Integrating IoT Devices Based on Automatically Generated Scale-Out Plans

Kálmán Képes, Uwe Breitenbücher, Frank Leymann

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{kepes, breitenbuecher, leymann}@iaas.uni-stuttgart.de

BibTeX:

```
@inproceedings{KepesScaleOut2017,
  author    = {K\'{a}lm\'{a}n K\'{e}pes and Uwe Breitenb\"{u}cher and
               Frank Leymann},
  title     = {Integrating IoT Devices Based on Automatically Generated Scale-
               Out Plans},
  booktitle = {2017 IEEE 10th Conference on Service-Oriented Computing and
               Applications, SOCA 2017, 22-25 November 2017, Kanazawa, Japan},
  year      = {2017},
  pages     = {155--163},
  doi       = {10.1109/SOCA.2017.29},
  publisher = {IEEE Computer Society}
}
```

Integrating IoT Devices based on Automatically Generated Scale-Out Plans

Kálmán Képes, Uwe Breitenbücher, Frank Leymann
Institute of Architecture of Application Systems, University of Stuttgart
70569 Stuttgart, Germany
{kepes, breitenbuecher, leymann}@iaas.uni-stuttgart.de

***Abstract*—The Internet of Things (IoT) is a trend of connecting physical objects to the Internet. IoT applications running together with interconnected objects enable the vision of gathering data and based on that act in physical environments with minimal human intervention. Maintaining such IoT applications needs to incorporate different software and hardware components to build up the network of things that delivers the data of sensors and controls actuators. However, the effort of installation, configuration and management of a growing number of IoT devices increases the complexity of integrating such heterogeneous hard- and software. Automating the configuration of IoT devices, i.e., reducing the effort of configuring the devices in the target vicinity, reduces cost, complexity and the occurrence of errors. We propose the usage of automatically generated Scale-Out Plans on the basis of the OASIS standard TOSCA. The overall approach is based on the ability to mark regions inside TOSCA application models, that must be scaled out when new devices are registered for the overall application. From this marked model we generate Scale-Out Plans that are able to create instances of software components enabling the automated configuration of IoT devices. We prove the technical feasibility of our approach by a prototypical implementation based on our previous work.**

***Index Terms*—Internet of Things, Middleware, Systems modeling, Large-scale systems, Scalability, System integration**

I. Introduction

The trend of connecting physical objects with the Internet gave birth to the term Internet of Things (IoT). IoT envisions applications running with interconnected objects working together to gather data and act in environments by controlling different actuators to enable new functionality with minimal human intervention [1]. This trend enables the development of applications that use data globally. During the maintenance of such IoT applications numerous software and hardware components are in place that together build up the network of things that is able to deliver data captured by different sensors and act in physical environments with the help of actuators.

However, with the growing amount of IoT devices the effort to install, configure and manage, thus binding a device to an application, is increasing in complexity, as applications using these devices must be able to integrate not just heterogeneous hardware and software components but as well be able to scale accordingly, e.g, by using Cloud Computing resources [2]. A single application that measures the temperature of a neighborhood must be able to manage dozens of devices of which each has to manage itself multiple sensors and actuators. Automating the binding of IoT devices and software, i.e., reducing the effort of installing and configuring the devices in the target vicinity, reduces cost, complexity and the occurrence of errors. We propose an approach that is based on the OASIS standard TOSCA, by enabling to mark so-called Scale-Out Regions by a modeler inside an application model. From this marked application model we generate Scale-Out Plans that are able to create instances from the components inside the marked regions, which in turn, enables to automatically install and configure IoT devices. By enabling modelers to generate Scale-Out Plans, that enable configuration and scaling of applications automatically, the complexity of modeling the behavior of an application when incorporating additional devices and sensors is reduced. In overall, automating the configuration of devices reduces time and enabling the standardized modeling of an application by using TOSCA, allows the reuse of components which in turn enables to incorporate heterogeneous devices. Enabling to incorporate commodity software and hardware such as the open-source home automation platform, and thus IoT Middleware, Home Assistant [3] and the IoT Device Raspberry Pi [4] reduces cost and automating their configuration reduces it even further. Message Brokers for publishing and subscribing to sensor data are also an integral part of IoT applications. In IoT, Message Brokers, such as Mosquitto [5], are employing typical IoT protocols, such as the OASIS standard MQTT [6], which are characterized to be suitable for devices with low bandwith, low power, high latency and other constraining factors. Our approach reduces the time of integrating additional IoT devices in running applications, hence, enabling IoT applications to scale out to additional devices.

This paper is structured as follows: In Section II, we describe a motivating scenario of a typical IoT application, consisting out of devices installed in different environments. In the same section we also describe the OASIS standard TOSCA by describing how to model the presented scenario. Afterwards, we describe our approach of solving the motivational scenario by automating the configuration of IoT devices with our generic method of generating Scale-Out Plans in Section III. In Section IV we describe our validation of our approach based on a prototypical implementation of our method within the OpenTOSCA Ecosystem. In Section V we discuss work related to IoT device integration, Section VI concludes the paper and outlines possible future work.

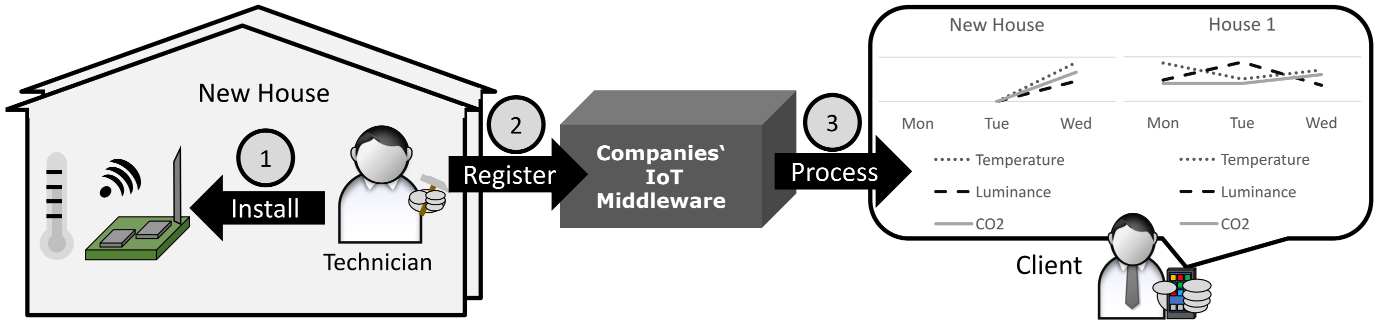

Fig. 1. Motivating scenario depicting a technician who installs a new device and registers it at the IoT Middleware, enabling clients to view their house data.

