Running Distributed and Dynamic IoT Choreographies

Jan Seeger, Rohit Arunrao Deshmukh, Arne Bröring

***Abstract*—IoT systems are growing larger and larger and are becoming suitable for basic automation tasks. One of the features IoT automation systems can provide is dealing with a dynamic system – Devices leaving and joining the system during operation. Additionally, IoT automation systems operate in a decentralized manner. Current commercial automation systems have difficulty providing these features. Integrating new devices into an automation system takes manual intervention. Additionally, automation systems also require central entities to orchestrate the operation of participants. With smarter sensors and actors, we can move control operations into software deployed on a decentralized network of devices, and provide support for dynamic systems. In this paper, we present a framework for automation systems that demonstrates these two properties (distributed and dynamic). We represent applications as semantically described data flows that are run decentrally on participating devices, and connected at runtime via rules. This allows integrating new devices into applications without manual interaction and removes central controllers from the equation. This approach provides similar features to current automation systems (central engineering, multiple instantiation of applications), but enables distributed and dynamic operation. We also quantitatively evaluate the performance of our chosen approach.**

***Index Terms*—Service Composition, Semantic Representation, Internet of Things, Dynamic Service Composition, Service Choreography, Semantic automation**

I. Introduction

The Internet of Things (IoT) is making rapid inroads into peoples' everyday life. The new breed of devices such as the Amazon Echo, Phillips Hue lights and the Nest thermostat allow users to build advanced functionality into their homes. Using simple configuration tools, users can easily modify their homes and add new devices or reconfigure old ones.

Similar developments are going on in context of automation for commercial buildings. Today, automation systems for commercial buildings are installed by specialists and typically only configured once during the deployment phase. Reconfiguring building automation (BA) systems after installation takes a high amount of effort from trained specialists. Also, the engineering of BA systems is mostly static, adding or removing devices requires the involvement of highly qualified personnel such as electricians and BA specialists for deploying and (re-) configuring the devices and system. Expressing dynamic behavior, where devices are added or removed during the operation of the system, is even more difficult. Sometimes, not even the installation plans of a BA system are available after installation, which means that a "reverse engineering" of the system becomes necessary for reconfiguration.

Current building automation systems are also centralized to a large degree, with one or more central controllers transferring and converting signals. The growing computation power of sensors and actuators will make these controllers superfluous, and will allow control algorithms to move into sensors and actuators. Also, controllers form a single point of failure for the system, where the failure of a controller renders building parts inoperable. Traditional building automation systems are a form of *orchestration*, where a central controller *orchestrates* the interaction of components. In this paper, we will move towards a *choreography* of sensors and actors, where the actions of each participant are not controlled by a single controller, but in a distributed fashion. This transition leads to a number of challenges in management and operation of the system.

We tackle these challenges in this paper by developing a mechanism for the dynamic and automated management of IoT choreographies at runtime. Our approach is based on semantic technology to describe the structure and configuration of a system based on so-called *Recipes*. A recipe defines the data flow between IoT devices, so-called *Offerings*, as an abstract template. We introduce runtime configuration of recipes and allow the definition of communication links to be expressed as rules, so-called *Offering Selection Rules*. These rules are evaluated at runtime whenever devices are added or removed from the IoT system, in order to keep the recipe choreography running and automatically incorporate new devices. We illustrate this approach at hand of a use case example from the building domain that is referred to throughout the rest of the paper. This use case validates the system design and is demonstrates the advantages of dynamic choreographies.

Our use case for evaluation is as follows: A recipe defines the interaction between multiple switches, office lights and motion sensors. The office lights are controlled by motion sensors and are switched on if motion is detected at any one sensor, but only if any of the switches is enabled. We will demonstrate in the rest of this paper how our framework allows the centralized creation and decentralized operation of such a system, allowing integration of new devices at runtime.

The remainder of this paper is structured as follows: Section II provides background to our work and outlines related work. In Section III, we describe how services are described in our automation system. Section IV describes how the selection of service components can be restricted. In Section V, we

Jan Seeger is with Corporate Technology, Siemens AG and TU München networking chair

Rohit Deshmuk is with TU Darmstadt

Arne Bröring is with Corporate Technology, Siemens AG

This paper was funded by the EU project 688038, "Bridging the Interoperability Gap of the Internet of Things"

define the process by which devices are added into the network and how offering selection rules are evaluated to build a choreography. In Section VI, we provide a performance evaluation of the central orchestration component. We conclude this paper in Section VII and illustrate future avenues of research.

