Policy-Aware Provisioning Plan Generation for TOSCA-based Applications

Kálmán Képes, Uwe Breitenbücher, Markus Philipp Fischer,
Frank Leymann, Michael Zimmermann

Institute of Architecture of Application Systems, University of Stuttgart, Germany,
lastname@iaas.uni-stuttgart.de

BibTeX:

```
@inproceedings{Kepes2017,
  author    = {K{\’a}lm{\’a}n K{\’e}pes and Uwe Breitenb{\”u}cher and Markus
               Philipp Fischer and Frank Leymann and Michael Zimmermann},
  title     = {Policy-Aware Provisioning Plan Generation for TOSCA-based
               Applications},
  booktitle = {Proceedings of The Eleventh International Conference on Emerging
               Security Information, Systems and Technologies (SECURWARE 2017),
               10-14 September 2017, Rome, Italy},
  year      = {2017},
  month     = {September},
  pages     = {142-149},
  isbn      = {978-1-61208-582-1},
  publisher = {Xpert Publishing Services}
}
```

These publication and contributions were presented at SECURWARE 2017
SECURWARE 2017 Web site:
https://www.iaria.org/conferences2017/SECURWARE17.html

Policy-Aware Provisioning Plan Generation for TOSCA-based Applications

Kálmán Képes, Uwe Breitenbücher, Markus Philipp Fischer,
Frank Leymann, and Michael Zimmermann

Institute of Architecture of Application Systems, University of Stuttgart,
70569 Stuttgart, Germany
Email: {kepes, breitenbuecher, fischer, leymann, zimmermann}@iaas.uni-stuttgart.de

Abstract—**A major challenge in enterprises today is the steadily increasing use of information technology and the required higher effort in terms of development, deployment, and operation of applications. Especially when different application deployment technologies are used, it becomes difficult to comply to non-functional security requirements. Business applications often have to fulfill a number of non-functional security requirements resulting in a complex issue if the technical expertise is insufficient. Therefore, the initial provisioning of applications can become challenging when non-functional requirements have to be fulfilled that arise from different domains and a heterogeneous IT landscape. In this paper, we present an approach and extend an existing deployment technology to consider the issue of security requirements during the provisioning of applications. The approach enables the specification of non-functional requirements for the automated deployment of applications in the cloud without the need for specific technical insight. We introduce a Policy-Aware Plan Generator for Policy-Aware Provisioning Plans that enables the implementation of reusable policy-aware deployment logic within a plug-in system that is not specific to a single application. The approach is based on the Topology and Orchestration Specification for Cloud Applications (TOSCA), a standard that allows the description of composite Cloud applications and their deployment. We prove the technical feasibility of our approach by extending our prototype of our previous work.**

Keywords–Cloud Computing; Application Provisioning; Security; Policies; Automation.

I. Introduction

A major challenge in enterprises today is the steadily increasing use of information technology (IT) due to the required higher effort in terms of development, deployment, and operation. Each new technology introduced to an enterprise's IT landscape also increases the complexity while the largest fraction of failures is beeing caused by manual operator errors [1]. These concerns have been addressed through outsourcing of IT to external providers, as well as through management automation of IT. Both of these aspects are enabled by Cloud Computing [2]. Due to a significant reduction of required technical knowledge, cloud services provide easy access to properties, such as elasticity and scalability [3].

Each IT solution has its functional and non-functional requirements that need to be addressed when using cloud services. Unfortunately, functional possibilities often outweigh the non-functional security issues that also have a need to be dealt with. Most cloud services are easy to use, but it is often difficult for users to extend and configure the cloud services to their particular needs, especially when non-functional aspects, such as security, have to be considered. Moreover, modern applications are often made up of complex and heterogeneous components that are hosted on cloud services or interact with them. Especially when different deployment technologies are used, it becomes difficult to comply to security requirements [4][5]. Such applications often have to fulfill a number of non-functional security requirements [6][7], which results in a complex provisioning challenge if the technical expertise is insufficient. Therefore, the initial automated provisioning of applications can become challenging when non-functional requirements have to be fulfilled that arise from many different domains and a heterogeneous IT landscape [8].

In this paper, we present a concept and extend an existing deployment technology to consider the issue of security requirements during the automated provisioning of applications. We present a *Policy-Aware Plan Generator* that enables generating executable *Policy-Aware Provisioning Plans* that respect policies that have to be fulfilled during the provisioning of an application. Our approach enables the implementation of reusable policy-aware deployment logic based on a plug-in system that is not specific to a single application. The approach is based on the *Topology and Orchestration Specification for Cloud Applications (TOSCA)*, a standard allowing the description of composite Cloud applications and their orchestration [9]. The extension to the existing technology [10] enables the fully automated deployment of Cloud applications while complying with security requirements defined as *Provisioning Policies*. Our approach enables the specification of non-functional requirements for the deployment of applications in the cloud without the need for specific technical insight other approaches require. Additionally, security experts of different domains are enabled to work collaboratively on a single model for applications. We validate our approach by a prototypical implementation based on the *OpenTOSCA Ecosystem* [11][12].

The remainder of this paper is structured as follows. In Section II, we explain the fundamental concepts of the TOSCA standard, which is used within our approach as a cloud application modeling language. Afterwards, we motivate our approach with a motivating scenario and introduce several exemplary Provisioning Policies in Section III. In Section IV, we describe our approach for generating executable Policy-Aware Provisioning Plans based on TOSCA. Section V presents a validation of the approach in the form of a prototypical implementation based on the OpenTOSCA Ecosystem. Section VI gives an overview of related work. Finally, we conclude this paper and give an outlook on future work in Section VII.