II. Motivating Scenario and TOSCA

In this section we describe our motivating scenario of a company's business model built around the IoT management of property of different clients (see in Figure 1). A new offering of the company is to monitor different properties of the managed house such as temperature, CO2 emissions and luminance (see charts in Figure 1). The data is gathered by the companies IoT middleware to be further processed to offer different additional services such as informing of energy consumption and to guide a more efficient use. The monitoring shall be done with devices that are able to control different types of sensors at the site (see house in Figure 1) and send it to a service of the company managing it (see IoT Middleware in Figure 1). As the company intends to grow, the system must be able to incorporate additional devices at runtime of the application. The company intends to incorporate new devices and sensors with three steps: (i) install the devices and sensors by a technician in the intended environment (see step 1 in Figure 1), (ii) register the device with its attached sensors by configuring it to send the measured data to the companies' IoT Middleware (see step 2 in Figure 1) and afterwards, (iii) the newly registered device is detected by processing the received data (see step 3 in Figure 1). One issue with the incorporation of devices is the installation of them itself, which entails setting up a device at the new environment that has to be monitored. After the installation process, the configuration to read the sensors and sending their measured data must be taking place. This requires the installation of software drivers that can read values from each attached sensor or issue commands to actuators. Both, can differ in their type, such as temperature or light sensors, and can be made by a wide range of manufacturers. Additionally, measured data must be send by software adapters to the target IoT Middleware with its used protocol and expected data format. In summary the integration of devices is a complex process, as companies cope with heterogeneous devices, sensors and middlware each with own drivers, adapters, protocols and data formats.

To model the described scenario we introduce the OASIS standard Topology and Orchestration Specification for Cloud Applications" (TOSCA) [7], [8]. TOSCA is a standard enabling the modeling of composite cloud applications in so-called *Topology Templates* (see Figure 2). Topologies specify components as *Node Templates* while the relations between them are specified as *Relationship Templates*. Each of them have an associated *Node-* and *Relationship Type* defining configuration parameters that can be set on their templates, e.g., a virtual machine Node Type may expose configuration parameters to set user names and passwords, which are specified on the Node Templates of a Node Type. The order of provisioning the modeled Node Templates is based on the used Relationship Types. In our scenario, the "hostedOn" Relationship Type indicates that a Node Template must be installed on another. Another type of relation is the dependency between components called "dependsOn", which allows, e.g, to model the need for a Python Application to have a Python Runtime available. The "connectTo" relationship in our scenario enables the modeling of connections between two components, e.g., the sending of sensor data from a Python Application to a Topic. The implementation of Node Types is based on so-called *Node Type Implementations* that combine *Implementation- (IAs)* and *Deployment Artifacts (DAs)*. IAs implement the management operations of a Node Type, e.g., a virtual machine Node Type exposes operations to upload files, which can be implemented by Web Services. DAs specify the artifacts that make up the business functionality of the components, e.g., a virtual machine image for the Ubuntu 14.04 Node Template. To specify management behavior of applications, TOSCA enables the use of *Management Plans* that can instantiate, alter and terminate topologies. To provision,

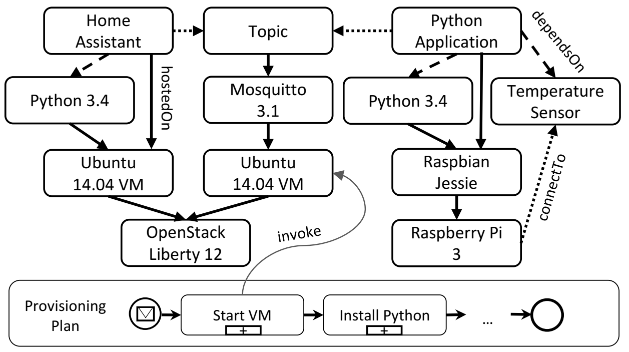

Fig. 2. Example TOSCA Topology Template of our motivating scenario.

