Institute of Architecture of Application Systems

Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA

Uwe Breitenbücher, Tobias Binz, Kálmán Képes, Oliver Kopp,
Frank Leymann, Johannes Wettinger

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

BibTeX:

```
@inproceedings{Breitenbuecher2014,
  author    = {Uwe Breitenb\"ucher and Tobias Binz and K\'{a}lm\'{a}n K\'{e}pes
               and Oliver Kopp and Frank Leymann and Johannes Wettinger},
  title     = {Combining Declarative and Imperative Cloud Application
               Provisioning based on TOSCA},
  booktitle = {Proceedings of the IEEE International Conference on Cloud
               Engineering (IEEE IC2E 2014)},
  year      = {2014},
  month     = {March},
  pages     = {87--96},
  doi       = {DOI 10.1109/IC2E.2014.56},
  publisher = {IEEE Computer Society}
}
```


Universität Stuttgart
Germany

Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA

Uwe Breitenbücher, Tobias Binz, Kálmán Képes, Oliver Kopp, Frank Leymann, Johannes Wettinger
Institute of Architecture of Application Systems
University of Stuttgart, Stuttgart, Germany
{breitenbuecher, lastname}@iaas.uni-stuttgart.de

***Abstract*—The automation of application provisioning is one of the most important issues in Cloud Computing. The Topology and Orchestration Specification for Cloud Applications (TOSCA) supports automating provisioning by two different flavors: (i) declarative processing is based on interpreting application topology models by a runtime that infers provisioning logic whereas (ii) imperative processing employs provisioning plans that explicitly describe the provisioning tasks to be executed. Both flavors come with benefits and drawbacks. This paper presents a means to combine both flavors to resolve drawbacks and to profit from benefits of both worlds: we propose a standards-based approach to generate provisioning plans based on TOSCA topology models. These provisioning plans are workflows that can be executed fully automatically and may be customized by application developers after generation. We prove the technical feasibility of the approach by an end-to-end open source toolchain and evaluate its extensibility, performance, and complexity.**

I. Introduction and Background

In recent years, Cloud Computing gained a lot of attention due to its economical and technical benefits. From an enterprise perspective, Cloud properties such as pay-on-demand pricing, scalability, and self-service enable outsourcing the enterprise's IT. This helps to achieve flexible, automated, and cheap IT operation and management [1]. On the other side, Cloud providers have to automate their internal Cloud management processes to achieve these properties for their Cloud offerings [2]. Especially the rapid provisioning of applications is of vital importance to enable self-service and pay-on-demand pricing. Therefore, one of the most important issues from a Cloud provider's perspective is to fully automate these provisioning processes.

The complexity of provisioning mainly depends on the application's structure and its components. Cloud applications typically consist of various types of components and employ several heterogeneous Cloud services. This complicates the provisioning because these components typically provide proprietary management APIs, are based on custom data formats, and use different technologies [2]. For example, IaaS offerings, such as Amazon EC2[1], are typically managed by using Web Service APIs whereas the installation of Web Servers, such as Apache Tomcat on an Ubuntu Linux operating system, is often done using script-centric technologies such as Chef[2] or Juju[3]. Thus, to fully automate the provisioning of such *composite Cloud applications*, different management technologies and APIs have to be integrated into one overall provisioning process. This is a difficult challenge because each technology and API comes with individual formats, domain-specific languages, invocation mechanisms, and prerequisites [2]. Thus, integrating different heterogeneous technologies increases the overall complexity. In addition, script-centric technologies often depend on the underlying infrastructure such as operating systems. Hence, *portability* of applications and their provisioning logic across different infrastructures can quickly degenerate to a serious problem. To tackle these issues, standardization efforts such as TOSCA are paving the way to enable standardized descriptions of Cloud applications and their management.

The *Topology and Orchestration Specification for Cloud Applications (TOSCA)* [3] is a new OASIS standard for describing Cloud applications and their management in a portable and interoperable way. TOSCA provides an XML-based language to model the structure of an application as *Topology Template*. A Topology Template describes all components and relations of an application. TOSCA applications are packaged as *Cloud Service Archives (CSARs)*, which contain all software artifacts required to provision, operate, and manage the application. CSARs run in standards-compliant *TOSCA Runtime Environments* hosted by providers. Thus, CSARs are portable across different providers and serve as exchange format for Cloud applications.

TOSCA supports management in two different flavors: (i) *imperative processing* and (ii) *declarative processing* [4]. Imperative processing requires that all needed management logic is contained in the CSAR. Therefore, CSARs contain fully automatically executable *Management Plans* that imperatively describe high-level management tasks such as provisioning, scaling, or updating the application. A Management Plan is a workflow orchestrating low-level management operations that are either provided by the application components themselves or by publicly accessible services, e. g., the Amazon Web Services API. To make plans portable, the *TOSCA Plan Portability API* is used by plans to communicate with the runtime, e. g., to access the Topology Template. Thus, CSARs containing plans are completely self-contained and able to package applications and their management functionalities in a portable and standardized format. The declarative processing flavor shifts management logic from plans to runtime: TOSCA Runtime Environments interpret application topologies to infer management logic without the need for plans. This requires a precise definition of the semantics of nodes and relations based on well-defined *Node Types* and *Relationship Types*. The set of provided management functionalities depends on the corresponding runtime and is not standardized by the TOSCA specification. In this paper, we focus only on one functionality: automated provisioning.

[1] http://aws.amazon.com/ec2
[2] http://www.opscode.com/chef
[3] https://juju.ubuntu.com

The two flavors offer application developers a means to automate the provisioning. However, both have advantages and drawbacks: the imperative approach enables application developers to define every detail of the provisioning explicitly by writing custom plans. The main drawback of creating plans manually is the labor-intensive nature of workflow authoring that is typically a hard, time-consuming, costly, and error-prone task: heterogeneous management services need to be orchestrated (e. g., SOAP-based services and RESTful services), script-centric technologies must be wrapped, and data formats must be handled—to name a few challenges [2], [5]. In addition, plans are tightly coupled to a certain application topology and sensitive to structural changes: different combinations of components lead to different plans [5]–[7]. Thus, plans for new applications often have to be created from scratch. Because Cloud application topologies quickly become complex, this manual creation of plans is not sufficient in many cases. The declarative approach solves this problem as plans are not needed. Obviously, this eases and speeds up the development of TOSCA applications. However, TOSCA Runtime Environments have to understand the components or at least the management operations they provide to infer provisioning logic. This limits provisioning capabilities to common types of components and pre-defined operations that are known and orchestrated by the runtime. In addition, application developers are not able to define complex custom provisioning logic that may be needed for the provisioning of complex applications. Thus, the declarative approach is rather suited for simple applications that consist of common components, relations, and technologies.