II. Background & Related Work

Traditional building automation is based on a static configuration created by highly specialized and developed tools. The per-application (room lighting, room shading, etc.) *room controller* (RC) provides functionality by accessing connected sensors and actors. All services available from the controller are preloaded on the controller, and may be parametrized via tools provided by vendors. When adding new devices, they need to be physically connected to the controller, and the controller needs to be parametrized to use the newly connected devices via the provisioning software (Siemens "ABT Site"[1] tool, or KNX Association "ETS tool"[2]). This configuration process means that room automation functionality is limited to preconfigured functionality on the controller, and that the implementation of dynamic services is difficult. Additionally, this means that that information on building configuration is only available off-line in the provisioning software configuration file, not on-line in the running system.

Research on building automation tools has led to some advances in the field: Model-based tools have not gained wide acceptance, but represent the current state of the art in building automation system tool research [1]. Model-based tools allow the configuration and management of BA systems on a higher level. However, they do not provide the required underlying technology to realize dynamic choreographies as our approach integrating both tools and underlying platform does. The semantic approach taken by Thuluva et al. [2] extends automation systems to allow engineering and operation of automation systems. While these approaches provide functionality for distributed operation of services, no dynamic configuration of the system is supported without user involvement.

In the commercial sphere, "lightweight" automation services have become popular. The foremost example here is probably "If This, Then That" (IFTTT)[3], which allows the limited composition of web services and IoT devices with a user-friendly interface. IFTTT and similar commercial automation services are however limited in their integration with other services. Integrating external services is difficult due to the inability of these services to export automation descriptions for use in other tools. By describing services with semantic technology, our system simplifies the creation of external tools to interact with our system.

The "recipe" concept is a composition language for automation components [3]. Service composition consists of discovering services and connecting them to each other. In the context of *web* services, there has been intense research activity on composition approaches [4], such as WSDL [5] or REST-based techniques [6], [7].

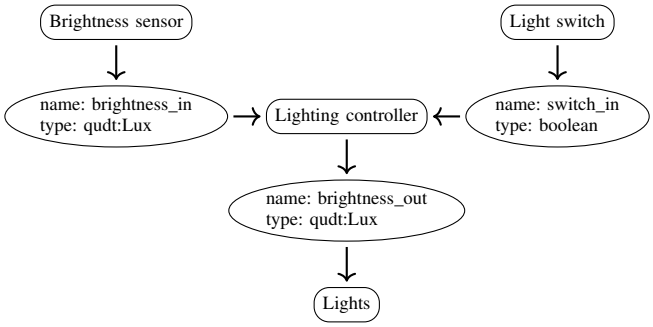

Figure 1. A lighting control recipe with sensors, switches and lights.

Other offerings for service composition are Node-RED[4] and FlowHub[5], which allow the creation of data-flow based compositions. Flow-based research systems include Calvin [8] and Distributed Node-RED [9]. However, the underlying models used by flow-based composition platforms are not expressive enough to ensure an error-free composition. The semantic descriptions used in our framework contain enough information to prevent incorrect service compositions.

Apart from the mechanics of composition, there has also been research on dynamically adapting systems. Using a system of rules allows the automation system to automatically adapt to changing circumstances for autonomous management [10] in a similar vein to our rule-based approach.

III. Offerings & Recipes

In the following, we present the recipe and offering models. These models have been elaborated as part of the BIG IoT project [11] and have been initially introduced in our previous work [3]. Here, we provide an update of these models. This overview is needed to describe the extensions of these models for dynamic choreographies and making them runtime-ready in the following sections.

"Recipes" define templates for compositions of *ingredients* and their *interaction*s. Ingredients are placeholders for *offerings*, devices and services that process and transform data. Interactions describe the dataflow between these ingredients. An example recipe is shown in figure 1 describing a lighting control system. A lighting controller takes input from brightness sensors, calculates the output brightness through an algorithm (averaging, for example) and outputs the calculated value to the connected lights, but only if one of the switches is switched on. Inputs and outputs have both a name and a type. The type is used for matching offerings with ingredients. This process will be described in Chapter IV.