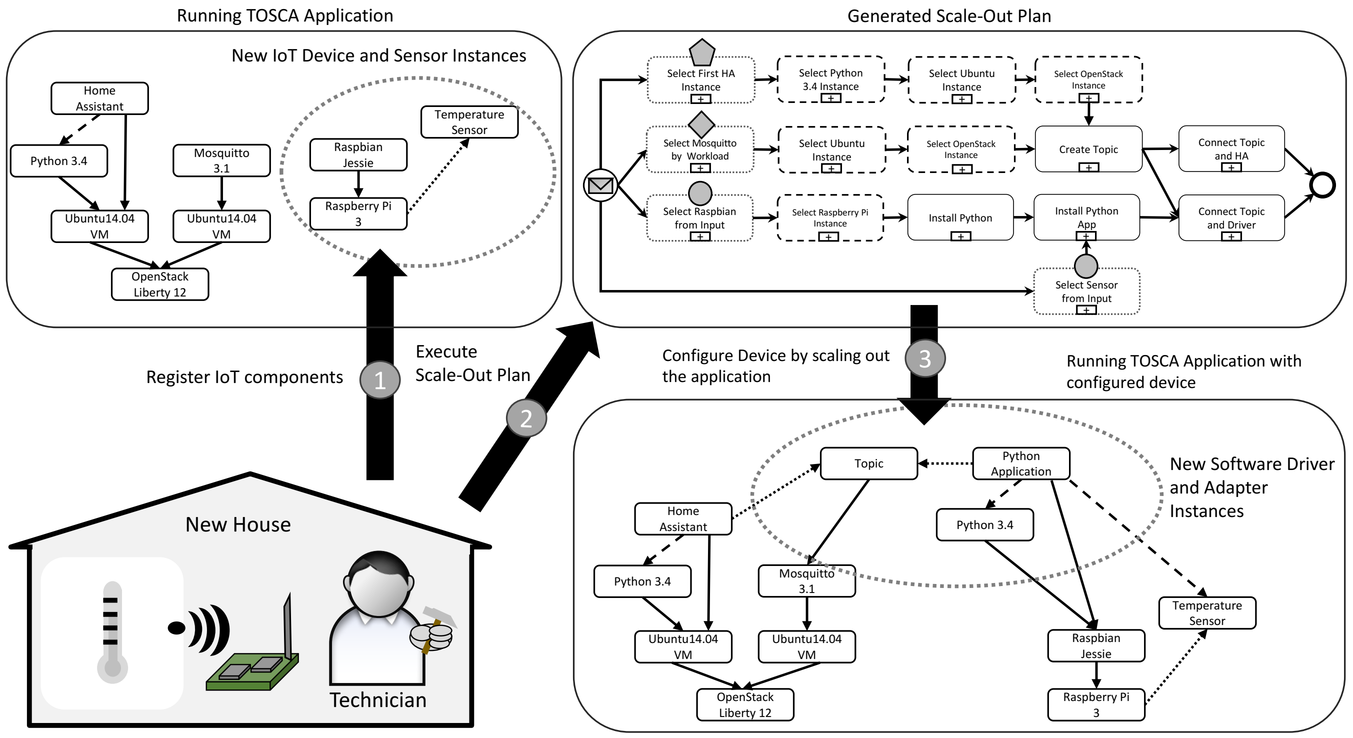

Fig. 3. Overview of our approach for registering, configuring and integrating IoT devices into running applications, by executing a generated Scale-Out Plan.

e.g., a virtual machine on an OpenStack cloud, a plan, in this case a *Provisioning Plan*, would invoke the exposed operations of an interface that is specified on an OpenStack Node Type (see “Start VM” subprocess in Figure 2). The plans may use configuration parameters of Node- and Relationship Templates as the input of management operations to create instances with the specified configurations. To create an instance, e.g., of the Home Assistant Node Template and its infrastructure (Python 3.4 and Ubuntu 14.04 VM) a plan would first start a virtual machine by invoking the “Start VM” operation. Afterwards, it would use operations of the Ubuntu Node Type, e.g., “Transfer File” and “Run Script” to upload the necessary binaries (DAs) and execute the installation scripts (IAs) on the started virtual machine. To model our scenario, a “Home Assistant” Node Template is of the Node Type “Python Application” that depends on a “Python 3.4” environment (see in Figure 2). To receive messages from the devices we specify an infrastructure to receive sensor data with the MQTT Broker Mosquitto, which enables creating Topics to either publish and subscribe data to (see Topic, Mosquitto 3.1, Ubuntu 14.04 in Figure 2). Both are hosted on an Ubuntu virtual machine which is started on an OpenStack Cloud at provisioning time (see Ubuntu 14.04 and OpenStack Liberty 12 in Figure 2). The sending of sensor data is achieved by modeling the device with its operating system (see Raspberry Pi and Raspbian Jessie in Figure 2), the needed software for reading and sending sensor values (see Python 3.4 and Python Application in Figure 2), and the sensor itself (see Temperature Sensor Node Template in Figure 2).

III. Automated Scale-Out Plan Generation

In the following we give an overview of our approach of solving the scenario by using *Scale-Out Plans* to integrate IoT devices into running IoT applications. Afterwards, we define so-called *Scale-Out Regions* that define the components which must be scaled out in Subsection III-A. To give insight, we present previous work in Subsection III-B, on how to generate Provisioning Plans able to instantiate a whole TOSCA application. Finally, we describe our method for generating Scale-Out Plans from specified Scale-Out Regions in Subsection III-C.

To automate the presented scenario of a company managing houses by using IoT devices, we identified the following steps which have to be considered: At first, a technician must (i) setup the device and the needed sensors at the clients' house, afterwards, the device and its sensors are (ii) configured by installing software components onto the devices to read sensor data and send it to companies' application. After the installation and configuration by the technician is finished, the sent sensor data is then (iii) processed in the companies application to, e.g., enable views of the measured data (see Section II). To ease the installation of software components and their configuration on IoT devices of the scenario, we propose the use of Scale-Out Plans that are able to provision and configure software components on running component instances with the following approach (see Figure 3): A technician installs the IoT device and its sensors at the target house, but instead of installing and configuring the software components himself, he registers the devices and sensors as TOSCA Node- and Relationship Template instances at a

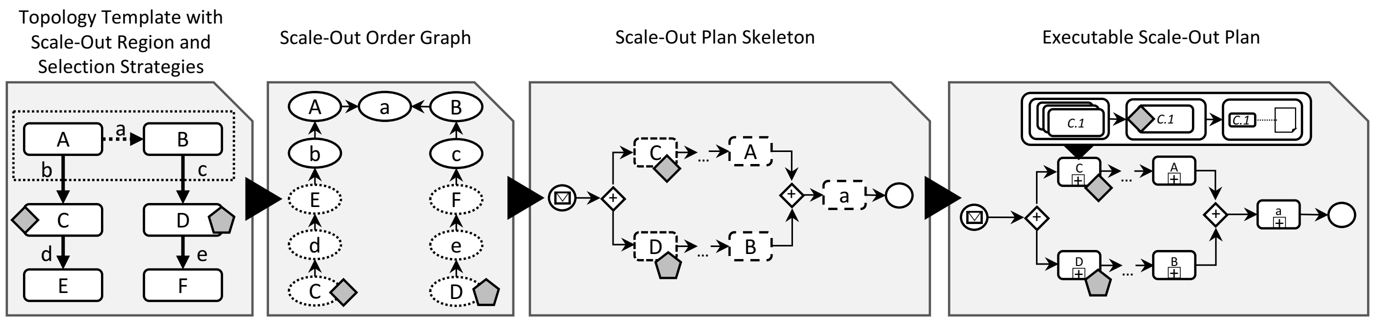
Topology Template with Scale-Out Region and Selection Strategies
Scale-Out Order Graph
Scale-Out Plan Skeleton
Executable Scale-Out Plan
A
B
a
b
c
C
D
d
e
E
F
C.1

Fig. 4. Three step method for generating Scale-Out Plans for TOSCA Topology Templates marked with Scale-Out regions and Selection Strategies.

TOSCA Runtime Environment that manages the companies' deployed application (see 1 in Figure 3). Afterwards, the technician requests the execution of a generated Scale-Out Plan that is able to install and configure the needed software on the registered device and sensor instances (see 2 in Figure 3). This is achieved by instantiating the Node- and Relationship Templates that read sensor data and send it to the companies' IoT middleware used by their clients (see 3 in Figure 3). To enable this approach, we propose the use of Scale-Out Regions to mark components that have to be scaled out by a Scale-Out Plan. We then apply a generic method that is able to generate Scale-Out Plans from these Scale-Out Regions, enabling to scale-out an application at runtime.

