how classic mashup applications like Xively[5] or IFTTT can build on top of these components. By sitting in-between the raw physical device or the algorithm and the final xWoT component, its associated tools support developers throughout the development lifecycle.

3.2 A Meta-Model for the xWoT

The core concept of the xWoT meta-model is the *Entity*. This reflects the point of view adopted by the application designer and gives birth to the base component. For the examples used throughout this paper, the Entity would be the abstract concept of a smart door or the abstract concept of a corridor. The previous section introduced virtual only goods as first class citizens of the xWoT. Therefore the Entity can be either a fully blown smart device or something purely virtual, like an algorithm. In the first case, we have learned from the examples above that a smart device has both a physical side and a virtual side. Algorithms, on the other hand, are virtual only and have no physical counterparts. Therefore, an Entity is always composed of a

[4]https://ifttt.com/

[5]http://xively.com

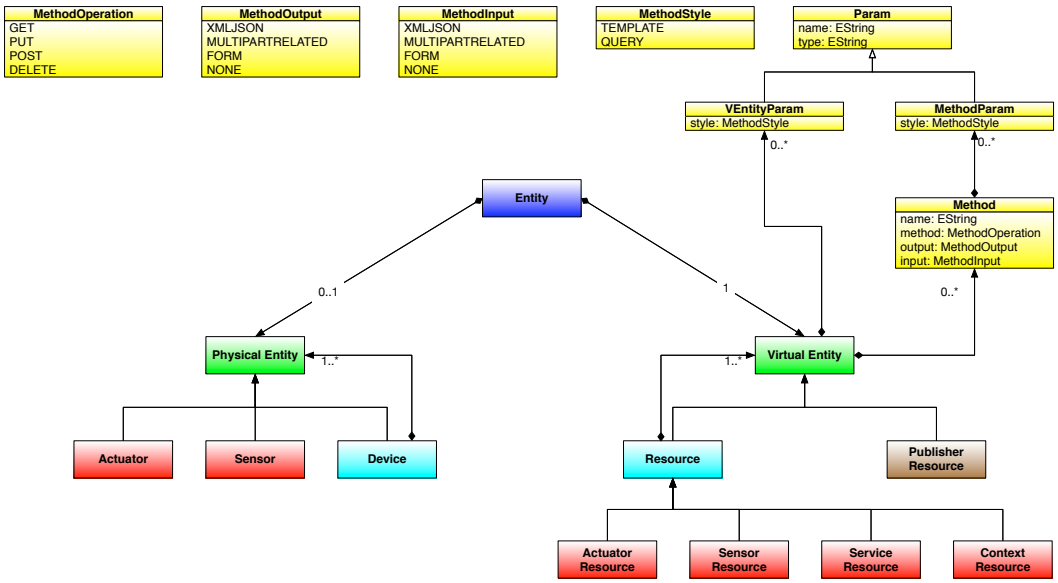

Figure 2: A Meta-Model for the WoT

virtual part called the *Virtual Entity* and, sometimes, also of a physical part called the *Physical Entity*. To ensure coherence between the physical and the virtual world, there is a one-to-one mapping from the Physical to the Virtual Entity. Furthermore, as shown in Figure 2, both are structured with the composite design pattern. The Physical Entity has one child node for composition (light blue) and several leaf nodes (red), one for each kind of hardware. The structure of the Virtual Entity is slightly different. It also has one node used for composition (light blue). However, its only leaf is the *Publisher Resource*. This is necessary to embed the publishing architecture into any kind of resource. Therefore, the concrete resources (red) are modeled as children of the composite node. *Actuator-* and *Sensor Resources* stand for their corresponding physical counterpart. *Service Resources* are necessary to model purely virtual goods like algorithms. Furthermore, in a very last step, the one-to-one mapping between the physical and the virtual side can sometimes be relaxed leading to more meaningful APIs. These simplifications ensure that the generated API (Application Programming Interface) is as flat as possible. One of these simplifications combines a Sensor Resource and an Actuator Resource into a *Context Resource*. Semantically, a Sensor Resource only responds to `GET` requests while an Actuator Resource responds to `PUT` requests. However, if the actuator and the sensor are highly coupled (e.g. light switch, and luminosity sensor), they can be combined into a single Context Resource accepting both, `GET` and `PUT` requests. Finally, items in yellow are necessary to define the possible interactions as well as inputs and outputs of the modeled RESTful service.