II. Topology and Orchestration Specification for Cloud Applications (TOSCA)

In this section, we introduce the TOSCA standard on which our approach and prototype are based. TOSCA enables to describe the automated deployment and management of applications in an interoperable and portable manner. In order to give a compact introduction to the OASIS standard, we only describe the fundamental concepts of TOSCA required to understand our presented approach. More details can be found in the TOSCA Specifications [9][13], the TOSCA Primer [14] and a more detailed overview is given by Binz et al. [15].

The structure of a TOSCA-modeled application is defined by a *Topology Template*, which is a multi-graph consisting of nodes and directed edges. The nodes within the Topology Template represent so called *Node Templates*. A Node Template represents software or infrastructure components of the modeled application, such as a hypervisor, a virtual machine, or an Apache HTTP Server. The edges connecting the nodes represent so called *Relationship Templates*, which specify the relations between Node Templates. Thus, the Relationship Templates are specifying the structure of a Topology Template. Examples for such relations are "hostedOn", "dependsOn", or "connectsTo". The semantics of the Node Templates and Relationship Templates are specified by *Node Types* and *Relationship Types*. These types are reusable classes that allow to define *Properties*, as well as *Management Operations* of a type of component or relationship. An "Apache HTTP Server" Node Type, for example, may specify Properties for the port number to be accessible and additionally define required credentials, such as username and password. The defined Management Operations of a Node Type are bundled in interfaces and enable the management of the component. For example, the "Apache HTTP Server" Node Type may define an operation "install" for installing the component itself and a "deployApplication" operation to deploy an application on the web server. A cloud provider or hypervisor Node Type typically defines Management Operations, such as "createVM" and "terminateVM" for creating and terminating virtual machines.

These Management Operations are implemented by so called *Implementation Artifacts (IAs)*. An Implementation Artifact itself can be implemented using various technologies. For instance, an Implementation Artifact can be a WAR-file providing a WSDL-based SOAP Web Service, a configuration management artifact executed by a tool, such as Ansible [16] or Chef [17], or just a simple shell script. Depending on the Implementation Artifact, they are processed in different ways: (i) IAs, such as shell scripts, are transfered to the application's target environment and executed there. (ii) IAs, such as WAR-files implementing a Web Service, are deployed and executed in the so called *TOSCA Runtime Environment* (See last paragraph). This kind of Implementation Artifact typically performs operations by using remote access to the components. (iii) Implementation Artifacts that are already running are just referred within the model, such as a hypervisor or cloud provider service and then are called directly with the help of adapters implemented within.

Besides Implementation Artifacts, TOSCA defines so called *Deployment Artifacts (DAs)*. Deployment Artifacts implement the business functionality of a Node Template. For example, a Deployment Artifact can be a WAR-file providing a Java application. Another example would be a PHP application where a ZIP file containing all the PHP files, images, and other required files implementing the application would be represented by the Deployment Artifact.

The automatic creation and termination of instances of a modeled Topology Template, as well as the automated management of the application is enabled by so-called *Management Plans*. A Management Plan specifies all tasks and their order for fulfilling a specific management functionality, such as provisioning a new instance of the application or to scale out a component of the application. A Management Plan that provisions a new instance of the application is called a *Provisioning Plan* in this paper. Management Plans invoke the Management Operations which are specified by the Node Types and implemented by the corresponding Implementation Artifacts of the topology. TOSCA does not specify how Management Plans should be implemented. However, the use of established workflow languages, such as the *Business Process Execution Language (BPEL)* [18] or the *Business Process Model and Notation (BPMN)* [19], is encouraged.

TOSCA also allows the specification of policies for expressing non-functional requirements. For example, a policy can define the security requirements of an application, e.g., that a component of the application must be protected from public access. Again, for reusability purposes, TOSCA allows the definition of *Policy Types*. A Policy Type, for example, defines the properties that have to be specified for a policy. However, the actual values of these properties are specified within *Policy Templates* attached to Node Templates for which the policy has to be fulfilled. A Policy Type can be also associated with a Node Type in order to describe the policies this component provides. Since TOSCA does not make any statement about policy languages, any language can be used to define them. We call policies that have to be fulfilled during the provisioning of the application *Provisioning Policies*.

In order to package Topology Templates, type definitions, Management Plans, Implementation Artifacts and Deployment Artifacts, as well as all required files for automating the provisioning and management of applications, the TOSCA Specification defines the so called *Cloud Service Archive (CSAR)*. A CSAR is a self-contained and portable packaging format for exchanging TOSCA-based applications.

Through the standardized meta-model and packaging format, CSARs can be processed and executed by any standard-compliant *TOSCA Runtime Environment*, thus, ensuring portability, as well as interoperability. However, since there are two approaches for provisioning an instance of a TOSCA-modeled application, there a two kinds of TOSCA Runtime Environments: (i) TOSCA Runtime Environments that support *declarative provisioning* and (ii) TOSCA Runtime Environments that allow *imperative provisioning* [10]. In declarative processing, the TOSCA Runtime Environment interprets the Topology Template to infer which Management Operations need to be executed in which order to provision the application, without the need for a Provisioning Plan. In imperative provisioning on the other side, the TOSCA Runtime Environment requires a Provisioning Plan provided by the CSAR to instantiate the application by invoking this plan. In this paper, we present a hybrid policy-aware deployment approach that interprets the declarative Topology Template and generates an imperative executable Policy-Aware Provisioning Plan.