A. Scale-Out Regions and Scale-Out Plans

Scaling can be classified into techniques that can either adapt the application in a *proactive* and *reactive* way [9] [10]. Concrete scaling methods can be categorized into horizontal ("scale-out") and vertical ("scale-up") scalings, where either the scaling is achieved by adding additional instances of a component, hence scaling out horizontally, or increasing the resources available to running components, hence scaling up vertically. In our approach we focus on scaling out, i.e., horizontally scaling out TOSCA applications, which we achieve by using so-called Scale-Out Regions and Scale-Out Plans described in the following.

a) Scale-Out Region: A Scale-Out Region is a weakly connected subset of Node- and Relationship Templates of a Topology Template, which shall be horizontally scaled by creating new instances of each Node- and Relationship Template inside an already running instance of a Topology Template, i.e., scale out an application by installing new instances of software components. Additionally, a Scale-Out Region must specify the proper selection of running instances that are not part of the components that shall be scaled out. This is necessary as some newly created component instances must be installed on already running ones, e.g., installing a Python Application on a running Raspberry Pi. This is enabled by defining *Selection Strategies* that define how a single instance of a Node Template must be selected at runtime. These strategies must be attached to the Node Templates that are connected to the region of the components that shall be scaled out. One class of Node Templates that have to be annotated with a strategy are connected to the region through outgoing infrastructure relationships, such as "hostedOn" and "dependsOn". In this case, the selection of Node Template instances may specify that an instance with the lowest CPU load (e.g. when selecting virtual machines) must be selected. Another selection must be taking place for Node Templates that are connected through a "connectTo" relationship to the specified region, as the scaled out components must be connected to an instance. In this case the selection, e.g, may be driven by Service Selection algorithms that rank the different Node Template instances according to their request load [11].

To describe our approach we define a digraph based metamodel for Topology Templates and Scale-Out Regions, whereby a Topology Template is an acyclic and possibly disconnected digraph. In contrast, a Scale-Out Region must be at least a weakly connected digraph to enforce selecting proper subgraphs of the topology, disabling the possibility of modeling a region that instantiates an application and its virtual machine, but not the needed middleware in between, e.g., by creating a region consisting of a Topic and Ubuntu Node Templates, but missing the needed Mosquitto Node Template that can host the topic. Let $\mathcal{T}$ be the set of all topologies and $\mathcal{SR}$ be the set of all Scale-Out Regions, then $t \in \mathcal{T}$ and $sr \in \mathcal{SR}$ are defined as the following tuples:

$$t = (V_{topo}, E_{topo}), sr = (V_{sr}, E_{sr}) \tag{1}$$

- V_{topo} is the set of components in t, whereby each $v_i \in V_{topo}$ represents a component of the application.
- $E_{topo} \subseteq V_{topo} \times V_{topo}$ is the set of component relations in t, consisting out of $e_i = (v_s, v_t) \in E_{topo}$ with v_s and v_t being the source and target, respectively.
- $V_{sr} = (V_{region} \cup V_{strat}) \subseteq V_{topo}, V_{region} \cap V_{strat} = \emptyset$ is the set of components in sr, whereby each $v_r \in V_{region}$ represents a component inside a Scale-Out Region which must be instantiated and each $v_{strat} \in V_{strat}$ represents a component which must be selected from available instances at runtime. The set V_{sr} must fulfill that $(\exists v \in V_{region} \exists e \in E_{topo}(\pi_1(e) = v \land \pi_2(e) \notin V_{region})) \Rightarrow (\pi_2(e) \in V_{strat} \land e \in E_{sr})$. This enforces the following: for any $v_r \in V_{region}$ which is connected to a $v_t \in$

$V_{topo} \land v_t \notin V_{region}$ through an edge $(v_r, v_t) \in E_{topo}$ it must be ensured that $v_t \in V_{strat}$ and $(v, v_t) \in E_{sr}$, enforcing that every needed node connected to the region, that has to be strategically selected, is specified.

- $E_{sr} \subseteq (V_{region} \times V_{region} \cup V_{region} \times V_{strat})$ is the set of component relations in sr, where each $e_j = (v_s, v_t) \in E_{sr}$ with v_s and v_t being the source and target, respectively. Additionally, a Scale-Out Region distinguishes between edges $e_r = (v_i, v_j), v_i, v_j \in V_{region}$ representing the relations between the components that have to be instantiated, and edges that connect these to the components $v_{strat} \in V_{strat}$ that have to be strategically selected at runtime, hence, edges of $e_j \in E_{sr}$ mustn't connect two components $v_{strat} \in V_{strat}$.

In summary, Scale-Out Regions sr are weakly connected subgraphs of Topology Templates t, which contain Node Templates $v_r \in V_{region}$ that shall be instantiated and Node Templates $v_{strat} \in V_{strat}$ that have to be selected at runtime according to a specified strategy. The component relations $e_j \in E_{sr}$ either model the subgraph that shall be instantiated ($e_j \in V_{region} \times V_{region}$) or are edges that connect the modeled region to the Node Templates for which an instance has to be selected at runtime ($e_j \in V_{region} \times V_{strat}$).

b) Scale-Out Plan: A Scale-Out Plan creates a single instance for each Node- and Relationship Template specified in a Scale-Out Region and connects these to or installs them on selected Node- and Relationship Template instances (See on the right of Figure 5). In other words, a Scale-Out Plan combines the provisioning of Node- and Relationship Templates with the selection of instances at the border of a Scale-Out Region at runtime. The provisioning part is based on the specified templates which are part of the Scale-Out Region, meaning that the configuration of the components intended to scale out are equal to instances created at provisioning time of the overall application. In contrast to this, the selection of template instances must be configurable as different types of templates imply different selection methods. Thus, a Scale-Out Plan must be able to select instances according to the specified Selection Strategies of a Scale-Out Region. Further, this runtime selection must be achieved before provisioning the Node- and Relationship Templates of the Scale-Out Region itself, as instances of these must be installed and configured on the selected instances. Additionally to the instance selection based on Selection Strategies, additional Node- and Relationship Template instances must be retrieved by a Scale-Out Plan on which the strategically selected Node Templates instances are hosted on. This is needed as some templates that are provisioned in the region may need the instance data of the component instances, such as the IP address and user credentials of a virtual machine. In other words, a Scale-Out Plan must retrieve the Node- and Relationship Template instances that are part of the infrastructure of the strategically selected instances, e.g., by following the outgoing "hostedOn" Relationship Templates in Topology Template of strategically selected Node Templates and according to the structure, select the appropriate instances.