Offerings describe service or device instances, and how to access these services or devices. Offerings are specified in a semantic format by the so-called "offering description". Offering descriptions contain information on the in- and outputs of an offering as well as information on how to access the underlying

[1] http://www.buildingtechnologies.siemens.com/bt/global/en/buildingautomation-hvac/building-automation/building-automation-and-control-system-europe-desigo/room-automation/pages/room-automation.aspx

[2] https://www.knx.org/in/software/ets/about/index.php

[3] http://ifthisthenthat.com

[4] http://nodered.org

[5] http://flowhub.io

```
1  {
2   "localId": "officeLightOffering",
3   "category": "schema:lighting",
4   "endpoints": [{
5     "uri":
6      "coap://127.0.0.1:5683/LuminaireController",
7     "endpointType": "COAP_PUT",
8     "acceptType": "APPLICATION_XML",
9     "contentType": "APPLICATION_XML"}],
10  "requestTemplate":
11   "<dimmableValue>@@brightness@@</dimmableValue>",
12  "responseMapping": null,
13  "inputData": [{
14    "name": "brightness",
15    "valueType": "xsd:float"}],
16  "outputData": [],
17  "extent": {"city": "Munich"}
18 }
```

Listing 1. Example offering description for a CoAP-enabled office light.

service or device (providing the offering implementation). An excerpted offering description for our switch-sensor-controller-light example is shown in listing 1.

The offering description contains functional as well as non-functional properties. Functional properties describe the implementation of the offering (e.g. which web service endpoint this offering accesses and procotol and payload of the request), while non-functional properties describe installation-specific metadata about the offering (such as the price or location of the offering). Non-functional and functional properties thus correspond to offering "interface" and "implementation", respectively. In detail, the offering description contains the following information:

The `inputData` and `outputData` (lines 14 and 18) functional properties contain information on the types of input and output that this offering consumes and produces. They are visible in the recipe in figure 1 as type annotations on the input and output nodes. Type annotations are URIs referencing for example a term in the schema.org [12] or QUDT [13] ontologies. Additionally, a category is used to classify the offering, for example, into "smart building" or "transportation" categories. While being useful for users during the creation of recipes, type and category properties are also used in the basic matchmaking algorithm described in the next section.

The internal properties `endpoints`, `requestTemplate` and `responseMapping` (lines 4, 11 and 13, respectively) specify how this offering accesses the underlying service or device. The endpoint describes the adress under which the web service implementing this offering is reachable. To define and parse communication payloads, the BIG IoT library can be used [14][6]. The BIG IoT library allows interpolation of input values into URLs, URL queries and request bodies, while the response can be parsed into output values via a simple parser that can be parametrized per offering description. Supported protocols for endpoints are HTTP and CoAP, with POST, PUT and GET methods supported for both protocols. Additionally, the asynchronous OBSERVE option is supported for CoAP. Payloads can have XML or JSON format.

For example, the offering in figure 1 allows dimming a light via an XML payload over CoAP as defined in the `endpointType` and `requestTemplate`. Finally, non-functional properties (`extent` line 19, in this example) contain information about the offering that support their discovery and selection restriction beyond the basic matching algorithm. Both algorithms (basic matching on functional properties and advanced matching on non-functional properties) will be described in the next section.

The duality between offerings and ingredients is central to our system: It allows us to utilize a recipe as choreography descriptions independent of concrete implementations. A recipe is concrete enough so it can be successfully created as a blueprint by users using our tools, but so generic that it can be implemented and run using a wide variety of service implementations without requiring modification of the recipe.

In the next chapter, we describe the process of turning a recipe into a runnable instance.

IV. Instantiating Recipes

"Instantiating" a recipe refers to the process of replacing ingredients with offerings, resulting in a recipe that's executable. A recipe may be instantiated multiple times with different offerings, depending on the requirements. To instantiate a recipe, suitable offerings are selected by their external properties described in Section III. Then, extra restrictions called "offering selection rules" can optionally be applied. Finally, a recipe can be executed as a choreography, as described in Chapter V.