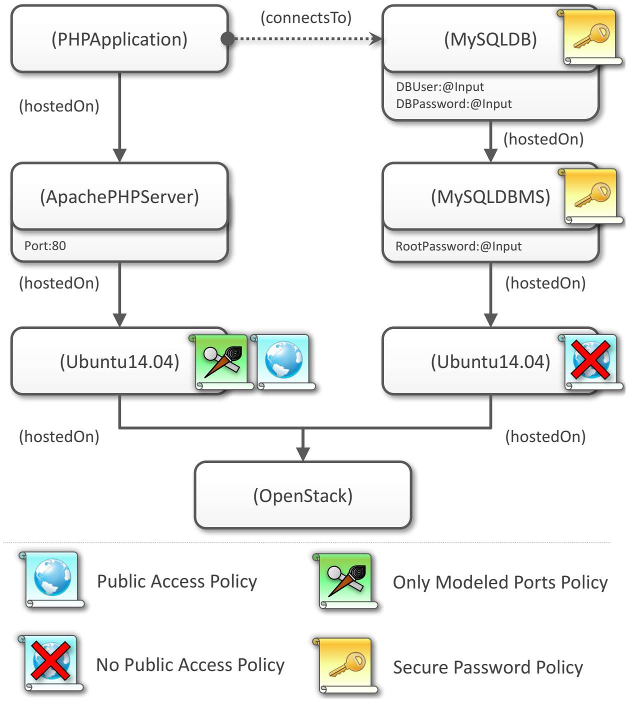

Figure 1. Our Motivating Scenario, where the backend needs to high security (right side), whereas the frontend needs to be publicly accessible.

III. Motivating Scenario

In this section, we describe a TOSCA-based motivating scenario that we will use throughout the paper to describe our approach. Our scenario is depicted in Figure 1 as a TOSCA Topology Template specifying a typical application to serve a website that is connected to a database system. The shown topology consists of a PHP web application with a MySQL database, additionally a set of Provisioning Policies are specified in the form of Policy Templates that must be enforced during provisioning. Components within the Topology Template are defined as TOSCA Node Templates (e.g., OpenStack, Ubuntu, Apache Web Server, PHP application, MySQL DBMS, and MySQL Database). These are connected through TOSCA Relationship Templates of the types “hostedOn” and “connectsTo” to either define that a component will be hosted on another component (e.g., MySQLDBMS is hosted on an Ubuntu 14.04 virtual machine) or to specify that a component is connected to another component (e.g., PHP application connects to its MySQL database by using the given password from the input of the DBPassword property). To instantiate an Ubuntu 14.04 virtual machine, the OpenStack Node Template exposes Management Operations, such as *createVM* which takes as parameters the specification of the virtual machine, e.g., RAM, CPUs, etc. Customizing the modeled application is restricted to setting credentials for the MySQL Database and its MySQL Database Management System at provisioning time. This is achieved by setting the value of the MySQL DB and MySQL DBMS Node Templates’ Properties “DBUser”, “DBPassword” and “RootPassword” to “@input”. To model the desired non-functional security requirements, *Provisioning Policies* are attached to Node Templates of the Topology Template. In the following subsections, we describe the Provisioning Policies of our scenario in detail.

A. Public Access Policy

With the *Public Access Policy* a modeler specifies that the deployment system must ensure that the associated component is available and accessible from outside the cloud environment, hence open for the public internet. In our scenario, the website owner wants to make sure that the Ubuntu 14.04 virtual machine on the OpenStack cloud is accessible by the public. Therefore, the Public Access Policy is attached to the Ubuntu 14.04 virtual machine Node Template of the front-end Ubuntu virtual machine (Left side in Figure 1).

B. No Public Access Policy

While the Public Access Policy enforces accessibility from outside the cloud environment, the *No Public Access Policy* has the opposite goal. Its main purpose is to restrict access to the associated component by allowing to serve requests solely from within the cloud. For our scenario, the owner wants to be sure that his virtual machine that hosts sensitive data within the database is not directly accessible from the internet. Thus, he attaches the No Public Access Policy to the Ubuntu 14.04 virtual machine of the MySQL database management system to enforce restricted access (Right side in Figure 1).

C. Only Modeled Ports Policy

The intent of the *Only Modeled Ports Policy* is to restrict access to the associated component to the modeled ports. This allows the application owner to further secure his front-end, e.g., the Ubuntu 14.04 virtual machine hosting the PHP Application shall allow access to only explicitly modeled ports. To do so, the Only Modeled Ports Policy is attached on the Ubuntu 14.04 Node Template restricting access only to port 80, as the only installed component specifying a port within the topology is the Apache Server (Left side in Figure 1) that hosts the front-end PHP application.

D. Secure Password Policy

As the final policy within our scenario, the owner uses the *Secure Password Policy* that enforces the use of strong passwords for components. This increases the barrier for attackers of the application by preventing the usage of weak passwords at provisioning and runtime, e.g., when the PHP application component is connected to MySQL database. Thus, the application owner attaches Secure Password Policies on both, the MySQLDB and MySQLDBMS Node Templates (See right side at the top in Figure 1) running on the backend Ubuntu virtual machine of the OpenStack cloud. As the passwords in the scenario are set at runtime (indicated by the Property value “@input”) the system must ensure at runtime that the given data is compliant with the set policy.

With the attached Provisioning Policies Public Access and Only Modeled Ports the owner of our scenario is able to ensure that the front-end is available to the public and restricts access to only intended ports of the application running on the modeled Ubuntu virtual machine. To secure his backend, he is able to use the No Public Access and Secure Password Policy to restrict access from outside and to enforce the usage of strong passwords for the database running on the back-end Ubuntu virtual machine. In the following section, we will show how our deployment approach provisions this application while strictly enforcing the specified Provisioning Policies.