The result of the discussion above is that a combination of both flavors would enable application developers to benefit from automatically provided provisioning logic based on declarative processing and individual customization opportunities provided by adapting imperative plans. Thus, we need a means to generate Provisioning Plans automatically that can be customized by application developers afterwards. Therefore, the driving research question of this paper is: *How to generate executable Provisioning Plans for TOSCA-based Cloud applications that can be adapted afterwards?* We answer this question by presenting an approach that interprets TOSCA Topology Templates for generating executable Provisioning Plans implemented in general-purpose workflow languages. These generated plans may be customized by application developers afterwards for individual needs. Thus, the approach combines declarative and imperative provisioning of TOSCA applications. The paper shows how component-specific provisioning operations can be combined with generic lifecycle-based provisioning operations to provision TOSCA applications fully automatically based on generated plans. The approach enables to benefit from strengths of both flavors that leads to economical advantages when developing applications with TOSCA. We prove the feasibility of the approach by a prototypical implementation and present an evaluation that considers extensibility, standards-compliance, performance, and computational complexity.

The remainder of this paper is structured as follows: in Section II, we introduce TOSCA in detail and provide a motivating scenario in Section III. In Section IV, the main contribution of this paper is presented: we show a concept for a plan generator to generate Provisioning Plans based on TOSCA application topologies. We evaluate the approach in Section VI and give an outlook on future work in Section VIII.

II. TOSCA - The Topology and Orchestration Specification for Cloud Applications

In this section, we introduce the fundamentals of TOSCA. To ease reading, we describe only the important concepts required to understand the presented approach. In addition, details are left out and constructs are simplified in order to give a compact background. For details, we refer the reader to the TOSCA Specification [3] and the TOSCA Primer [4]. A compact overview on TOSCA is given by Binz et al. [8].

A *Topology Template* defines the structure of an application. It is a directed graph that consists of *Node Templates* (vertices) and *Relationship Templates* (edges). Node Templates represent components or software of an application such as virtual machines, operating systems, or services. Relationship Templates represent the relations between nodes, e. g., that a node is hosted on or calls another node. Node and Relationship Templates are typed: each Node Template has a certain *Node Type*, each Relationship Template a certain *Relationship Type*. These types are reusable and define properties and operations of the template. For example, an "ApacheWebServer" Node Type may define IP-address and credentials as properties and a "deploy"-operation that gets these properties and a reference to the files to be deployed as input parameters. Then, each Node Template of type "ApacheWebServer" provides this operation. Types can be inherited to enable reusability: the "ApacheWebServer" Node Type may be of supertype "WebServer" that defines general and common properties and operations of Web Servers.

Deployment Artifacts (DAs) implement the application's functionality. For example, a Deployment Artifact for a Web application of type "PHP" may be a ZIP file that contains all PHP files, images, and CSS files. A Deployment Artifact for an Apache Web Server may be a binary file that is used to install the Web Server on an operating system. Deployment Artifacts are typed and may define additional type-specific information. For example, the ZIP file for the Web application is modeled as a DA of type "ZIP" which defines a property that references the location of the ZIP file. Such application-specific DAs can be attached to Node Templates directly whereas reusable, application-independent DAs, such as Web Server installables, can be attached to the corresponding types. This enables reusing types and their artifacts in different Topology Templates.

Implementation Artifacts (IAs) implement the operations defined by Node Types and Relationship Types. Node Templates and Relationship Templates inherit these operations. Take as example the "ApacheWebServer" Node Type operation to deploy PHP applications that are packaged as ZIP file. For this operation, a corresponding Implementation Artifact is defined in the Node Type that implements the operation. IAs are typed and may contain additional type-specific information. For example, an IA implementing the operations defined by the "ApacheWebServer" Node Type may be of type "WAR". This means that the Implementation Artifact is implemented as Java Web Application and packaged using the WAR-format. To run and invoke this IA, type-specific information needs to be provided: (i) a reference to the corresponding WAR file, (ii) a deployment descriptor that defines how to run the IA, (iii) a property defining that it implements a SOAP Web Service, and (iv) a reference to the service's WSDL-file which contains all details about the service and how to invoke it. Implementation Artifacts are processed following

one of three kinds: (i) the TOSCA Runtime Environment runs IAs in its management environment, (ii) IAs run in the application's target environment, or (iii) IAs refer available services that can be called directly. Examples for the first kind of IAs are management services for components. The TOSCA runtime runs the IA of the "ApacheWebServer" Node Type in its local management environment to make the implemented operations accessible for the runtime (declarative approach) or plans (imperative approach). Thus, Node Types may bring their own management logic that runs in the TOSCA Runtime Environment. The second kind of IAs implement management logic that typically runs on the application's infrastructure, e. g., scripts. For example, LAMP-based applications (Linux, Apache, MySQL, PHP) consist of a PHP application that connects to a database. Such a connection may be established using a script that is copied onto the operating system of the PHP application and executed with corresponding parameters passed as environment variables: the script writes the endpoint information of the database to a certain location that is read by the PHP application. This enables implementing custom logic for individual applications that runs on the application's infrastructure directly, i. e., local to the application. The third kind of Implementation Artifacts are available services that are "always on", e. g., management APIs of Cloud providers. The corresponding Implementation Artifact only defines information needed to access these services, e. g., a WSDL-file.

TOSCA specifies an exchange format called *Cloud Service Archive (CSAR)* to package Topology Templates, types, associated artifacts, plans, and all required files into one self-contained package. This package is portable across different standards-compliant TOSCA Runtime Environments.

III. Motivating Scenario

In this section, we describe a motivating scenario that is used throughout the paper to explain the approach. The scenario describes a LAMP-based TOSCA application to be provisioned. The application implements a Web-shop in PHP that uses a database to store product and customer data. The application's Topology Template is shown in Figure 1. The visual notation used to render Topology Templates is "Vino4TOSCA" [9]. Therefore, names of Node Templates and Relation Templates are normal, undecorated text. The type of templates is enclosed by parentheses. The application consists of two application stacks: the left stack provides the infrastructure for the application logic whereas the right stack hosts the database. Both stacks run on Amazon's public IaaS offering "Amazon EC2". Therefore, a Node Template of Node Type "AmazonEC2VM" is used to model the virtual machines. This Node Type provides an operation "createVM" that is implemented by an Implementation Artifact of type "WAR". Invoking this operation instantiates a new virtual machine. The Implementation Artifact also specifies that the WAR implements a SOAP/HTTP Web Service by referencing the corresponding WSDL file. Similarly, the Node Template above of Node Type "UbuntuLinux" provides an Implementation Artifact to run scripts. We left out the details about Ubuntu version and properties such as SSH Credentials etc. to simplify the figure. Both nodes are connected by a Relationship Template of Relationship Type "hostedOn". Consequently, the source Node Template is hosted on the target Node Template of this relation. Up to this point, the structure and, thus,