The matching algorithm to select suitable offerings works as follows: For each ingredient in the recipe, the database is searched for offerings that can replace this ingredient. Replacement is governed by the following algorithm:

Let i be an ingredient, and o an offering. We also define *category*(), *inputs*() and *outputs*() to access the so-named properties of the offering and ingredient description in Chapter III.

Furthermore, we use the "subclass of" operator $\sqsubseteq$ to express subclass relations.

o can replace i iff:

- The category of the offering is a subclass of the category of the ingredient: $\mathit{category}(o) \sqsubseteq \mathit{category}(i)$.
- For each input of the offering, the ingredient has at least one input with the same or subclassed type: $\forall in_o \in \mathit{inputs}(o) : \exists in_i \in \mathit{inputs}(i) : in_i \sqsubseteq in_o$.
- For each output of the offering, the ingredient has at least one output with the same or superclassed type: $\forall out_o \in \mathit{outputs}(o) : \exists out_i \in \mathit{outputs}(i) : out_o \sqsubseteq out_i$.

Note that this allows offerings to have fewer inputs than the ingredient, as well as more outputs. Superfluous outputs and inputs are ignored. In order to instantiate a recipe, each of the ingredients in the recipe has to be filled by at least one offering.

This purely type-based matching is very generic, but also rather limited. Realizing simple use cases such as "control all lights in room 3 via any switch in the same room" would require defining categories specifically for this application scenario (e.g., defining the category type "lighting in room 3").

To address this and to keep recipes generic, we introduce the concept of "offering selection rules" (OSRs), which allow

[6]Available at https://gitlab.com/BIG-IoT/lib-java

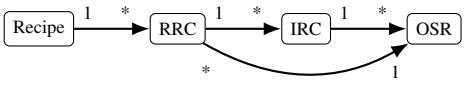

Figure 2. Relation of Recipes and OSRs

users to specify offerings that should participate in a recipe in fine-grained detail. These rules are attached to an ingredient and specify additional requirements on its non-functional properties that an offering needs to provide to be considered for filling this ingredient.

Offering selection rules are evaluated on the non-functional properties of an offering (see Section III). Non-functional properties can include location of the component, owner of the component or the energy efficiency of this component. Because the offering description is specified in a semantic format, non-functional properties can be extended easily.

OSRs can query these properties for equality or inequalities to a literal value. Multiple OSRs can be composed using boolean operators `AND` and `OR`.

Additionally, the cardinality of an ingredient can be specified using OSRs. This means that the minimum and maximum number of offerings replacing an ingredient can limited.

Using this set of OSRs, it is possible to constrain recipes in complicated ways going beyond the basic matching algorithm. The light recipe from figure 1 could for example be constrained to only match lights, sensors and switches in room A, with the controller not being constrained to a certain location, but to a cardinality of one. Instantiating the recipe would then result in one controller being connected to all sensors, all lights and all switches in room A. By adding a different set of OSRs to the system, the recipe might be constrained to room B.

This functionality is provided in current automation systems by defining templates that describe the a single deployment and then instantiating these templates multiple times, once for each room. Templates are not held available during the runtime of the system, and thus cannot be reevaluated for dynamic operation of the system. Using OSRs, the policy that led to the system's current configuration is always accessible and available. The policy can thus be reevaluated on system changes, something that is not possible with the template-based approach.

Conceptually, we have implemented these concepts as follows: Recipes, offerings, ingredients and OSRs are stored in an Apache Jena triple store[7] as a semantic graph. The objects in this semantic graphs are *recipes*, *recipe runtime configurations*, *ingredient runtime configurations* and finally *offering selection rules*. The relations between those concepts are shown in figure 2.

Recipes are designed using the recipe design tool described by Thuluva et al. [3] and are stored in the central repository. Recipes are then instantiated by creating a "recipe runtime configuration" (RRC) for this recipe. Each RRC describes a specific instantiation of a recipe. A RRC is an installation-specific instantiation of a recipe, because it can (and often will) contain restrictions on non-functional properties (such as the location) of offerings. Recipes, on the other hand, describe installation-independent patterns of interaction.