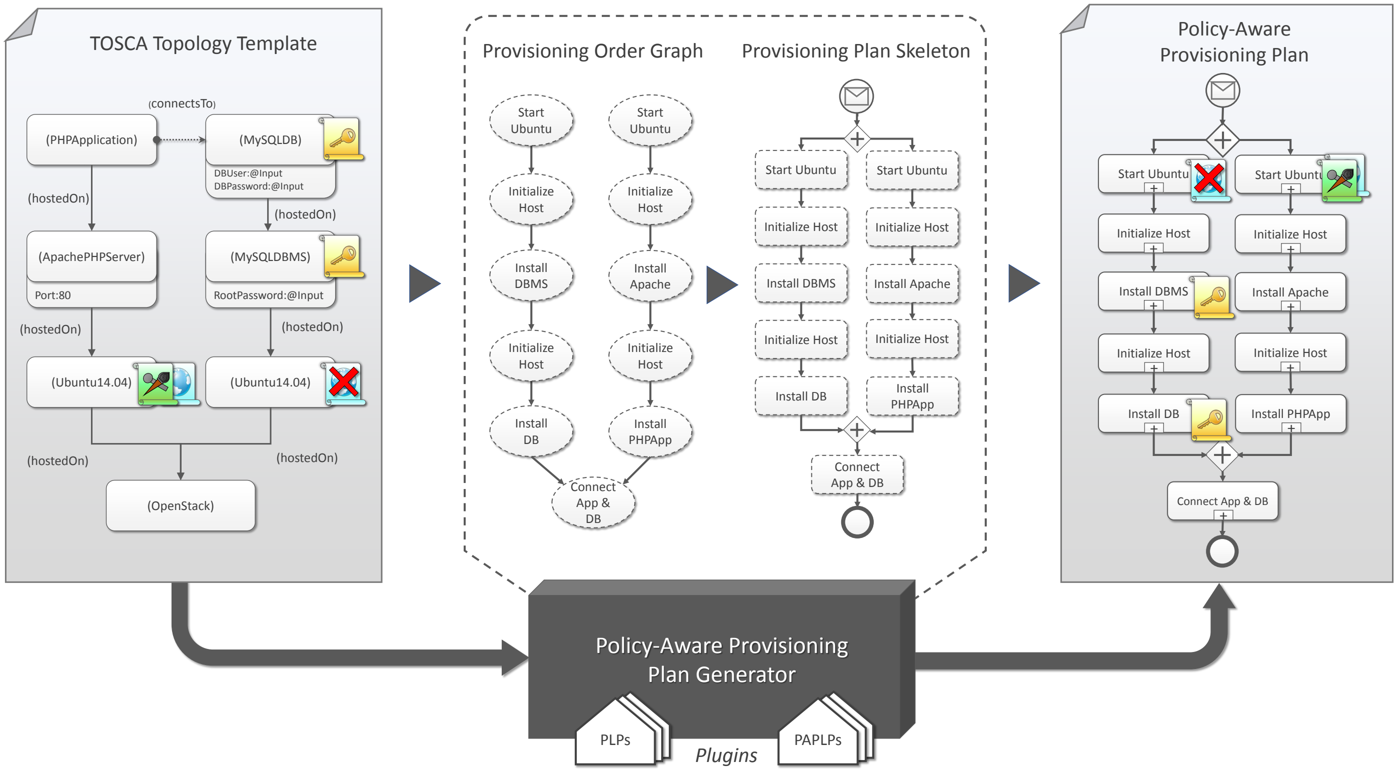

Figure 2. Overview of our approach for transforming a TOSCA Topology Template into a Policy-Aware Provisioning Plan.

IV. Policy-Aware Provisioning Plan Generation

In this section, we present our approach of provisioning applications while enforcing specified non-functional security requirements that are specified as policies attached to the deployment model. This section is structured as follows: In the first subsection, we detail out a *Role Model* for our approach. In the second subsection, we give an overview of our approach that entails transforming a TOSCA Topology Template into an executable Provisioning Plan that is able to provision the application while enforcing all specified Provisioning Policies. Afterwards, we describe all transformation phases in detail.

A. Role Model

Creating a TOSCA CSAR is taken care of by a *Cloud Service Creator* [9] by developing the Topology Template, the associated types, artifacts, and the policies to enforce. TOSCA enables reusing type definitions, so common Node-, Relationship- and Artifact Types can be reused without the need to create them from scratch. The final CSAR contains only the TOSCA Topology Template without a Provisioning Plan. This CSAR is then given to our *Policy-Aware Provisioning Plan Generator* that generates a *Policy-Aware Provisioning Plan* that also enforces the specified policies of the Topology Template during provisioning. Afterwards, the creator can inspect the generated Policy-Aware Provisioning Plan to customize the provisioning. Please note: the generated plan correctly enforces the policies, but additional tasks that cannot be modeled declaratively may be added to customize the execution if required. For example, a task for sending an e-Mail to the administrator can be added. However, the adaptation of the generated plan may also be forbidden in order to ensure that no policy-related tasks get influenced negatively. As the final step, the CSAR with the generated plan is sent to a *TOSCA-enabled Cloud Provider* hosting a *TOSCA Runtime Environment*, which is able to process and execute the generated Provisioning Plan contained in the CSAR.

B. Overview of the Plan Generation Phases

In this section, we describe an overview of how we generate a Policy-Aware Provisioning Plan for TOSCA Topology Templates. In the first phase, (i) a *Provisioning Order Graph (POG)* is generated. The POG only specifies the provisioning order of the Node and Relationship Templates for the given Topology Template. This graph is then (ii) translated into a *Provisioning Plan Skeleton (PPS)* in a particular plan language, for example, BPEL or BPMN. The PPS contains placeholder activities for the provisioning of all Node and Relationship Templates. These placeholder activities contain no provisioning logic and are following the provisioning order specified by the POG. Thus, each node and relation in the POG is translated into such a placeholder activity, the edges of the POG are translated into control flow constructs of the respective plan language. These two phases are the same as in our previous work [10], for which the goal was to generate Provisioning Plans. To enable policy-aware provisioning, we extend the last phase of our previous work in this paper as follows: In the last phase, the PPS is (iii) completed to an executable *Policy-Aware Provisioning Plan* where the placeholder activities of the skeleton are replaced by concrete Management Operation calls, which provisions the Node- and Relationship Templates while ensuring that all attached Policy Templates are fulfilled during execution. Figure 2 depicts this approach and in the following subsections, we will describe each phase in detail.