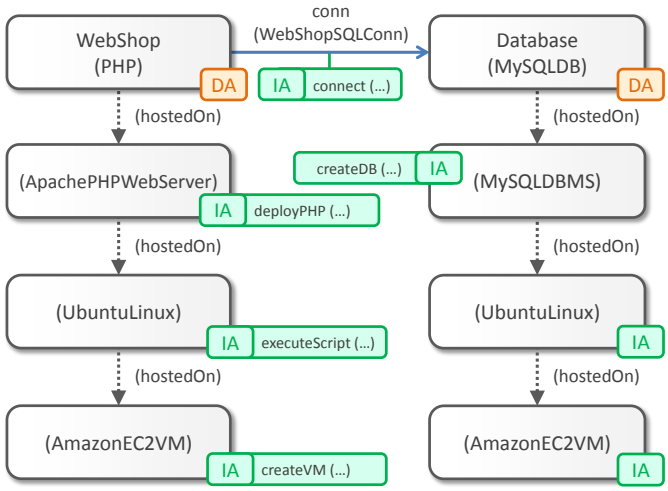

Fig. 1. TOSCA Topology Template describing a LAMP-based application.

the provisioning of both stacks are equal. The left stack runs the PHP application in an Apache Web Server. The corresponding Node Type "ApachePHPWebServer" provides an IA that implements a deploy-operation to deploy PHP applications. The corresponding "WebShop" Node Template provides a Deployment Artifact that refers to a ZIP file containing the application files. The DA also specifies its type "ZippedPHPApplication" to enable understanding the content of the artifact. On the right stack, similar operations are provided to instantiate a new database instance. A Deployment Artifact is attached to the "Database" Node Template that refers to an SQL script which creates tables and inserts default product data. In the last step, the two stacks have to be connected. This is done by using the Implementation Artifact of the Relationship Type "WebShopSQLConn" (which inherits from "SQLConnection") that is the type of the Relationship Template "conn" between the "WebShop" and "Database" Node Templates. The IA implements the connectToDatabase-operation defined by the supertype "SQLConnection" as shell script and defines that it has to be executed on the operating system of the source node, i. e., the operating system of the "WebShop" Node Template. Thus, the executeScript-operation of the "UbuntuLinux" Node Type is used. Of course, there are multiple ways to model this application. Especially whether the virtual machine Node Templates need an underlying infrastructure Node Template providing the createVM-operation or whether they provide the operation themselves depends on the modeling style. However, this does not affect the presented approach. Therefore, we minimized the number of nodes to simplify the scenario.

IV. A Provisioning Plan Generator for TOSCA

This section presents the main contribution of this paper. We propose an approach to generate *Provisioning Plans* for given Cloud Service Archives. A Provisioning Plan is a workflow that can be executed fully automatically to provision the Topology Template of a CSAR with all its Node Templates and Relationship Templates. Thus, running a Provisioning Plan *instantiates* a new application instance. Provisioning Plans are, therefore, used to implement higher-level provisioning logic for entire applications by orchestrating lower-level operations that provision single Node Templates or Relationship Templates.

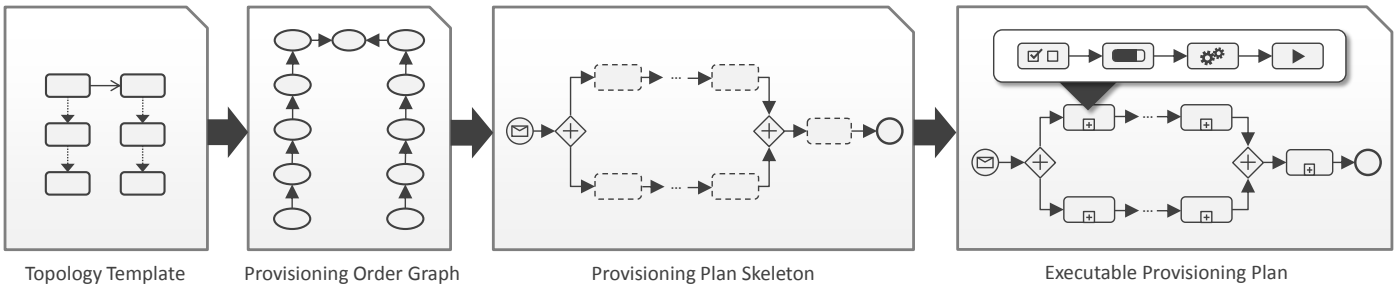

Fig. 2. Provisioning Plan Generation Concept: A Topology Template is used to generate a Provisioning Order Graph that is transformed into a Provisioning Plan Skeleton to be completed to an Executable Provisioning Plan. For space reasons, we left out some activities in both plan diagrams (depicted as "...").

A. Use Case and Role Model

Before the plan generation is presented in detail, we introduce the overall use case and role model. A *Cloud Service Creator* [4] develops a CSAR containing the Topology Template of the application to be provisioned and all required types, artifacts, and files—without Provisioning Plan. The creator employs the presented *Plan Generator* to generate a Provisioning Plan fully automatically. After the generation, the creator may adapt the generated plan to customize or complete the Provisioning Plan. The final CSAR is then sent to a *TOSCA-enabled Cloud Provider* hosting a TOSCA Runtime Environment. The provider is responsible for installing the CSAR and maintaining the environment infrastructure. Plans and IAs are deployed by the runtime fully automatically. To instantiate application instances, providers typically offer Web-based APIs or self-service portals to start Provisioning Plans.

B. Conceptual Approach

In this section, we describe a high-level overview on the plan generation. Details are explained in the following subsections. The approach is shown in Figure 2 and subdivided into three steps: first, the CSAR to be provisioned serves as input for the (i) generation of a *Provisioning Order Graph*. This graph is (ii) translated into a *Provisioning Plan Skeleton* which (iii) gets finally completed to an *Executable Provisioning Plan*.

In the first step, the Topology Template contained in the input CSAR is analyzed and transformed into a workflow language-independent Provisioning Order Graph (POG). This graph defines the order in which the topology's nodes and relations have to be provisioned. Each vertex in the POG represents the task to provision a certain Node or Relationship Template. The directed edges between two vertices define the temporal provisioning order: the vertex at the source of the edge must be processed, i. e., provisioned, before the vertex at the edge's target. This definition is based on Mietzner [10] but extended to support the explicit provisioning of relationships, too. All details of this step are explained in Section IV-C.