[7] https://jena.apache.org/

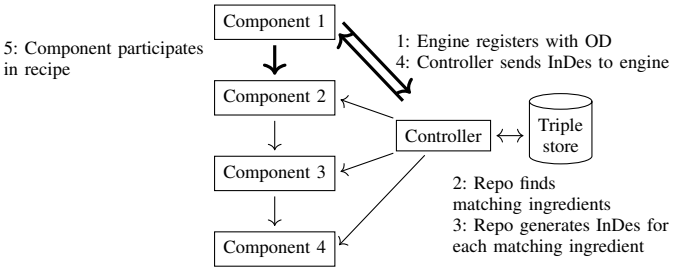

Figure 3. Incorporation of new devices into orchestration

The RRC can contain per-recipe OSRs such as cardinality, and always contains per-ingredient information called "ingredient runtime configuration" (IRC). Each RRC contains multiple IRCs, one for each ingredient in the recipe. IRCs contain runtime information describing the current cardinality of the ingredient, as well as the offerings which are currently implementing the ingredient and finally, offering selection rules (OSRs) restricting the set of offerings that can replace this ingredient.

The specification of a service composition as a recipe refined by a set of queries allows the creation of dynamic systems. We describe the realisation of such systems in the next section.

V. Design for Enabling IoT Choreographies

The concept of OSRs allow the addition of offerings into a running system without manual intervention.

To realize this functionality, our system consists of three parts, as seen in figure 3:

- A component for computing choreographies ("controller")
- A triple store for data storage
- An "engine" running on participating components

The controller is the central component of our system. The controller instantiates recipes, and handles the addition and removal of offerings using the triple store in the background for persistent data storage. The "engine" implements a gateway and enables devices to participate in the system. This is done by accepting input from other engines running on other components, passing this input to the offering implementation via the mechanisms described in Section III, parsing the output of the implementation, and sending it to the next offerings currently part of the recipe. Thus, recipes are turned into distributed choreographies.

In the future, the engine will run on smart sensors and actuators directly, and enable direct integration of these devices. Currently, the engine is on Raspberry Pi single board computers connected to hardware devices.

It is crucial to note that the controller is *not* a single point of failure. Without the controller, all recipes will continue operating, only the addition and removal of components is impacted.

```
1  {"offering": "bigiot:light-control",
2    "recipeRuntimeConfiguration":
3      "bigiot:rrc1",
4    "outputs": {
5      "brightness": {
6        "http://lamp1/input":
7          "on\_off",
8        "http://lamp2/input":
9          "on\_off"
10     }},
11   "inputs": {
12     "bigiot:sensor1": ["sensorin"],
13     "bigiot:switch1": ["switchin"]}}
```

Listing 2. Example interaction descriptor for a lighting controller connected to two lights and one sensor and switch.

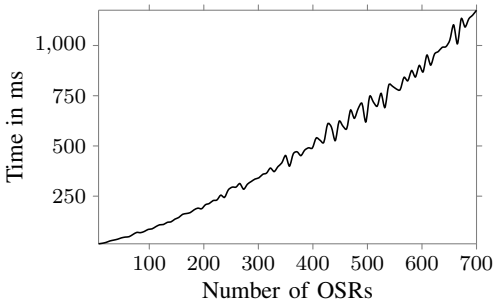

Figure 4. Performance evaluation of controller

The workflow for realizing dynamic choreographies is shown in figure 3. When a new component is connected to the network, the engine registers at the controller with its offering description (OD) (step 1). This offering description includes all the information necessary for deciding whether the component should be part of a choreography. The offering description is added to the triple store.

In step 2, the controller finds all matching IRCs for an offering by computing the matching between the IRC's ingredient and the new offering with the algorithm in Section IV. Then, for each IRC matching the new offering, all associated OSRs are evaluated. This is done by serializing the OSRs to SPARQL queries [15] and running them on the triple store. This results in a number of IRCs that the new component will be added to.

From this information, *interaction descriptors* (InDes) are generated in step 3. Each interaction descriptor describes the communication behavior of one device as part of a choreography. InDes' are derived from the recipe by finding all other offerings that an offering should communicate with and accept input from. An example interaction descriptor is shown in listing 2. An interaction descriptor contains information on where to send the outputs of this offering (lines 5–12) and which inputs to expect (lines 13–16).

Finally, interaction descriptors are sent to each device participating in the choreography (step 4), in order to inform it of its communication partners.