C. Provisioning Order Graph Generation Phase

In the first phase, we generate a *Provisioning Order Graph (POG)* that describes only the order in which the Node- and Relationship Templates have to be provisioned (see second graph from the left in Figure 2). In our scenario, we need to generate a POG that represents the deployment order of the front-end (i.e., Ubuntu 14.04 virtual machine, Apache PHP Server, PHP Application), the back-end (i.e., another Ubuntu 14.04 virtual machine, MySQL Database Management System, MySQL Database), and the connection of the two stacks (i.e., initializing the "connectsTo" Relationship Template). The POG consists of nodes for each Node and Relationship Template that represents the step of provisioning the respective component or relation. Edges between the nodes of the POG specify the order of the provisioning steps, i.e., the provisioning order of the Node Templates and Relationship Templates.

The calculation of the ordering is based on the semantics of the Relationship Templates' types: while a "hostedOn" Relationship Type expects that the target Node Template is provisioned before the source, the "connectsTo" Relationship Type forces the order of provisioning that both the source and target Node Templates must be provisioned before the connection can be initialized. The Relationship Types"hostedOn" and "connectsTo" are abstract types, i.e., concrete types derived from these abstract types inherit the semantic behavior. For our scenario, the POG would contain a series of nodes that represent the provisioning of the front- and back-end in parallel. For each Node Template that is the source of a "hostedOn" Relationship Template the abstract POG contains a node for the Node- and Relationship Template each. For example, in Figure 2 there is a node in the POG for each Node- and Relationship Template that must be started (e.g., virtual machines) or installed (e.g., Apache and MySQL DBSMS). After the provisioning of the front- and back-end stacks, the last Relationship Template of the type "connectsTo" is provisioned: To configure the PHP Application with the needed credentials for the database stack, an operation "connectTo" is invoked on PHP Application Node Template with the database credentials properties as parameters. More details on generating a Provisioning Order Graph can be read in [10]. Please note that the calculation of the provisioning order is independent of the specified Provisioning Policies.

D. Provisioning Plan Skeleton Transformation Phase

The second phase transforms the POG into a plan language-dependent *Provisioning Plan Skeleton (PPS)* that follows the provisioning order of the Node and Relationship Templates specified by the POG (third graph in Figure 2 from the left). These PPSs are implemented in a certain process modelling language, such as BPEL [18], BPMN [19], or BPMN4TOSCA [20][21]; but only entail empty *placeholders* for deployment activities and the general provisioning order. Therefore, each node in the POG is transformed into one placeholder while the provisioning order of the POG is translated into language-specific control flow constructs between these placeholders. Thus, the skeleton is not executable yet as the placeholders contain no provisioning logic. The Policy-Aware Plan Generator has different plug-ins for generating skeletons in different plan languages. For example, in BPEL these placeholders can be realized as empty `<scope>` activities.

```
procedure: CompletePPS(Topology Template $t$, Provisioning
    Plan Skeleton $s$)
 1: for ($\forall$ Node- and Relationship Templates $temp \in t$) do
 2:   if ($temp$ has attached Policy Set $p \wedge p \neq \emptyset$) then
 3:     for ($\forall$ Policy-Aware PLPs $papl$) do
 4:       if ($papl$ can handle $temp$ enforcing $p$) then
 5:         Replace Placeholder in $s$ of $temp$ with Exe-
            cutable Policy-Aware Provisioning Logic from
            $papl$
 6:       else if (No Policy-Aware PLP $papl$ can handle
          $temp$ with $p$) then
 7:         Abort completion of $s$
 8:       end if
 9:     end for
10:   else
11:     for ($\forall$ PLPs $plp$) do
12:       if ($plp$ can handle $temp$) then
13:         Replace Placeholder in $s$ of $temp$ with Exe-
            cutable Provisioning Logic from $plp$
14:       else if (No PLP $plp$ can handle $temp$) then
15:         Abort completion of $s$
16:       end if
17:     end for
18:   end if
19: end for
```

Figure 3. Pseudocode for completing a Provisioning Plan Skeleton into an executable Policy-Aware Provisioning Plan.

E. Policy-Aware Provisioning Plan Completion Phase

In the third phase, the generated Provisioning Plan Skeleton is completed by adding the technical deployment activities into the placeholders. The Plan Generator provides a plugin system for *Provisioning Logic Providers (PLPs)*, which are responsible for filling the placeholders by executable technical provisioning logic. Each PLP is capable of providing the technical activities for the provisioning of one Node Type or Relationship Type to complete the skeleton into an executable form (See the graph on the right in Figure 2). We extend this original completion phase [10] by *Policy-Aware Provisioning Logic Providers (PAPLPs)*, which are capable of providing provisioning logic similarly to PLPs but additionally ensure that all policies attached to the Node or Relationship Template they support are enforced by the injected technical provisioning logic. An overview of this extended phase is outlined in Algorithm 3, which we now describe in detail.