In the second step, the Provisioning Order Graph is used to generate a Provisioning Plan Skeleton. This skeleton is implemented in a certain workflow language, e. g., BPEL [11] or BPMN [12], but defines only the structure of the final Provisioning Plan. Thus, the skeleton is not executable and contains only *Empty Provisioning Activities* for each Node and Relationship Template to be provisioned. These activities are placeholders for the actual provisioning logic. Therefore, vertices in the Provisioning Order Graph get transformed into Empty Provisioning Activities, POG edges into control constructs between the respective activities. Thus, the partial order of provisioning steps defined by the POG is retained. The details of this step are explained in Section IV-D.

In the last step, the Provisioning Plan Skeleton gets completed to an Executable Provisioning Plan, which is a fully automatically executable workflow. Therefore, each Empty Provisioning Activity is replaced by one or multiple *Provisioning Subprocess Templates* that implement the actual logic to provision a certain Node or Relationship Template based on its type. The resulting plan may be adapted by the Cloud Service Creator afterwards to customize the provisioning. The details of this step are explained in Section IV-E.

C. Provisioning Order Graph Generation

This section describes the generation of the Provisioning Order Graph in detail. The input for this step is the Topology Template of the CSAR that contains all components and relations to be provisioned or established as Node Templates or Relationship Templates, respectively. Each vertex in the Provisioning Order Graph represents the task to provision a certain Node Template or Relationship Template. Each edge connecting two vertices defines the temporal order of the tasks.

The order in which Node and Relationship Templates have to be provisioned depends on the types of the Relationship Templates. For example, if a Relationship Template of type "hostedOn" connects two Node Templates, the target Node Template of the relation must be instantiated before the Relationship Template and the source Node Template. The source node is hosted on the target node, thus, the target node must exist before. If two Node Templates are connected by a Relationship Template of type "connectsTo", source and target Node Templates must be instantiated before the Relationship Template can be instantiated: it is neither possible to establish a connection *from* a non-existing node nor *to* a non-existing node. Similar to "hostedOn" are "dependsOn", "runsOn", "installedOn", etc. All these types require the existence of the target node before the relation and the source node can be instantiated. We define "dependsOn" as supertype for all these types. Thus, each of these Relationship Types inherits from type "dependsOn". Equally, we define "uses" as supertype for "calls", "invokes", "connectsTo", etc., because all these relations require the existence of both nodes before the relation can be instantiated. In the following, we consider these two supertypes as basis for the presented approach.

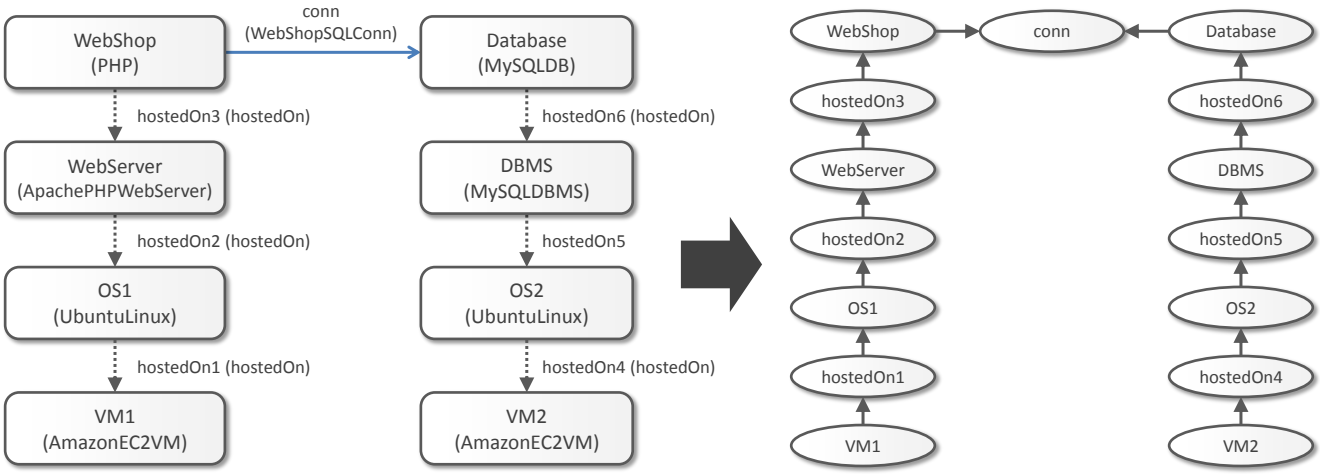

Fig. 3. Provisioning Order Graph Generation: LAMP-based motivating scenario Topology Template (left) gets transformed into Provisioning Order Graph (right).

To generate the POG, we introduce a *Provisioning Order Graph Generation Algorithm*. The algorithm gets the Topology Template as input and consists of two steps: (i) creating a vertex in the POG for each Node and Relationship Template in the topology and (ii) connecting the created vertices based on the types of the Relationship Templates. For Relationship Templates of type "dependsOn", we define the following provisioning order: target Node Template → Relationship Template → source Node Template. For Relationship Templates of type "uses", the orders are: target Node Template → Relationship Template and source Node Template → Relationship Template. Figure 4 shows the algorithm in pseudocode: first, vertices are created in the POG for all Node Templates. Afterwards, vertices are created for each Relationship Template. Based on the Relationship Template's type, two edges are inserted into the POG: for "dependsOn", we insert an edge from the target Node Template's vertex to the relationship's vertex and another edge from the relationship's vertex to the source Node Template's vertex. For "uses", we insert an edge from the target Node Template's vertex to the relationship's vertex and another edge from the source Node Template's vertex to the relationship's vertex.

Figure 3 shows an example for applying the algorithm. The Topology Template of the motivating scenario serves as input. The output is a POG that contains a vertex for each Node and Relationship Template. The vertices are connected following the definition of "dependsOn" and "uses": both application stacks lead to successive provisioning tasks except the "conn" Relationship Template that is of type "WebShopSQLConn", which inherits from "uses". Therefore, both stacks have to be provisioned before the connection can be established.

procedure GENERATEPOG($V_{topology}, E_{topology}$)
  Let $G_{POG} = (V_{POG}, E_{POG})$ an empty directed graph
  for all $v \in V_{topology}$ **do**
    $V_{POG} \leftarrow V_{POG} \cup \{v\}$
  end for
  for all $e \in E_{topology}$ **do**
    $v_{rel} \leftarrow$ new vertex for e
    $V_{POG} \leftarrow V_{POG} \cup \{v_{rel}\}$
    if $basetype(e) =$ "$dependsOn$" **then**
      $E_{POG} \leftarrow E_{POG} \cup \{(tar(e), v_{rel}), (v_{rel}, src(e))\}$
    else if $basetype(e) =$ "$uses$" **then**
      $E_{POG} \leftarrow E_{POG} \cup \{(tar(e), v_{rel}), (src(e), v_{rel})\}$
    end if
  end for
end procedure

Fig. 4. Provisioning Order Graph Generation Algorithm.