The application of the composite pattern is also the key to building reusable components. As shown in the top part of Figure 3 for the smart door example, the physical side is a *Device* representing the door. The latter is composed of a *Device* for the open/close mechanism and a second one for the locked/unlocked mechanism. Each of these Devices is composed of a *Sensor* and an *Actuator*. Translated to the virtual world, this gives birth to the exact same structure as a RESTful web service and the outcome of this model is exactly one xWoT component. We call this type of composition atomic.

In the previous section, we discussed a smart corridor example. Here, the Physical Entity is a Device representing the corridor. Its only children are other Devices, one for each smart thing composing the smart corridor, e.g. smart doors or smart light bulbs. Such compositions are non-atomic, meaning that they translate to more than one xWoT component; one for the smart corridor, one for each smart door and one for each smart light bulb. Accordingly, the compiler translating xWoT models into code skeletons creates three service modules. Listing 1 shows these three modules. Since components are reusable, if somebody has already build and deployed smart doors and/or smart light bulbs, then only the additional smart corridor component need to be implemented. However, this last component will no re-implement functionality already provided by the two others. Instead, requests will be redirected to these components to achieve the desired effect.

```
1  aragorn@gondor:src$ ll REST-Servers/
2  total 0
3  drwxrwxr-x 3 aragorn staff 1.0K Aug 18 11:03 NM-
       ↪ _corridor_Server/
4  drwxrwxr-x 3 aragorn staff 1.1K Aug 18 11:03 __int:
       ↪ doorid__Server/
5  drwxrwxr-x 3 aragorn staff 612B Aug 18 11:03 __int:
       ↪ lbid__Server/
```

Listing 1: Generated modules for the smart corridor use-case

3.3 Meta-Model Compilers

The meta-model also introduces a Model Driven Architecture approach for the xWoT. Instead of writing code, developers deal with models to create a figure of the physical world. These models serve as starting point to automatically generate the necessary code. The rules introduced with our meta-model allow for such an approach. Since there is a

one-to-one mapping from the Physical to the Virtual Entity, the developer only model physical interactions in terms of actuators and sensors on the physical side. For the previous smart door example, the developer would model the physical properties like the ones of Figure 3.

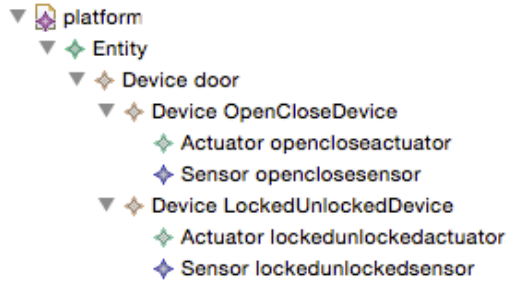

Figure 3: Physical side of a smart door

In a second step, the corresponding virtual side can be generated automatically by applying the rules introduced with the meta-model (one-to-one mapping plus relaxing constraints). Whenever more than one option is possible, the compiler ask the user for a resolution. The outcome of this compilation is a new xWoT model. For the smart door example, the outcome of this step would produce the same model as the one discussed previously on Figure 3. Of course, the developer can refine this virtual representation by adding supplementary resources having no physical counterparts. Once the virtual side is finished, a second compiler takes it as input and generates code skeletons like those from Listing 1. The model compiler takes care of the non-atomic compositions and creates the right number of xWoT components. Also, the compiler produces the necessary code for the event mechanism, where needed. Thus, the outcomes of the model are almost ready to deploy xWoT components. These skeletons contains the basic RESTful interface definition plus the event infrastructure. The developers still needs to connect this code with the raw hardware. Since there are many ways to link hardware to code (I2C for example) the compiler creates one class making the interface between the hardware and the code. This has the advantage that the hardware might change without affecting the rest of the code. Furthermore the developers also needs to take care of the different inputs and outputs (XML, JSON HTML etc.). Once these two gaps filled in, the components are ready to be deployed.

The common point of all xWoT components is the meta-model which is expressed as a model in EMF. Therefore, it is language agnostic. This allows the model compiler to generate various different implementations ranging from Node.js over Python to Ruby. For each model, the developer can choose among the proposed target platforms the most suitable one and have the according code generated.