As the input of the policy-aware completion phase the Topology Template t and its Provisioning Plan Skeleton (PPS) s is given (see Input before line 1 in Algorithm 3). The completion phase will begin to cycle through all Node- and Relationship Templates $temp$ (line 1) of the topology t, and checks whether $temp$ has a non-empty set of Provisioning Policies p attached (line 2). If this is the case, the algorithm starts to cycle through all available PAPLPs $papl$ (line 3) and checks whether there is one $papl$ that can provide executable activities to provision $temp$ while enforcing all attached Provisioning Policies p (line 4). A PAPLP is allowed to inspect the whole Topology Template t to decide whether it can provide provisioning logic for the given Node- or Relationship Template, respectively, while fulfilling the attached policies. For example, by traversing the topology t from a Node

Template $temp$ to find all port properties specified, which have to be set for enforcing the Only Modeled Ports Policy. If a $papl$ is found that is able to add technical activities for $temp$ ensuring that the attached policies p are enforced, the generator requests this $papl$ to inject the necessary activities to provision and enforce the policies into the corresponding placeholder in the PPS s (line 5). The injected activities typically invoke the Management Operations provided by the respective Node or Relationship Template and range from uploading files to executing scripts, etc. When there is no suitable $papl$ that can provide provisioning logic for a certain Node or Relationship Templates $temp$ while fulfilling all attached policies, (line 6), the system will abort the completion as it is not possible to provisioning the given topology t while enforcing all specified policies p (line 7).

While cycling through the Node and Relationship Templates and the algorithm detects that a $temp$ has no attached Policy Set p, it will start to cycle through the set of normal (non policy-aware) PLPs plp (line 11). Within the cycle it checks whether $temp$ can be provisioned by one of the PLPs plp (line 12), and if one plp is found it will be requested to replace the respective placeholder of $temp$ in s (line 13). When no plug-in plp is found (line 14), the algorithm aborts (line 15).

This algorithm forces the completion step to produce either Policy-Aware Provisioning Plans where policy enforcement is guaranteed or the generation will be aborted. Please note: We strictly separate the handling of Node and Relationship Templates that have attached policies from the ones that specify no policies. Thus, the algorithm does not try to find a PAPLP for a template that specifies no policies. The reason for this is that the injected provisioning logic should reflect exactly the deployment model and should not be more restrictive as required in terms of adding unrequested security stuff. However, the algorithm could be easily modified in line 11 to also check if there is a PAPLP $papl$ capable of providing (possibly unnecessarily secured) provisioning logic for a certain Node or Relationship Templates that does not specify any Provisioning Policy to be enforced.

After the step of completing the Provisioning Plan Skeleton into an executable Policy-Aware Provisioning Plan, all placeholder activities resembling the deployment graph of the original POG are replaced by language-dependent sets of executable provisioning activities whose execution provisions the respective Node- and Relationship Template while enforcing the attached Policy Templates. Finally, the Cloud Service Creator is able to review the generated plans and if needed, and may alter the technical activities to his needs. However, after modification of the generated activities, it cannot be guaranteed that the altered Policy-Aware Provisioning Plan correctly enforces the specified Provisioning Policies. Therefore, a manual adaptation is possible but should be considered carefully.

The presented algorithm is completely independent of any policy language used to specify Provisioning Policies: A PAPLP by itself decides if (i) it understands all attached policies of a Node or Relationship Template, respectively, and (ii) if it is able to provide appropriate provisioning logic. Thus, the presented algorithm is extensible to any policy language, by developing PAPLPs that are able to process elements of such a language and generate appropriate activities that enforce the specified policy.

V. Validation

In this section, we describe a prototypical implementation of our approach to validate its practical feasibility. In the following two subsections, we describe how the Cloud Service Creator (See Subsection IV-A) can model the motivating scenario within the OpenTOSCA Ecosystem [11] using the TOSCA modeling tool *Winery* [12] and how the create can generate a Policy-Aware Provisioning Plan that provision the application while fulfilling all Provisioning Policies. Moreover, we explain how the *OpenTOSCA Container* [11] is able to execute these plans automatically to provision the application.

Modeling TOSCA Topology Templates and the generation of the Policy-Aware Management Plans is done solely within the Winery modeling tool. Winery is a Java-based web application that can be deployed on servers, such as Apache Tomcat (http://tomcat.apache.org). Users can define all their types, such as Node, Relationship, Artifact, and Policy Types inside the *Entity Modeler* and use them inside the *Topology Modeler* to graphically model the wanted Topology Template for their Service Template. By creating Node Templates from Node Types and connecting them by Relationship Templates of a certain Relationship Type, the user is able to specify a topology of the application. Afterwards, the modeler is able to configure the Topology Template by setting appropriate Properties on the Node- and Relationship Templates. To finally attach the required Provisioning Policies, Winery supports specifying Policy Templates from defined Policy Types. After modeling, the user can request the creation of a Policy-Aware Provisioning Plan for the Topology Template. Winery will then invoke the Policy-Aware Plan Generator to generate a BPEL Provisioning Plan, which is then packaged into the CSAR.

For the generation of Policy-Aware Provisioning Plans, we implemented our approach by extending the Plan Generator prototype that already is able to generate BPEL Provisioning Plans executable within the OpenTOSCA Ecosystems' TOSCA Runtime Environment *OpenTOSCA* (https://github.com/OpenTOSCA/container). The original implementation itself is written in Java and utilizes the OSGI-Framework as a plug-in system that enables adding additional provisioning logic as Provisioning Logic Providers (PLPs). For implementing our approach, we extended the plug-in system to allow the usage of Policy-Aware Provisioning Logic Providers (PAPLPs) and specified an according plugin interface. After creating the BPEL Provisioning Plan Skeleton, the additional policy plug-in layer is invoked to add its logic by processing attached Policy Templates while cycling through the Node- and Relationship Templates with the set of available PAPLPs as described in Section 2. Each of the PAPLPs is able to verify whether it can create activities that can enforce the given Policy Templates while provisioning the Node- or Relationship Templates. The implementation of the PAPLP plug-ins resulted in extended versions of the original PLP plug-ins [10].