Based on the information contained in the InDes, each component has the knowledge to participate in a choreography (step 5). Thus, each choreography can now run autonomously, with the new component integrated into it.

This process works analogously for offering removal. When a device unregisters or fails, its offering description is removed from the triple store. The controller finds all IRCs that contained this offering and removes it. Optionally, the removed offering may be replaced by another offering already available in the triple store.

In the next section, we will quantitatively evaluate the computational cost of OSR resolution.

VI. Evaluation

To evaluate the scalability of our implementation, we measured the performance of the controller when adding new components. The computational factor dominating the addition of new components is the matching and resolution of OSRs. To find the set of ingredients that the new component can replace, all OSRs need to be evaluated. Thus, it is expected that the computation time for the addition of new components scales with the number of RRCs in the system.

In order to evaluate this, we measured the time between the addition of a component into the database and the conclusion of OSR computation, when a list of all suitable choreographies is available. To check the scalability of the controller, we measured performance with an increasing number of RRCs ranging from 7 to 700 using a set of 7 OSRs instantiated n times for n from 1 to 100. The number of components or recipes in the system does not influence the matching performance, since only RRCs are checked for a match with the new component.

Testing was done on a machine with 8 GB of RAM and a 2.4 GHz i5 mobile Intel processor with 4 threads.

With the controller being the only central component of the system, its performance dominates that of the complete system, and is therefore a suitable indicator of the overall performance. As can be seen in figure 4, the controller scales well enough, with a computation time of one second being broken at about 650 RRCs in the controller. The system scales quadratically in the number of RRCs, but with low constant factors. Overall, the controller performance only becomes unacceptable for our purposes with very large systems of more than 650 components.

VII. Conclusions & Future Work

In this paper, we present a concept, implementation and evaluation for running dynamic IoT choreographies. These dynamic choreographies provide am approach that is novel in IoT environments and particularly useful in the domain of building automation systems. By expressing service compositions as recipes together with selection rules, IoT components can be dynamically updated and recomposed. This allows the automatic integration of new components into existing compositions without requiring user interaction. The choreography approach remove single points of failure, and leverages the computation power of network nodes. The system is reasonably efficient and scales acceptably with growing numbers of devices. The quadratic scaling behavior is problematic with very large systems, but performance tuning of the triple store will improve the scaling behavior of the system. Additionally, the recipe concept is limited in the compositions it can express,

only allowing the composition of REST services, without the ability to add custom properties or scripts. By extending the recipe context in the future, we will be able to express a wider range of automation services, and further reduce the reliance on centralized control algorithms.

Additionally, we are working on using the OSR mechanism as a building block for reliability of orchestrations. This is a crucial task, since failure of single IoT components may remain unnoticed (or noticed quite late) in distributed workflows. Using the OSR concept, failures in the orchestration can be automatically corrected if suitable components are available. This research will make our system ready for more complex deployments.

References

[1] B. Butzin, F. Golatowski, C. Niedermeier, N. Vicari, and E. Wuchner, "A model based development approach for building automation systems", in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, Sep. 2014, pp. 1–6. DOI: 10.1109/ETFA.2014.7005365.

[2] A. S. Thuluva, K. Dorofeev, M. Wenger, D. Anicic, and S. Rudolph, "Semantic-based approach for low-effort engineering of automation systems", in *On the Move to Meaningful Internet Systems. OTM 2017 Conferences*, ser. Lecture Notes in Computer Science, Springer, Cham, Sep. 23, 2017, pp. 497–512, ISBN: 978-3-319-69458-0. DOI: 10.1007/978-3-319-69459-7_33. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-69459-7_33 (visited on 11/28/2017).

[3] A. S. Thuluva, A. Bröring, G. P. Medagoda, H. Don, D. Anicic, and J. Seeger, "Recipes for IoT applications", in *Proceedings of the Seventh International Conference on the Internet of Things*, ser. IoT '17, New York, NY, USA: ACM, 2017, 10:1–10:8, ISBN: 978-1-4503-5318-2. DOI: 10.1145/3131542.3131553. [Online]. Available: http://doi.acm.org/10.1145/3131542.3131553 (visited on 12/21/2017).