D. Provisioning Plan Skeleton Generation

In this section, we describe how a Provisioning Order Graph gets transformed into a Provisioning Plan Skeleton. The plan skeleton defines the structure of the Provisioning Plan to be generated. Thus, it is implemented in the same workflow language as the resulting final Executable Provisioning Plan. This step is the first one that is workflow-language specific. Therefore, we employ workflow language-specific *Provisioning Plan Skeleton Generators* to transform the Provisioning Order Graph into the Provisioning Plan Skeleton. The Skeleton Generators are responsible for transforming the POG into a workflow language-specific skeleton with Empty Provisioning Activities and may inject additional language-specific navigation activities such as gateways, subprocesses, or structured loops. The way Empty Provisioning Activities are transformed into an element in the workflow language that has the semantics of a placeholder depends on the workflow language. For example, in BPMN, we use "Abstract Tasks", in BPEL "Opaque Activities". To abstract from concrete languages, we refer to these language-specific constructs also as Empty Provisioning Activities in the following. Each Empty Provisioning Activity has to store the ID of the Node or Relationship Template that is provisioned by the activity. This is required for injecting the actual provisioning logic in the next step. Universally applicable transformation rules are defined as follows [10]: (i) vertices of the POG that have no incoming edges can be executed in parallel once the workflow starts. (ii) After the template represented by a vertex is provisioned, each outgoing edge is evaluated to true and the target provisioning task (i. e., the target vertex) can be executed. (iii) If multiple edges leave a provisioned vertex, the semantics are equal to an "and-split" (or "parallel-split" [13]) [10]. Thus, all provisioning tasks represented by the corresponding target vertices are executed in parallel. (iv) If a vertex has multiple

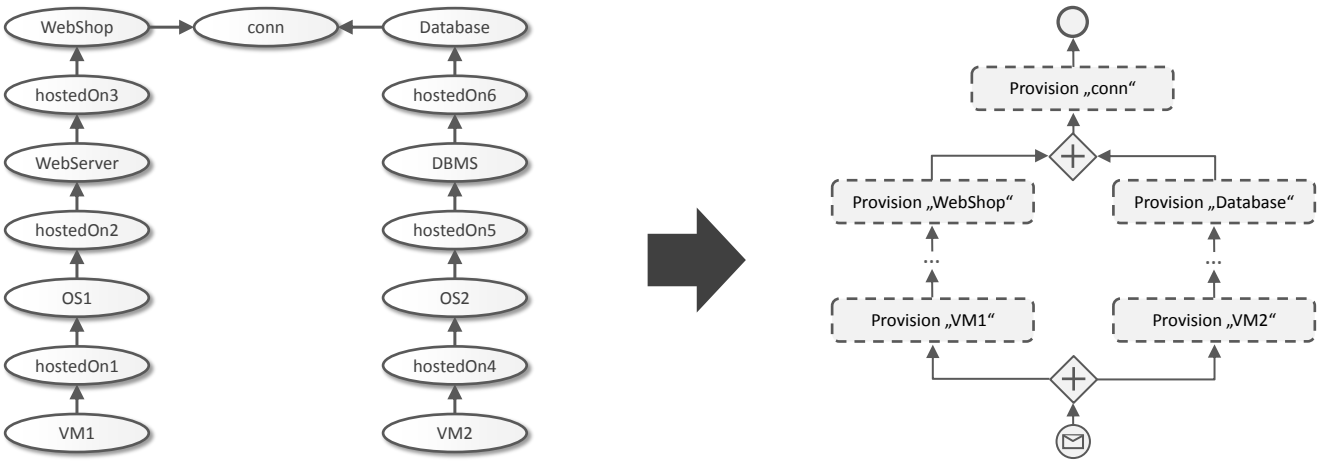

Fig. 5. The Provisioning Order Graph (left) gets transformed into a Provisioning Plan Skeleton (right) in a specific workflow-language, e. g., BPMN.

incoming edges, the corresponding provisioning task must not be executed until all tasks represented by the source vertices of those edges are completed. This corresponds to an "and-join" [13]. Figure 5 shows an example for transforming the Provisioning Order Graph of the motivating scenario to a BPMN skeleton. For Empty Provisioning Activities, the generator inserts "Abstract Tasks". Thus, each Node or Relationship Template is provisioned by one of these tasks. For parallel execution of activities, parallel gateways are employed.

E. Provisioning Plan Completion

In this section, we explain how a Provisioning Plan Skeleton gets completed to an Executable Provisioning Plan. Therefore, a workflow language-independent *Plan Completion Manager* is used to manage the replacement of Empty Provisioning Activities. The overall procedure is shown in Figure 6: the manager gets Topology Template, Provisioning Order Graph, and Provisioning Plan Skeleton as input. It traverses the POG following the provisioning order defined by vertices and edges and calls a workflow language-specific *Provisioning Logic Provider (PLP)* for each vertex in the POG. The PLPs insert provisioning logic for the corresponding Node or Relationship Template into the skeleton. Thus, the skeleton gets refined iteratively in this completion step. After the manager processed all vertices, all Empty Provisioning Activities have been replaced by provisioning logic and the skeleton became an Executable Provisioning Plan. In the following, we explain Provisioning Logic Providers and POG traversal in more detail. A Provisioning Logic Provider is responsible for replacing Empty Provisioning Activities by executable provisioning logic. Each PLP registers in the *Provisioning Capabilities Table* (shown in Figure 6 in the middle) for one or multiple Node or Relationship Types it *may* be able to provide the corresponding provisioning logic in a certain workflow language. Therefore, the Provisioning Capabilities Table maps tuples of (Element Type, Workflow Language) to PLPs. For example, "PLP 3" is registered for the provisioning of Node Templates of type "MySQLDB" in BPEL and BPMN. Thus, it may be able to provide the provisioning logic in these two workflow languages to create a MySQL database. In general, the PLP is able to provide the provisioning logic for the element types it has registered for. However, for certain constellations, a registered PLP may not be able to insert the required logic. For example, the PLP that installs a MySQL database on an operating system may be able to do this installation for a certain set of Linux operating systems, but not for Windows. Another example is "PLP n" in Figure 6, which is registered for "SQLConnection". This kind of PLPs typically has preconditions that must be checked by the PLP itself, e. g., if the source component provides a certain operation to set the database endpoint information. Thus, there are typically more than one PLPs