The implementation for enforcing the Secure Password Policy resulted in a PAPLP plug-in for the MySQL Database and MySQL Database Management System Node Types. Each checks either the input of the BPEL plan at runtime or the specified passwords in the Node Template Properties, and stops execution of the plan when the given password data was not strong enough based on commonly accepted criteria. To enforce the No Public Access, Public Access, and Only Modeled Ports Provisioning Policies, we extended the PLP plug-in that already was able to provision an Ubuntu 14.04 virtual machine on an OpenStack cloud in two ways: The (i) first extension was made for the (No) Public Access Policy, which was implemented in a PAPLP plug-in that is additionally able to add activities to the BPEL Provisioning Plan Skeleton that configures the security group of a virtual machine to be either publicly available or not. The extension for the Only Modeled Ports Policy resulted in a (ii) second extension of the plug-in that enables it to additionally add activities that set a Unix Cron job to regularly re-set the ports modeled inside the application. The plug-in determines based on the hostedOn relations of the Topology Template which component is installed on the Ubuntu 14.04 Node Template attached with an attached Only Modeler Ports Policy Template, and while doing so, fetches the set ports or reads them at runtime and configures the activities in the BPEL Provisioning Plan Skeleton to set the Cron job on the Ubuntu 14.04 virtual machine after provisioning.

In summary, we implemented our scenario within the OpenTOSCA Ecosystem which already was extended by us to generate BPEL Provisioning Plans to be executed in the TOSCA Runtime Environment OpenTOSCA Container. In this paper, we further extended the prototype of the Plan Generator component to enable policy-aware provisioning by allowing to register Policy-Aware Provisioning Plugins that are able to generate provisioning activities for Node Templates of specific Node Types while enforcing the specified Policy Templates.

VI. Related Work

In this section, we present related work, which range from Management and Deployment Frameworks to Workflow and configuration management technologies that focus on enforcing policies at provisioning time.

Walraven et al. [22] present *PaaSHopper*, a Policy-Driven middleware for multi-PaaS environments. The main components of the approach enabling policy-awareness are the *Dispatcher* and the *Policy Engine*. While the Policy Engine retrieves data about the PaaS execution environment to monitor whether or not a policy is enforced, the Dispatcher uses the Policy Engine to decide based on the current context of the policies to which component a request is dispatched and, additionally, handles the deployment of components. To adapt the applications on changing policies at runtime, the PaaSHopper middleware is able to change deployment descriptors of the application components. The main difference to our approach is the ability to model policy-aware applications not only restricted to PaaS solutions and the ability to audit the generated plans whether the polices are enforced correctly at provisioning time as we explicity generate an executable Policy-Aware Provisioning Plan model.

The contributions of Ouyang et al. [23] integrate policies into workflows by using a *Policy Server* and a *Policy Repository*. The Policy Server acts as a service bus between the components and the workflow at runtime, similar to [22], but with the exception that the evaluation of actions to be taken is done in so-called *Decision Point Activities* of the workflow at runtime by interacting with the Policy Server. The difference to our approach is the use of a Policy Server, while our approach is based on provisioning logic injected by specific plugins to automatically generate a Policy-Aware Provisioning Plan.

Blehm et al. [24] present an approach to define policies on TOSCA Topology Templates similar to ours. The main difference between the two approaches is within the initial configuration and enforcement of the policies. While both phases of initial configuration and enforcement in the approach of Blehm et al. rely on special services packaged with the Policy Type definitions, our approach utilizes the Management Operations of the Node Templates itself and the policy-aware provisioning logic is injected by external plugins. An additional difference is that the approach of Blehm et al. requires manually developing the Provisioning Plan, while our approach supports automatically generating this plan.

In Keller et al. [25] the *CHAMPS* system to enable Change Management of IT systems and resources is presented. Similiarly to our approach Keller et al. generate so-called *Task Graphs* that specifies the abstract steps that have to be taken to serve a so-called *Request for Change* for the used IT systems and resources. These Task Graphs are then transformed into an executable plan as within our approach. CHAMPS also enables the specification of policies and SLAs, although the work gives no detail on how these are processed by their system.

Mietzner et al. [26] present the standards-based enterprise bus *ProBus* that is able to optimize resource and service selection based on policies. Clients are able to send invocation request with attached policies to ProBus, which then must be enforced by the service providers. Similar to our approach is the usage of processes to orchestrate provisioning services while enforcing the set policies, however, these processes are developed manually, a complex and error-prone task, and not generated as within our approach.

Jamkhedkar et al. [27] present a *Security on Demand* architecture which allows to provision and migrate virtual machines (VMs) with different security requirement levels for the servers they are running on. A user is able to request the provisioning of a VM along with a security policy that is processed by the so-called *Policy Validation Module*. The Policy Validation Module is connected to a *Trust Monitor* that monitors properties of the available servers the virtual machines are running on. Based on the properties collected, the Trust Monitor derives security capabilities, such as the isolation mechanism of the environment, for the hosting servers. These capabilities are matched by the Policy Validation Module to select an appropriate server to provision, or in case of changing server capabilities, migrate a VM. The main difference to our approach is the enforcement point of policies. While Jamkhedar et al. enforce security requirements on the level of hypervisors, our approach is generic and can support various types of components, e.g., also PaaS-based deployments.

Waizenegger et al. [28] present two approaches to implement security policy enforcement based on TOSCA. The two approaches are the *IA-Approach* and *P-Approach*. Within the IA-Approach the Implementation Artifacts implementing Node Type Management Operations are extended to enable policy enforcing capabilities by implementing the same operations but with additional policy enforcing steps. The P-Approach extends the Provisioning Plan of an application with policy enforcing activities similar to our approach, but with the difference that the plans determine the policies to enforce at runtime. Additional differences to our approach are the missing generation of plans and the need for extending Implementation Artifacts for each policy type to support.