To clarify what kind of activities a Scale-Out Plan has to execute we define a digraph based metamodel which we call *Scale-Out Order Graphs (SOG)* where its nodes represent abstract activities and the edges represent the control flow of the overall plan. While the abstract activities define either that a Node- and Relationship Template is instantiated or selected from running instances, the edges represent the order of these activities. Let $\mathcal{SOG}$ be the set of all Scale-Out Order Graphs, then $sog \in \mathcal{SOG}$ is defined as the following tuple:

$$sog = (V_{sog}, E_{sog}) \tag{2}$$

- $V_{sog} = V_{provisioning} \cup V_{selection} \cup V_{recursive}$, $V_{provisioning}, V_{selection}, V_{recursive}$ being pairwise disjoint is the set of abstract activities in sog, whereby $v_i \in V_{provisioning}$ represent an abstract activity that indicates that a Node- or Relationship Template must be instantiated in the sog . The abstract activities $v_j \in V_{selection}$ are used to select instances of Node- and Relationship Templates according to the specified Selection Strategies, while $v_k \in V_{recursive}$ represent abstract activities that select Node- and Relationship Template instances that are connected to the strategically selected instances as infrastructure components.
- $E_{sog} \subseteq (V_{selection} \times V_{recursive}) \cup (V_{selection} \times V_{provisioning}) \cup (V_{recursive} \times V_{provisioning}) \cup (V_{provisioning} \times V_{provisioning})$ is the set of control flow edges in sog. The subsets $V_{selection} \times V_{provisioning}$ and $V_{recursive} \times V_{provisioning}$ enforce the control flow of the sog to select the Node- and Relationship Template instances before any provisioning activities can be executed. In addition, with the subset $V_{selection} \times V_{recursive}$ we enforce that a strategic selection always occurs before any recursive selection of infrastructure Node- and Relationship Template instances is taking place. The control flow of the provisioning itself is mapped to the edge subset $V_{provisioning} \times V_{provisioning}$ representing the order of instantiating the Node- and Relationship Template instances of a Scale-Out Region.

In summary, a Scale-Out Plan executes different activities to instantiate Node- and Relationship Template instances ($V_{provisioning}$), strategically select Node Template instances ($V_{selection}$) and their infrastructure as instances of Node- and Relationship Templates ($V_{recursive}$).

B. Provisioning Plan Generation

The generation of provisioning activities $v \in V_{provisioning}$ and the edges $e \in V_{provisioning} \times V_{provisioning}$ of a $sog \in \mathcal{SOG}$ is based on our previous work, which is described in the following. Breitenbücher et al. [12] present an approach to automatically generate Provisioning Plans for TOSCA Topology Templates. The approach itself is split into three steps which are: (i) Generation of a Provisioning Order Graph (POG) from the given Topology Template, (ii) Transformation of the POG into a Provisioning Plan Skeleton (PPS) and, as the last step, (ii) complete the PPS into an executable Provisioning

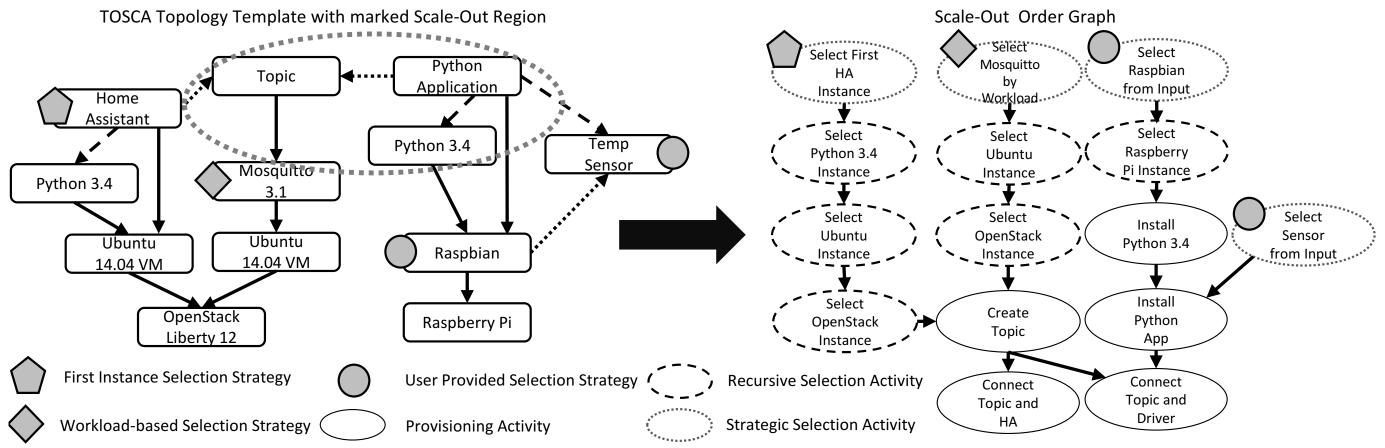

Fig. 5. Scale-Out Order Graph Generation for the Topology Template of our motivating scenario with Scale-Out Region and Selection Strategies.

Plan by a set of Provisioning Logic Providers (PLPs). As the first step of the approach a POG is generated by creating an abstract *Provisioning Activity* for each Node- and Relationship Template of a topology ($v \in V_{provisioning} \subseteq V_{sog}$). The control flow connections ($e \in V_{provisioning} \times V_{provisioning}$) between the abstract provisioning activities is based on the semantics of the Relationship Types used by the Relationship Templates. A "hostedOn" type enforces the provisioning order to provision first the target Node Template of such a relation and the source after. In contrast, a "connectTo" Relationship Type forces the provisioning to first provision both source and target Node Templates before a connection between the two components can be made. After generating a POG from the Topology Template it is transformed into a language-dependent PPS that must be equal to the order of the Provisioning Activities of the POG. The PPS is at the end a language-dependent skeleton with placeholder activities that resemble the order of the activities of the generated POG. This PPS is completed into an executable Provisioning Plan in the final step of the approach by so-called *Provisioning Logic Providers (PLPs)* that are able to process the Node- and Relationship Templates of the Topology Template based on their Node- and Relationship Type respectively and add executable Provisioning Activities to the generated PPS by replacing the specified placeholder activities. After this step, if all placeholders are replaced by PLPs, the original PPS is transformed to an executable Provisioning Plan.