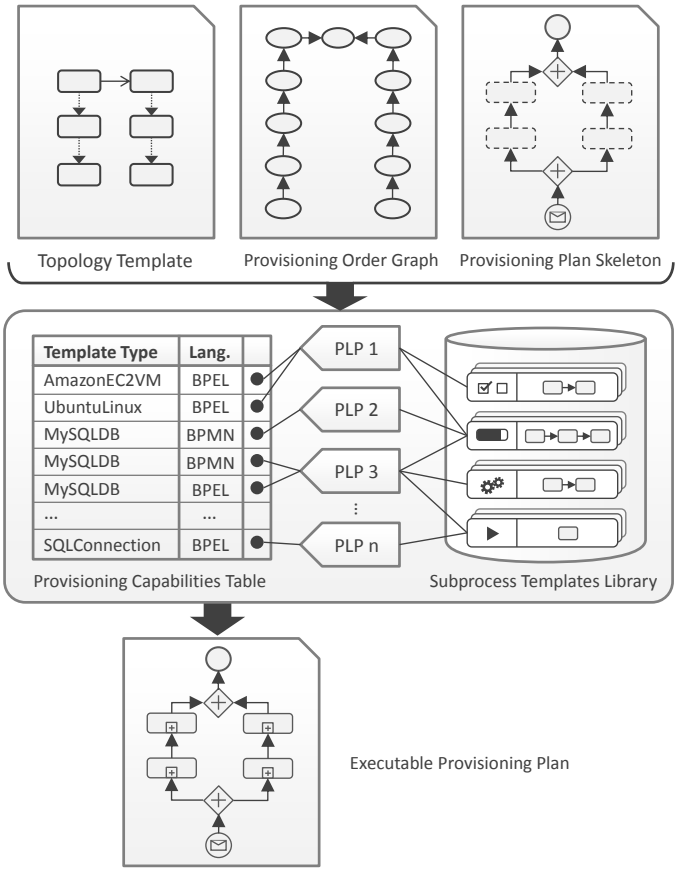

Fig. 6. Plan Completion Manager Architecture.

registered for one tuple that differ in their preconditions, e. g., "PLP 2" and "PLP 3". The Plan Completion Manager tries to find one capable. Therefore, the Plan Completion Manager processes the POG as described in the following paragraph.

The order in which Empty Provisioning Activities are replaced follows the provisioning order defined by the POG. Traversing the workflow language-independent POG keeps this procedure abstract and avoids that the Plan Completion Manager has to understand different workflow languages. This makes the approach extensible as new workflow languages can be added seamlessly. The manager starts with vertices in the POG that have no incoming edges and follows the provisioning order specified by outgoing edges. For each vertex, it looks capable *candidate PLPs* up in the Provisioning Capabilities Table. Therefore, the manager compares the type of the template and the language of the skeleton with table entries: each PLP that has registered for (i) a type that is equal or a supertype of the template's type and (ii) for the language of the skeleton in one entry, is a candidate PLP. If one or multiple candidate PLPs are found, the manager selects the one whose registered type is closest to the template's type in terms of inheritance. For example, in our motivating scenario, we use a "WebShopSQLConn" Relationship Type that inherits from "SQLConnection". If there are registered PLPs for "WebShopSQLConn", the manager selects one of them first. If not and there are only PLPs for "SQLConnection", the manager tries these supertype PLPs. This enables defining custom types that implement operations defined by their supertypes, which can be used by higher-level PLPs that are not aware of the custom type to provide the required provisioning logic based on these operation implementations. After selecting a candidate, the manager passes the current Provisioning Plan Skeleton, the ID of the template to be provisioned (extracted from the POG vertex), and the Topology Template to the PLP that has registered for the closest type. Therefore, each PLP implements a uniform interface that defines these three input parameters. PLPs use the ID to retrieve the template to be provisioned from the Topology Template. If the PLP is able to provision this template in the context of this topology, it injects the provisioning logic into the skeleton. Therefore, it replaces the corresponding Empty Provisioning Activity, which is identified by the template's ID, by executable workflow code. If the injection was successful, it returns the enriched Provisioning Plan Skeleton. If not, i. e., the PLP is not able to provide the required logic, the PLP returns "null". In this case, the manager tries the next candidate PLP that is registered for the considered tuple following the rules defined above. If there is no capable PLP for a template, the corresponding Empty Provisioning Activity remains unchanged in the skeleton. Developers may replace this activity manually afterwards if explicit provisioning logic is required which is not provided by the available PLPs but needed to obtain a correct provisioning plan. However, this is not always necessary: if a template requires no explicit provisioning logic, there is no need to replace the corresponding Empty Provisioning Activity as it represents only a placeholder and does not affect the workflow's execution. An example for templates that often do not require explicit provisioning logic are "hostedOn" relations: if a node is hosted on another node, this relation gets often established implicitly when the node is deployed. Thus, the relation needs not to be instantiated explicitly. However, if a special type of "hostedOn" is needed, the type system of TOSCA allows to create a subtype that is processed by a special PLP. As a result, different types can be handled specifically and templates that are not *explicitly* provisioned by a PLP may be provisioned *implicitly* by another PLP. After the manager processed all vertices in the POG, it returns the Executable Provisioning Plan, which may be checked and adapted manually to ensure correctness and completeness.

To replace Empty Provisioning Activities, PLPs use, configure, adapt, and combine *Provisioning Subprocess Templates*, which are small workflows implementing provisioning logic. They enable integrating various management technologies such as proprietary APIs or script-centric configuration management tools, e. g., Chef or Puppet, seamlessly into the overall provisioning plan by implementing one or multiple of the following *Provisioning Phases*: (i) *Prepare Phase*: in this phase, the provisioning gets prepared. For example, script Implementation Artifacts are copied onto the corresponding operating system. (ii) *Install Phase*: this is the main step of the provisioning that installs / instantiates the corresponding node or relation. (iii) *Configure Phase*: in this phase, the provisioning is configured. For example, credentials of an installed Web Server Node Template are changed. (iv) *Start Phase*: in the last phase of the provisioning lifecycle, the component gets started or the relation gets established. These phases are an extension of the TOSCA Lifecycle Interface [4] focusing on provisioning. Provisioning Subprocess Templates are generic building blocks that can be combined by PLPs to separate concerns. For example, one template may be used to *prepare* the installation of a Web Server by copying Chef-scripts onto the underlying operating system whereas another template executes them in the *installation* phase. In this example, both templates have a generic character: copying and executing scripts is independent from the Node Type to be provisioned. Thus, these subprocess templates can be reused by other PLPs. Therefore, templates are stored in a *Subprocess Templates Library*. In contrast, a subprocess template may be specific for the provisioning of a certain Node or Relationship Type. For example, a PLP that instantiates a virtual machine on Amazon EC2 has to do a sequence of HTTP calls to the Amazon Web Service API. Such templates are explicitly built for the provisioning of one particular type.