4 Validation

To prove the applicability of the presented meta-model and its associated tools we have set up a laboratory where students can build and deploy xWoT compatible smart devices. The laboratory provides an isolated testbed where devices can be deployed an tested. Its infrastructure supports a network topology where devices can live in separated networks. This is important to test features like cross-border discovery, pushing information beyond physical network boundaries etc. Finally, the setup is also responsible to separate the laboratory from the rest of the university network and still providing Internet access to all of the deployed smart-devices.

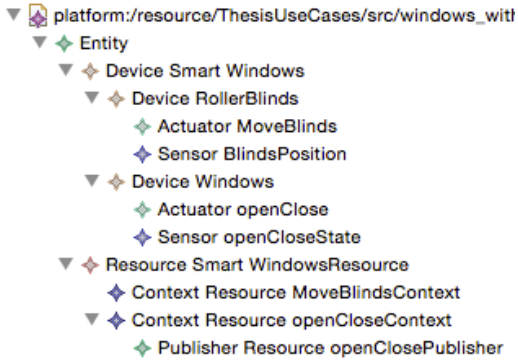

Figure 4: Entity Model of a smart windows

Since all smart devices need to have their own HTTP server, we choose Raspberry Pies as a platform for all smart devices. They natively run Python and Node.js which also explains why our model compiler supports, among others, these two target languages. To simplify the communication with the hardware, we implemented the required sensors and actuators on various Arduino boards. The boards are then wired up over the I2C bus with the Raspberry and allow for a seamless integration. This combination of Raspberry Pi and Arduino is used for most smart devices deployed in our lab. The Raspberry Pies run the generated RESTful web servers and use the I2C protocol to communicate with sensors, actuators or Arduinos. For example, Figure 4 shows the physical and virtual model of a smart window with automatic blinds. The blinds can be opened and closed and so can the window itself. These actions are represented in the physical part of the model (upper part of Figure 4). With a first compiler run, the lower part of Figure 4 is generated. This part can now be tweaked an refined to fit the individual needs of the current smart device.

```
1  pi@window-2 ~/shutter-window/app $ ls -Rl
2  .:
3  total 296
4  -rw-r--r-- 1 pi pi 1603 Jul 19 08:34 description.jsonld
5  -rw-r--r-- 1 pi pi 450 Aug 14 16:07 runserver.py
6  drwxr-xr-x 2 pi pi 4096 Sep 29 07:32 xwot_app

8  ./xwot_app:
9  total 40
10 -rw-r--r-- 1 pi pi 388 Sep 26 16:59 RootResource.py
11 -rw-r--r-- 1 pi pi 1105 Sep 29 07:32 BlindsResource.py
12 -rw-r--r-- 1 pi pi 1025 Sep 26 16:59 WindowResource.py
13 -rw-r--r-- 1 pi pi 777 Jul 19 08:34 __init__.py
```

Listing 2: Generated Server Code for the Smart Blinds

The second compiler translates the model of Figure 4 into code skeletons which need to be completed. Listing 2 shows some of the generated files by the compiler. Once the code gaps filled (mainly I2C bus and I/O formats) the code can be deployed on the Raspberry Pi and wired up with the hardware. Figure 5 shows the final hardware implementaion like it is deployed in our laboratory.