C. Scale-Out Plan Generation

In this subsection we describe how to generate a Scale-Out Plan from a Scale-Out Region. We propose a 3 step method that is shown in Figure 4. The three steps of the method are the (a) *Scale-Out Order Graph Generation*, (b) *Scale-Out Plan Skeleton Transformation* and (c) the *Scale-Out Plan Completion* step, which in the following we describe in detail.

a) Scale-Out Order Graph Generation: As the first step a *Scale-Out Order Graph (SOG)* is generated from the given Topology Template marked with Scale-Out Regions and specified Selection Strategies (See Scale-Out Order Graph in Figure 4). A SOG represents an order of abstract activities of selecting and provisioning Node- and Relationship Template instances, the generation of such a SOG, we will now describe based on our motivating scenario (See Figure 5) and an algorithm (See Figure 6) in detail. For our scenario, we marked a region which contains the Python Application, its Python 3.4 Runtime and the Topic the sensor data will be sent to (See on the left in Figure 5). As these components will be installed on running component instances a selection at the border of the region must occur. In our scenario, we annotate the Raspbian and Temp Sensor Node Templates with a "User Provided Selection Activitiy" which selects an component instance based on the input of the generated Scale-Out Plan. This enables the technician of our scenario to reference the installed devices and sensors in the input of the plan. To create a Topic for the sensor data an instance of the Mosquitto Broker must be selected, this is achieved by annotating the Node Template with the "Workload-based Selection Strategy", that selects an instance based on the workload, enabling the company to start additional brokers, thus, scale the messaging infrastructure independent of the technicians. Integrating the device to the processing part of our scenario is achieved with the last annotation called "First Instance Selection Strategy" on the Home Assistant Node Template. To connect the data to the application, the annotated Selection Strategy only selects the first instance of a Home Assistant Node Template, which in our example scenario is only a single centralized middleware for the companies' customers, i.e., the application will always have only a single instance of it at runtime.

The generation of a $sog \in \mathcal{SOG}$ (See Definition 2 in III-A) from a Scale-Out Region $sr \in \mathcal{SR}$ (See Definition 1 in Subsection III-A) of our scenario is achieved by executing the algorithm in Figure 6. In the first phase of the algorithm we reuse the Provisioning Order Graph Generation of our previous work (See Line 2 in Figure 6 and Subsection III-B), which generates our provisioning activities $v \in V_{provisioning}$ of sog. The different abstract selection activities $V_{selection}$, $V_{recursive} \subset V_{sog}$ are added to V_{sog} like the following: For

```
procedure: genSOG($V_{topo}$, $E_{topo}$, $V_{region}$, $E_{region}$, $V_{strat}$)
1:  Let $G_{sog}$ = ($V_{sog}$ =, $E_{sog}$) an empty digraph.
2:  $G_{sog} \leftarrow generatePOG(V_{region}, E_{region})$ based on [12]
3:  for $\forall v \in V_{strat}$ do
4:      $V_{sog} \leftarrow V_{sog} \cup strat_v$
5:      $p = \{((v,a),(a,b),..,(y,z)) | w_i = \pi_i(x), w_i \in E_{topo} \wedge$
        $w_i$ is an infrastructure relation$\}$
6:      if $p = \emptyset$ then
7:          for $\forall e \in E_{region}, \pi_2(e) = v$ do
8:              $E_{sog} \leftarrow E_{sog} \cup (strat_v, prov_e)$
9:          end for
10:     else
11:         for $\forall x \in p$ do
12:             for $\forall w_i = \pi_i(x)$ do
13:                 $V_{sog} \leftarrow V_{sog} \cup recursive_{w_i}$
14:                 $V_{sog} \leftarrow V_{sog} \cup recursive_{\pi_2(w_i)}$
15:                 $E_{sog} \leftarrow E_{sog} \cup (recursive_{w_i}, recursive_{\pi_2(w_i)})$
16:                 $E_{sog} \leftarrow E_{sog} \cup (recursive_{\pi_1(w_i)}, recursive_{w_i})$
17:             end for
18:             for $\forall e \in E_{region}, \pi_2(e) = v$ do
19:                 $E_{sog} \leftarrow E_{sog} \cup (recursive_{\pi_{|x|}(x)}, prov_e)$
20:             end for
21:         end for
22:     end if
23: end for
```

Fig. 6. Pseudocode for generating a Scale-Out Order Graph.

each Node Template annotated with a Selection Strategy a Strategic Selection Activity will be added (See Line 4 in Figure 6). Afterwards, the algorithm determines all outgoing paths of the Node Template that are connected through infrastructure edges, such as a "hostedOn" or "dependsOn" relation (See Line 5 in Figure 6). If such paths don't exist, the algorithm will directly connect the selection activity $strat_v$ to the provisioning activity of all edges that have the strategically selected node as the target node (See Lines 6-9 in Figure 6). If such paths exist, the algorithm generates Recursive Selection Activities for each Node- and Relationship Template of such a path and additionally connects them in the same order (See Lines 11-17 in Figure 6). In the final phase of the algorithm, the generated paths must be connected to the Provisioning Activities of the region. This is achieved by connecting the last Recursive Selection Activity of a path with the Provisioning Activities of the Relationship Templates connected to the annotated Node Templates (See Lines 18-20 in Figure 6).

b) Scale-Out Plan Skeleton Transformation: After we generated the SOG from the Topology Template the next step is to transform the SOG into a language-dependent Scale-Out Plan Skeleton (SPS) (See Scale-Out Plan Skeleton in Figure 4). This step is analogous to our previous approach, where a language-dependent Plan Skeleton containing placeholders in the order of the abstract graph is generated. In our previous and the extended approach of this work, these skeletons are generated for a target language and preserve the order of a POG/SOG it is created from, but instead of abstract activities the skeleton contains a placeholder of the target language, e.g., in BPEL these placeholder activities might be <*empty*> activities inside a <*flow*> activity that preserves the order by adding <*links*> according to the SOG. Additional code, such as, variables that will hold instance data at runtime or basic input and output parameters can be added in this step.