Provisioning Subprocess Templates are documented and identified by an unique URI. This enables PLP developers to reuse them: PLPs retrieve subprocess templates from the library, copy them into the skeleton, and inject orchestration logic. In this step, templates are typically configured, i. e., placeholders and variables in the subprocess get replaced or initialized, respectively. For example, the aforementioned subprocess template that copies a script onto an operating system uses two variables to identify the (i) Node Template providing the script-IA and the (ii) operating system Node Template. These variables have to be initialized by the PLP. Therefore, PLPs use mechanisms of the respective workflow language, e.g., by injecting an assign activity in BPEL.

F. Type-specific and Generic PLPs

The PLPs described in the previous sections are called *Type-specific PLPs* as they are built to provision a Node or Relationship Template of a certain type. However, provisioning logic is often independent from types, e. g., if only scripts have to be executed to install and start a component. Therefore,

we introduce *Generic Provisioning Logic Providers (GPLP)* to decouple generic, reusable provisioning logic from specific Node or Relationship Types: GPLPs provision elements based on operations provided by management interfaces independently from the template's type. A GPLP requires, therefore, a certain management interface and a certain Implementation Artifact (IA) implementing this interface. To identify management interface types and IA types clearly, TOSCA uses unique URIs. Thus, different Node or Relationship Types may provide a specific interface implemented by a specific IA type that is understood by a GPLP—independently from the actual template type itself. For example, a Node Type may provide the operations defined by the TOSCA Lifecycle Interface [4] (identified by a unique URI) and Implementation Artifacts of type "ShellScript" that implement these operations. Then, a GPLP that exactly knows how to handle this combination of interface and Implementation Artifacts is capable of provisioning this template: the GPLP may traverse the Node Templates below the template to be provisioned following "hostedOn" relations until it finds a Linux operating system Node Template that provides properties to access the system via SSH, i. e., IP-address, SSH user, and SSH password properties. These properties can be used by the GPLP to configure the script-subprocess templates introduced above that copy and execute scripts on an Linux operating system. This approach enables developers to implement custom, reusable provisioning logic based on custom, reusable management interfaces and Implementation Artifact types. GPLPs register in the Provisioning Capabilities Table for any type (using the wildcard symbol "*") but for a certain workflow language. When a GPLP is called by the Plan Completion Manager, the GPLP inspects the template to be provisioned and replaces the corresponding Empty Provisioning Activity in the skeleton in case the needed interface is provided and implemented by a suitable IA. Otherwise, it returns null. Thus, both PLP types are processed equally. However, GPLPs are only called if no type-specific PLP is capable to ensure that available type-specific provisioning logic is preferred to generic provisioning logic.

The concept of GPLPs, typed management interfaces, and typed Implementation Artifacts enables implementing different technical realizations of provisioning logic for abstract management interfaces. As an example, the shell script Implementation Artifact above may have been created based on a particular Unix shell such as Bash. This script cannot be used on completely different platforms such as the Windows operating system. To provide a loosely coupled approach, TOSCA enables implementing one management interface by multiple Implementation Artifacts of possibly different types, e. g., "WindowsScript". Thus, a Node Type may provide different script-based Implementation Artifacts for one management interface that can be used by different GPLPs to install the node on multiple different operating system Node Types. The approach also provides an integration layer for different technologies: type-specific PLPs can integrate technologies transparently whereas GPLPs allow integrating individual management logic directly through consuming custom Implementation Artifacts that implement a well-known abstract management interface.

V. Architecture and Prototype

To prove the technical feasibility of the presented approach, we implemented a Java prototype. Implementation details can be found in [14]. The architecture is shown in Figure 7: The "Provisioning Plan Generation Manager" is a high-level component that manages the generation. CSAR Importer and Exporter are used to extract the contents from CSARs and put the generated Provisioning Plans into it. The "POG Generator" is a workflow language-independent component that consumes CSARs and generates a POG. For POGs, we used a Java workflow model that enables defining partial task orderings. The "Skeleton Generator" is a plugin-based system to generate skeletons in a certain workflow language. These skeletons are then completed by the language-independent "Plan Completion Manager", which provides a plugin mechanism to register PLPs. All plugins are based on OSGi. All kinds of workflows, i. e., subprocess templates, skeletons, and plans, are implemented in BPEL. We conducted case studies in which we implemented BPEL-based PLPs to support end-to-end provisioning examples. For example, to deploy LAMP-based / Java-based applications on Amazon using different services such as IaaS (Amazon EC2) and PaaS (Amazon Beanstalk, Amazon RDS).

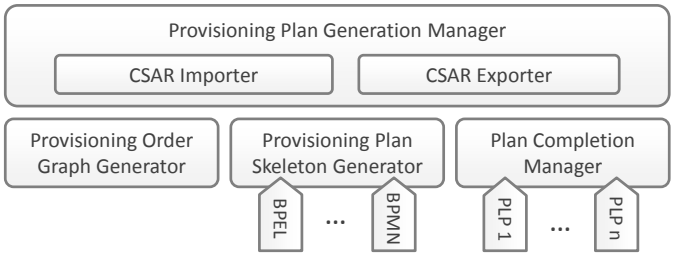

Fig. 7. Architecture of the Plan Generation Component.

VI. Evaluation

A. Computational Complexity and Performance

The complexity of the complete plan generation is linear, i. e., $\mathcal{O}(n)$ with n being the number of templates in the topology: (i) generating a POG can be done in linear time as each template is processed exactly once. (ii) Translating the POG into a skeleton depends on the used language. However, for BPEL and BPMN, this translation can be done in linear time following the description in Section IV-D. (iii) For replacing Empty Provisioning Activities in the last step, at most all registered PLPs are invoked for each activity. As the number of PLPs is fixed, the complexity is linear. We exclude analyzing PLPs because their complexity depends on custom requirements. However, we found that this is also possible in linear time for all PLPs used in our case studies. We conducted a performance measurement based on our prototype to underpin the theoretical analysis. The experimental setup was based on a MacBookPro with Intel Core 2 Duo (2.53GHz) and 4 GB RAM. We created Topology Templates of different size and measured the time needed to generate Provisioning Plans (including serialization to BPEL files). The results shown in Table I indicate that the required time increases linearly to the number of templates.