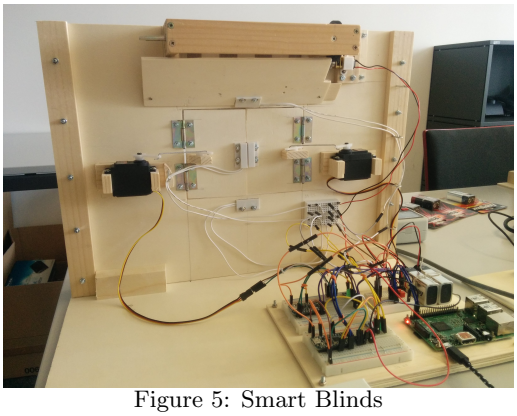

Figure 5: Smart Blinds

5 Conclusion

Through a number of advanced student works conducted in our xWoT lab, we have successfully evaluated the applicability of the xWoT meta-model. Our experiences have shown that the meta-model greatly simplifies the task of creating new xWoT components. Another benefit of the meta-model is the decoupling of the raw hardware from the RESTful interface. The communication between the two is encapsulated in one module, letting the developer implement the hardware side according to his preferences. This decoupling also allows developers to use mock hardware during the development and testing phases. If later the real hardware is hooked up, it is just a matter of changing one module. Although the meta-model may seem small, it covers all the necessary aspects of xWoT components and is still open for future extensions. Currently, we have ongoing research in two directions: (1) A discovery mechanism tailored for the xWoT allowing a late binding in mashup applications. (2) Semantics describing the capabilities of the component. Furthermore, if our components contain a semantic description, we can start semantic discovery. Therefore, it would not only be possible to discover actuators and sensors, but also a smart door or a thermometer. Additionally, user interface can rely upon the semantic description of the resource to get a hint about what it contains and therefore choose an appropriate representation. In the near future, the meta-model, or at least the associated compiler, should also take care of these aspects and produce semantically discoverable xWoT components. This would lead to reusable, semantically meaningful and discoverable xWoT components.

6 References

[1] Internet of Things Architecture. http://iot-a.eu (accessed August 19, 2015).

[2] Web of Things Interest Group. http://www.w3.org/WoT/IG/ (accessed August 19, 2015).

[3] D. Guinard. *A Web of Things Application Architecture – Integrating the Real-World into the Web*. PhD thesis, ETH Zurich, Zurich, Switzerland, 8 2011.

[4] D. Guinard, M. Mueller, and J. Pasquier. Giving RFID a REST: Building a Web-Enabled EPCIS. In *Proceedings of Internet of Things 2010 International Conference (IoT 2010)*, Tokyo, Japan, 2010.

[5] D. Guinard, V. Trifa, T. Pham, and O. Liechti. Towards Physical Mashups in the Web of Things. In *Proceedings of INSS 2009 (IEEE Sixth International Conference on Networked Sensing Systems)*, Pittsburgh, USA, 6 2009.

[6] D. Guinard, V. Trifa, and E. Wilde. A Resource Oriented Architecture for the Web of Things. In *Proceedings of Internet of Things 2010 International Conference (IoT 2010)*, Tokyo, Japan, 2010.

[7] V. Gupta, R. Goldman, and P. Udupi. A network architecture for the Web of Things. In *Proceedings of the Second International Workshop on Web of Things*, WoT '11, pages 3:1–3:6. ACM, 2011.

[8] K. Kenda, C. Fortuna, A. Moraru, D. Mladenić, B. Fortuna, and M. Grobelnik. Mashups for the Web of Things. In *Semantic Mashups*, pages 145–169. Springer, 2013.

[9] J. Lathem, K. Gomadam, and A. P. Sheth. SA-REST and (S)mashups: Adding Semantics to RESTful Services. In *Proceedings of the International Conference on Semantic Computing*, ICSC '07, pages 469–476, Washington, DC, USA, 2007. IEEE Computer Society.

[10] L. Ledbetter and B. Cox. Software-ICs. *Byte Magazine*, 10(6):307–316, June 1985.

[11] P. Lubbers, B. Albers, R. Smith, and F. Salim. *Pro HTML5 Programming: Powerful APIs for Richer Internet Application Development*. Apress, Berkely, CA, USA, 1st edition, 2010.

[12] S. Mayer and D. S. Karam. A computational space for the web of things. In *Proceedings of the Third International Workshop on the Web of Things*, WOT '12, pages 8:1–8:6. ACM, 2012.

[13] S. Meyer and A. Ruppen. An Approach for a Mutual Integration of the Web of Things with Business Processes. In J. Barjis, A. Gupta, and A. Meshkat, editors, *Enterprise and Organizational Modeling and Simulation*, volume 153 of *Lecture Notes in Business Information Processing*, pages 42–56. Springer Berlin Heidelberg, 2013.

[14] S. Meyer, A. Ruppen, and C. Magerkurth. Internet of Things-aware Process Modeling: Integrating IoT Devices as Business Process Resources. In *CAiSE 2013*, 2013.

[15] C. Pautasso, O. Zimmermann, and F. Leymann. Restful web services vs. "big"' web services: making the right architectural decision. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, New York, NY, USA, 2008. ACM.

[16] A. Ruppen, J. Pasquier, and T. Hürlimann. A RESTful architecture for integrating decomposable delayed services within the web of things. *Int. J. Internet Protoc. Technol.*, 6(4):247–259, 6 2011.

[17] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[18] R. Tuchinda, P. Szekely, and C. A. Knoblock. Building Mashups by example. In *Proceedings of the 13th international conference on Intelligent user interfaces*, IUI '08, pages 139–148. ACM, 2008.