c) Scale-Out Plan Skeleton Completion: The last step completes a SPS by adding language-dependent and executable activities with so called *Instance Selection Logic Providers (ISLP)* and by reusing *Provisioning Logic Providers (PLPs)* from our previous work (See Executable Scale-Out Plan in Figure 4). For each activity in the SPS which is used to provision a Node- or Relationship Template we use a PLP as described by Subsection III-B, in the following we describe the usage of ISLPs in detail. These are able to process the Node Templates with annotated Selection Strategies to generate executable activities for the PPS, that implement the requested Selection Strategy. An ISLPs is therefore specified to be able to support a set of Selection Strategies for at least a single Node Type. An example for such an ISLP implementing the "User-Provided Selection Strategy" (see in Figure 5) would add activities that achieve the selection based on the input of the Scale-Out Plan. This could be done by simply adding an additional input parameter and activities reading the given data from the input. A more sophisticated ISLP could create a task inside a Workflow Management System which starts the whole task of installing a new device by a technician, and only continues execution if the task is finished. ISLPs can be implemented in many ways but are restricted to select only a single instance of all available instances. This ensures that the generated Scale-Out Plan in overall only generates activities that handle a single Node- or Relationship Template by either provisioning or selecting it. The Recursive Selection Activities inside a PPS can be handled in a generic way, by selecting an instance of a Node- and Relationship Template based on the Topology Template model and previously selected instances, e.g., by reading instance data inside the plan.

In summary, we extended our previous approach by adding additional activities to select instances at runtime. The extension is based on the specification of Scale-Out Regions with Selection Strategies. From a Scale-Out Region we generate a Scale-Out Order Graph that specifies the provisioning order of the Node- and Relationship Templates in an abstract way. In addition to provisioning, a Scale-Out Order Graph contains abstract Strategic Selection and Recursive Selection Activities for selecting Node- and Relationship Templates that are used to connect or install the provisioned components. After generating a Scale-Out Order Graph we transform it into a language-dependent Scale-Out Plan Skeleton which preserves the order of the original graph but instead of abstract activities replaces these with placeholders of the target language. The skeleton is afterwards completed by a set of Provisioning Logic Providers and Instance Selection Logic Providers that replace the placeholders with concrete, language-dependent activities to obtain an executable Scale-Out Plan.

```
1  <ServiceTemplate
2   id="HomeAssistant_ServiceTemplate">
3   <Tags>
4    <Tag name="scalingplans"
5     value="scaleout_todevice"
6    />
7    <Tag name="scaleout_todevice"
8     value="Python_3_4,PythonApp_3_4,Topic;
9     PythonHostedOnRaspOS,
10    PythonAppHostedOnRaspOS,
11    PythonAppDependsOnPython,
12    PythonAppConnectsToTopic;
13    UserProvided[Raspbian],
14    UserProvided[TempSensor],
15    WorkloadBased[Mosquitto_3_1],
16    FirstInstance[HomeAssistant]"
17   />
18  </Tags>
```

Fig. 7. Snippet of a TOSCA Service Template of our motivating scenario with its marked regions for scaling out to a Raspberry Pi device.

IV. Validation: A Prototype based on OpenTOSCA

In this section we describe the validation of our approach by a prototypical implementation within the OpenTOSCA Ecosystem and enabling our approach of motivating scenario modeled as a TOSCA Topology Template. We extended our Provisioning Plan Builder Prototype to generate additional Scale-Out Plans in BPEL 2.0 [13] when a TOSCA CSAR Archive is received containing a topology marked with at least one Scale-Out Region. We mapped regions as TOSCA Tags as in Listing 7. A single tag with the key *scalingplans* (see Line 4 in Listing 7) and a comma-separated list of names as value represents the defined set of regions. These names are the keys of other tags defined on the Service Template and used to find the region definitions. We defined a region to scale-out the device configuration by installing a Python App on a Raspberry Pi running Raspbian to send sensor data to a Topic (see Line 5 in Listing 7). Scale-Out Regions are defined by the tag values defined as a comma-seperated list of Node- and Relationship Template IDs (see Lines 8-12 in Listing 7). To define Selection Strategies the region definition also specifies a list of the strategies with the target Node Template as a reference (see Lines 13-16 in Listing 7). The prototype itself is an OSGI-based service integrated into the OpenTOSCA Ecosystem [14] [15] (https://github.com/OpenTOSCA/container). For the implementation of Provisioning Logic Providers that generate BPEL 2.0 activities to provision templates we reused the providers from our previous work. For our motivating scenario we used providers to start Ubuntu virtual machines on an OpenStack Cloud, installation of software, such as Python applications and the Mosquitto Broker, are handled by a provider that uses Management Operations of a Node Type exposing a component life-cycle interface (e.g. “install”, “start”,..), enabling the management of instances. Each of these operations were either implemented as a SOAP Web Service deployed on a Apache Tomcat or a shell script, that are uploaded to a virtual machine or device before hand. Configuration of the Raspberry Pi was achieved by a service that can upload files and execute scripts on Debian-based operating systems, which was also used for virtual machines.

For the Instance Selection Logic Providers we implemented three prototypical implementations of providers. One provider is only able to select the next available instance of any kind of Node Type, this provider implements the “First Instance Selection Strategy” by accessing the OpenTOSCA API at runtime and looks for the needed instance data. A second provider was implemented to handle the “User Provided Selection Strategy” by adding an additional input parameter to the plan that expects a reference to an already instantiated Node Template instance. This allows the users of our scenario to add instances of the needed Node Templates (in this case Raspberry Pi, Raspbian and Temperature Sensor) manually and afterwards execute the generate Scale-Out Plan and reference these instances in the input of the plan. The last ISLP was implemented for the *Workload-based Selection Strategy* by simply checking how many Topics are installed on an instance of the Mosquitto Node Templates and select the one with the lowest count. This ISLP is specific for the Node Type Mosquitto as it must check for Topic Node Template instances. Our Recursive Selection Activities are implemented by using the variables holding the instance data at runtime. These variables are generated in a generic way by the Plan Builder prototype, as all Node- and Relationship Template instance data must be available during the execution of a plan.

V. Related Work

In this section we discuss and compare closely related work on integrating IoT devices into applications.

Carlsson et al. [16] propose a so-called *Configuration Service* that holds pre-defined configurations for different types of devices. Each device will at startup contact the Configuration Service to receive its configuration. After receiving its configuration it will install and configure the needed components itself. One of the main differences to our approach is the predefined configurations for each device type, our approach enable to model a configuration within TOSCA by reusing available Node- and Relationship Types, which in turn enables fast development of configurations. Another main difference is the missing incorporation of the cloud which is able to deliver resources that can store and process data sent by IoT devices.