VII. Conclusion and Future Work

In this paper, we presented an approach that enables users to model and provision composite Cloud applications with their set non-functional requirements specified as Provisioning Policies. The approach extends our previous work in which we showed how to provision applications by transforming an application model into an executable Provisioning Plan. To transform such an application model our approach (i) generates a Provisioning Order Graph (POG) that specifies the order of provisioning, afterwards, this is (ii) transformed into a language-dependent Provisioning Plan Skeleton that has placeholders with the same order to provisioning the application. As the last step (iii), the placeholders are replaced with provisioning activities to generate an Executable Provisioning Plan. The approach presented in this paper extends our previous work by replacing placeholders of the Provisioning Plan Skeleton with activities to provision applications while enforcing specified non-functional requirements. This extension eases the modeling of non-functional requirements, as the user only has to specify the Provisioning Policies for his application without the need of deep technical knowledge. We validated our approach by a prototypical implementation within the OpenTOSCA Ecosystem and by applying our approach to a scenario with the focus on non-functional security requirements. In future work, we plan to decouple the provisioning logic from the policy enforcement logic and extend the set of policy types applicable to scenarios, such as the Internet of Things.

Acknowledgment

This work is partially funded by the projects SePiA.Pro (01MD16013F) and SmartOrchestra (01MD16001F) of the BMWi program Smart Service World.

References

[1] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems (USITS 2003). USENIX, Jun. 2003, pp. 1–16.

[2] F. Leymann, "Cloud Computing: The Next Revolution in IT," in Proceedings of the 52th Photogrammetric Week. Wichmann Verlag, Sep. 2009, pp. 3–12.

[3] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz et al., "Above the Clouds: A Berkeley View of Cloud Computing," University of California, Berkeley, Tech. Rep., 2009.

[4] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, and M. Wieland, "Policy-Aware Provisioning of Cloud Applications," in Proceedings of the Seventh International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2013). Xpert Publishing Services, Aug. 2013, pp. 86–95.

[5] U. Breitenbücher, T. Binz, C. Fehling, O. Kopp, F. Leymann et al., "Policy-Aware Provisioning and Management of Cloud Applications," International Journal On Advances in Security, vol. 7, no. 1&2, 2014.

[6] S. Subashini and V. Kavitha, "A survey on security issues in service delivery models of cloud computing," Journal of network and computer applications, vol. 34, no. 1, 2011, pp. 1–11.

[7] C. A. Ardagna, R. Asal, E. Damiani, and Q. H. Vu, "From Security to Assurance in the Cloud: A Survey," ACM Computing Surveys (CSUR), vol. 48, no. 1, 2015, p. 2.

[8] W. Han and C. Lei, "A survey on policy languages in network and security management," Computer Networks, vol. 56, no. 1, 2012, pp. 477–489.

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

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

[11] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann et al., "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.

[12] 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.

[13] OASIS, TOSCA Simple Profile in YAML Version 1.0, Organization for the Advancement of Structured Information Standards (OASIS), 2015.

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

[15] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, TOSCA: Portable Automated Deployment and Management of Cloud Applications, ser. Advanced Web Services. Springer, Jan. 2014, pp. 527–549.

[16] M. Mohaan and R. Raithatha, Learning Ansible. Packt Publishing, Nov. 2014.

[17] M. Taylor and S. Vargo, Learning Chef: A Guide to Configuration Management and Automation. O'Reilly, Nov. 2014.

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

[19] OMG, Business Process Model and Notation (BPMN) Version 2.0, Object Management Group (OMG), 2011.

[20] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, "BPMN4TOSCA: A Domain-Specific Language to Model Management Plans for Composite Applications," in Proceedings of the 4th International Workshop on the Business Process Model and Notation (BPMN 2012). Springer, Sep. 2012, pp. 38–52.

[21] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann, and T. Michelbach, "A Domain-Specific Modeling Tool to Model Management Plans for Composite Applications," in Proceedings of the 7th Central European Workshop on Services and their Composition, ZEUS 2015. CEUR Workshop Proceedings, May 2015, pp. 51–54.

[22] S. Walraven, D. Van Landuyt, A. Rafique, B. Lagaisse, and W. Joosen, "PaaSHopper: Policy-driven middleware for multi-PaaS environments," Journal of Internet Services and Applications, vol. 6, no. 1, 2015, p. 2.

[23] S. Ouyang, "Integrate Policy based Management and Process based Management–A New Approach for Workflow Management System," in Computer Supported Cooperative Work in Design, 2006. CSCWD'06. 10th International Conference on, IEEE. IEEE, 2006, pp. 1–6.

[24] A. Blehm, V. Kalach, A. Kicherer, G. Murawski, T. Waizenegger et al., "Policy-framework-eine methode zur umsetzung von sicherheits-policies im cloud-computing." in 44. Jahrestagung der Gesellschaft fr Informatik. GI, 2014, pp. 277–288.

[25] A. Keller, J. L. Hellerstein, J. L. Wolf, K.-L. Wu, and V. Krishnan, "The CHAMPS System: Change Management with Planning and Scheduling," in Proceedings of the 10th Network Operations and Management Symposium (NOMS 2004). IEEE, Apr. 2004, pp. 395–408.

[26] R. Mietzner, T. Van Lessen, A. Wiese, M. Wieland, D. Karastoyanova et al., "Virtualizing services and resources with probus: The ws-policy-aware service and resource bus," in Web Services, 2009. ICWS 2009. IEEE International Conference on, IEEE. IEEE, 2009, pp. 617–624.

[27] P. Jamkhedkar, J. Szefer, D. Perez-Botero, T. Zhang, G. Triolo et al., "A framework for realizing security on demand in cloud computing," in Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on, vol. 1, IEEE. IEEE, 2013, pp. 371–378.

[28] T. Waizenegger et al., "Policy4TOSCA: A Policy-Aware Cloud Service Provisioning Approach to Enable Secure Cloud Computing," in On the Move to Meaningful Internet Systems: OTM 2013 Conferences. Springer, Sep. 2013, pp. 360–376.