[4] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu, "Web services composition: A decade's overview", *Information Sciences*, vol. 280, pp. 218–238, Oct. 1, 2014, WOS:000339132700014, ISSN: 0020-0255. DOI: 10.1016/j.ins.2014.04.054.

[5] D. Martin, M. Burstein, D. Mcdermott, S. Mcilraith, M. Paolucci, K. Sycara, D. L. Mcguinness, E. Sirin, and N. Srinivasan, "Bringing semantics to web services with owl-s", *World Wide Web*, vol. 10, no. 3, pp. 243–277, 2007.

[6] J. Kopecký, K. Gomadam, and T. Vitvar, "hRESTS: An HTML microformat for describing RESTful web services", in *Web Intelligence and Intelligent Agent Technology, 2008. WI-IAT'08. IEEE/WIC/ACM International Conference on*, IEEE, vol. 1, 2008, pp. 619–625.

[7] R. Verborgh, T. Steiner, D. Van Deursen, R. Van de Walle, and J. G. Vallés, "Efficient runtime service discovery and consumption with hyperlinked restdesc", in *Next Generation Web Services Practices (NWeSP), 2011 7th International Conference on*, IEEE, 2011, pp. 373–379.

[8] P. Persson and O. Angelsmark, "Calvin – merging cloud and IoT", *Procedia Computer Science*, The 6th International Conference on Ambient Systems, Networks and Technologies (ANT-2015), the 5th International Conference on Sustainable Energy Information Technology (SEIT-2015), vol. 52, pp. 210–217, Supplement C Jan. 1, 2015, ISSN: 1877-0509. DOI: 10.1016/j.procs.2015.05.059. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050915008595 (visited on 11/06/2017).

[9] N. K. Giang, M. Blackstock, R. Lea, and V. C. M. Leung, "Developing IoT applications in the fog: A distributed dataflow approach", in *2015 5th International Conference on the Internet of Things (IOT)*, Oct. 2015, pp. 155–162. DOI: 10.1109/IOT.2015.7356560.

[10] M. Burkert, H. Krumm, and C. Fiehe, "Technical management system for dependable building automation systems.", in *20th IEEE Conference on Emerging Technologies & Factory Automation, ETFA 2015, Luxembourg, September 8-11, 2015*, 2015, pp. 1–8. DOI: 10.1109/ETFA.2015.7301656. [Online]. Available: http://dx.doi.org/10.1109/ETFA.2015.7301656.

[11] A. Bröring, S. Schmid, C. K. Schindhelm, A. Khelil, S. Käbisch, D. Kramer, D. L. Phuoc, J. Mitic, D. Anicic, and E. Teniente, "Enabling IoT ecosystems through platform interoperability", *IEEE Software*, vol. 34, no. 1, pp. 54–61, Jan. 2017, ISSN: 0740-7459. DOI: 10.1109/MS.2017.2.

[12] R. Guha, D. Brickley, and S. Macbeth, "Schema. org: Evolution of structured data on the web", *Communications of the ACM*, vol. 59, no. 2, pp. 44–51, 2016.

[13] R. Hodgson and P. J. Keller, "Qudt-quantities, units, dimensions and data types in owl and xml", *Online (September 2011) http://www. qudt. org*, p. 34, 2011.

[14] S. Schmid, A. Bröring, D. Kramer, S. Käbisch, A. Zappa, M. Lorenz, Y. Wang, A. Rausch, and L. Gioppo, "An architecture for interoperable IoT ecosystems", in *Interoperability and Open-Source Solutions for the Internet of Things: Second International Workshop, InterOSS-IoT 2016, Held in Conjunction with IoT 2016, Stuttgart, Germany, November 7, 2016, Invited Papers*, I. Podnar Žarko, A. Broering, S. Soursos, and M. Serrano, Eds., DOI: 10.1007/978-3-319-56877-5_3, Cham: Springer International Publishing, 2017, pp. 39–55, ISBN: 978-3-319-56877-5. [Online]. Available: https://doi.org/10.1007/978-3-319-56877-5_3.

[15] S. Harris, A. Seaborne, and E. Prud'hommeaux, *SPARQL 1.1 query language*, ser. W3C Recommendations. W3C, 2013. [Online]. Available: https://www.w3.org/TR/sparql11-query/.