Perera et al. [17] present an *Context Aware Sensor Configuration Model* which eases the selection and configuration of IoT devices and sensors by using 6 phases. The goal of the first phase is to find available task description that the user is interested in by answering questions asked by the system. The questions are based on the available task descriptions, e.g., if there is a task description about measuring temperature in certain regions, a set of questions that are linked to such a task is asked from the user to determine whether his overall goal is related. After selecting a task, the next phases are used to determine the needed components and sensors based on their inputs and outputs appropriate for the task. Some component may be able to produce views for temperature data and a sensor may be able to produce the needed data. The following phases are optional and are used to (i) inform the user with recommendations for sensor deployment and component acquisition, (ii) discovery of additional context information and (iii) present cost calculations for the selected solution. In overall Perera et al. present an approach that enables guided configuration of IoT applications based on tasks and user interactions. The presented approach is based on user interactions to determine the general task to achieve and according to that determines a set of sensors, their configuration and the binding between gathered data and applications. Main difference to our approach is the focus on supporting users to create applications while ours focuses on integrating IoT components into running applications.

Hirmer et al. [18] introduce a system for automated provisioning, configuration and monitoring of IoT devices. The system is based on the approach on modeling of so-called *Blueprints* of smart environments that specify different devices with their sensors and actuators within the environment. After such a Blueprint is modeled it will be registered in the proposed system and additionally connected to ontologies that describe devices, sensors and actuators. These ontologies, e.g., specify the type, location and usable software adapters to bind the devices to their system. To provision software adapters and configure these, the ontologies are used to determine which adapter from a repository must be used to retrieve and send data to the system. The actual provisioning is achieved by reusing TOSCA Topology Templates that encapsulate the devices with their sensors and actuators, which enables the automated provisioning and configuration of the software adapters needed to bind against the system. After the binding of the devices with the help of adapters their data is send to their systems *Resource Management Platform* that is responsible for enabling IoT applications to receive sensor data and send commands to actuators. Two of the main differences to our approach is the dependency of using a middleware (Resource Management Platform) that abstracts the heterogeneous devices, sensors and actuators. Our approach enables to model applications which use different IoT middlewares and enables incorporating heterogeneous IoT devices by developing adapters as TOSCA artifacts. Additionally, our approach is not bound to the use of ontologies for the selection of devices, and hence the selection of the right software adapters, as a Selection Strategy can implement arbitrary selection methods.

VI. Conclusion

We presented an approach of integrating IoT devices by using our method of generating so-called Scale-Out Plans. We extended TOSCA to enable marking of Scale-Out Regions that can be annotated with Selection Strategies to control the instance selection at runtime from a modeling perspective. From these regions we apply our method to automatically generate Scale-Out Plans, thus, reducing the complexity of installing and configuring software components on IoT devices.

Future works is the incorporation of additional methods, such as migration and scale-up techniques, and the investigation of their applicability to the Internet of Things. Another research direction is to further automate the integration process of IoT devices by enabling a guided installation, e. g., by incorporating Human Tasks [19] into TOSCA Plans.

Acknowledgment

This work was funded by the BMWi project SmartOrchestra (01MD16001F).

References

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

[2] A. Botta, W. De Donato, V. Persico, and A. Pescapé, "Integration of Cloud computing and Internet of Things: A survey," *Future Generation Computer Systems*, vol. 56, pp. 684–700, 2016.

[3] Home Assistant, "Home Assistant Official Site," Jul. 2017. [Online]. Available: https://home-assistant.io

[4] Raspberry Pi Foundation, "Raspberry Pi Official Site," Jul. 2017. [Online]. Available: https://www.raspberrypi.org

[5] Eclipse Foundation, "Mosquitto Official Site," Jul. 2017. [Online]. Available: https://mosquitto.org

[6] OASIS, *Message Queuing Telemetry Transport (MQTT) Version 3.1.1*, Organization for the Advancement of Structured Information Standards (OASIS), Oct. 2014.

[7] ——, *Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0*, Organization for the Advancement of Structured Information Standards (OASIS), 2013.

[8] ——, *Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0*, Organization for the Advancement of Structured Information Standards (OASIS), 2013.

[9] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014.

[10] G. Galante and L. C. E. de Bona, "A Survey on Cloud Computing Elasticity," in *Utility and Cloud Computing (UCC), 2012 IEEE 5th International Conference on.* IEEE, 2012, pp. 263–270.

[11] L. Sun, H. Dong, F. K. Hussain, O. K. Hussain, and E. Chang, "Cloud service selection: State-of-the-art and future research directions," *Journal of Network and Computer Applications*, vol. 45, pp. 134–150, 2014.

[12] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, and J. Wettinger, "Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA," in *International Conference on Cloud Engineering (IC2E 2014).* IEEE, Mar. 2014, pp. 87–96.

[13] OASIS, *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*, Organization for the Advancement of Structured Information Standards (OASIS), 2007.

[14] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "OpenTOSCA - A Runtime for TOSCA-based Cloud Applications," in *Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013).* Springer, Dec. 2013, pp. 692–695.

[15] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, "Winery – A Modeling Tool for TOSCA-based Cloud Applications," in *Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013).* Springer, Dec. 2013, pp. 700–704.

[16] O. Carlsson, P. P. Pereira, J. Eliasson, J. Delsing, B. Ahmad, R. Harrison, and O. Jansson, "Configuration Service in Cloud based Automation Systems," in *Industrial Electronics Society, IECON 2016-42nd Annual Conference of the IEEE.* IEEE, 2016, pp. 5238–5245.

[17] C. Perera, A. Zaslavsky, M. Compton, P. Christen, and D. Georgakopoulos, "Semantic-driven Configuration of Internet of Things Middleware," in *Semantics, Knowledge and Grids (SKG), 2013 9th International Conference on.* IEEE, 2013, pp. 66–73.

[18] P. Hirmer, U. Breitenbücher, A. C. F. da Silva, K. Képes, B. Mitschang, and M. Wieland, "Automating the Provisioning and Configuration of Devices in the Internet of Things," *Complex Systems Informatics and Modeling Quarterly*, no. 9, pp. 28–43, 2016.

[19] OASIS, *Web Services - Human Task (WS-HumanTask) Version 1.1*, Organization for the Advancement of Structured Information Standards (OASIS), Aug.

All links were last accessed on September, 30th 2017.