TABLE I. Plan Generation Duration

# Templates	∅ Plan Generation	∅ Time / Template
50	0.833 s	0.016 s
100	1.574 s	0.016 s
200	3.038 s	0.015 s
1000	14.489 s	0.015 s
5000	77.394 s	0.015 s

B. Standards Compliance and Portability

Standards are a means to achieve reusability, interoperability, portability, and maintainability of software and hardware leading to a higher productivity and help to align the enterprise's IT to its business. However, most available provisioning approaches are based on proprietary APIs or domain-specific languages (DSLs). For example, Amazon provides a proprietary API to manage Cloud offerings. Script-centric technologies such as Chef employ proprietary DSLs based on Ruby. However, none of these DSLs is standardized. This prevents these approaches to be portable across other technologies or environments. In addition, each proprietary DSL has to be learned by developers. This is a difficult and time-consuming task. The approach presented in this paper addresses these issues as it exclusively employs standards: the OASIS standard TOSCA is used to describe applications, standard workflow languages such as BPEL or BPMN are used to implement Provisioning Plans. Thus, the approach enables portability of applications and their management plans based on standards.

C. Toolchain—End-to-End Prototype Support

The presented approach bridges the gap between declarative application modeling and imperative provisioning. Therefore, a modeling tool and a capable TOSCA Runtime Environment are needed to provide end-to-end support. We developed an open source TOSCA modeling tool called "Winery" [15] that provides a graphical user interface for modeling Topology Templates and a management backend to manage types and artifacts. Winery provides export and import functionality for CSARs. Thus, the tool can be used to create CSARs and adapt existing CSARs afterwards, e. g., to customize the generated plan. We also developed a TOSCA Runtime Environment called "OpenTOSCA" [16]. This runtime is able to run CSARs containing Java-based Web Service Implementation Artifacts and BPEL Management Plans, which are bound to the IAs by the runtime. Thus, the generated plans can be executed by this environment fully automatically. These two tools can be combined with the presented approach as shown in Figure 8: Winery is used to create CSARs that contain no Provisioning Plan. These CSARs are processed by the plan generator which generates and injects a Provisioning Plan for the Topology Template into the CSAR which runs in OpenTOSCA. Thus, we provide tools supporting full end-to-end support.

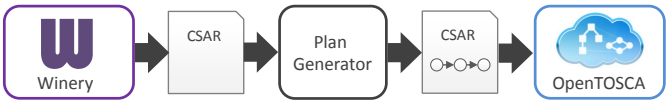

Fig. 8. End-To-End Toolchain.

D. Extensibility

The presented approach is extensible on multiple layers. First, workflow languages can be added by implementing Provisioning Plan Skeleton Generators and associated PLPs. Therefore, the framework offers a plugin interface for generators. The Provisioning Capabilities Table provides a generic means to register PLPs independently from the employed language. Thus, no additional effort is needed to integrate a new language into the Plan Completion Manager. Second, using custom Node and Relationship Types is supported as the framework is completely independent from Node Types and provides a means to integrate new Relationship Types by inheriting from the two super Relationship Types "dependsOn" and "uses". In addition, types provide an abstract description of their management interfaces and properties. Hence, they are independent from the employed workflow language. Third, operating IAs and binding plans to them is up to the used TOSCA Runtime Environment. Thus, if the environment supports a certain workflow language, the generated plan can be executed without further changes.

VII. Related Work

The Cafe-framework [10] automates provisioning of Cloud applications based on generating Provisioning Plans by orchestrating "Component Flows" that implement a uniform interface to provision components. Cafe also generates a POG which gets transformed into an executable plan. However, it is not possible to model different kinds of relations between nodes explicitly—except the deployedOn-dependency. Therefore, the POG generation is based on inverting deployedOn-dependencies and calculating dependencies based on variability points. In contrast to this, our approach determines provisioning order directly from typed relationships. A major strength of our approach is the ability to consider the provisioning of relationships explicitly based on their types. This enables implementing custom provisioning logic for relationships without the need to modify existing Node Types, their management interfaces, or provided management operations.

Lu et al. [17] present concept and implementation of a deployment service that is similar to our approach in terms of defining the application's structure through a topology model that is used for provisioning. The approach searches provisioning actions for each component contained in the topology, similar to our approach. However, they execute the operations (implemented using Chef) directly without generating an explicit plan that can be modified afterwards. In addition, the approach presented by Lu is much more restrictive: Lu et al. employ an own declarative, XML-based domain-specific language (DSL) to model applications whereas our approach is based on the TOSCA standard that enables defining portable application descriptions. Their proprietary DSL is based on virtual machines whereon software can be deployed automatically using Chef. The approach is not able to include various kinds of Cloud services and different provisioning, configuration, and management technologies. In contrast, because our approach uses TOSCA as topology definition language, it is extensible in terms of Node Types. Thus, any Node Type can be defined in TOSCA and extend our system. In addition, our approach is not restricted to Chef and supports integrating any technology by writing custom Provisioning Logic Provider that may employ and utilize any technology to provision a certain Node or Relationship Type.

There are several works [5]–[7], [18], [19] that attempt to bridge the gap between imperative provisioning logic and a model describing the provisioning declaratively based on AI planning and graph covering techniques. In these works, so-called *desired state models* are used to describe the state into which an application shall be transferred. These models are used to find a partial order of provisioning operations, workflows, or scripts that transfer the application into this state.

Therefore, AI planning and graph covering techniques are used to analyze dependencies between different nodes, relations, properties, and the effects and preconditions of operations to generate a workflow that brings the application into the desired state. The exclusively type-based PLP concept is a simplified variant of these techniques as it only compares template types with PLP capabilities and employs exactly one PLP to insert provisioning logic for a node or relation, which is sufficient for the provisioning of most applications according to Eilam et al. [5]. However, also the PLP concept enables a detailed analysis: the templates to be provisioned can be analyzed programmatically to ensure injecting correct provisioning logic: complex queries and graph traversals can be executed by a PLP to analyze the context in all details. Also more complex concepts such as Generic Lifecycle Management Planlets [2] that enable to provision nodes ore relations based on provided operations can be realized with the presented approach: GPLPs, which are related to this concept, provide similar means.

The CHAMPS System [20] focuses on Change Management, which modifies IT systems through so-called "Requests For Change (RFC)", e. g., provisioning or configuration requests. The approach analyzes dependencies between components, relations, and the effects of an RFC. Based on this analysis, so-called "Task Graphs" are generated that carry out the RFC. These graphs are workflows that are scheduled by a "Planner and Scheduler" afterwards to achieve a high degree of parallelism. In a later work [21], the authors show how BPEL is used to integrate CHAMPS with general-purpose process engines and deployment engines that execute the tasks orchestrated by the generated workflow. Compared to our approach, the explicit consideration of relations to achieve a high degree of parallelism is similar. However, our approach is more extensible as it enables integrating various kinds Node and Relationship Types and provisioning approaches through the concept of PLPs: different deployment engines, management APIs, and script-centric technologies can be integrated seamlessly for individual provisionings as specific logic is outsourced to dedicated, type-specific or generic PLPs.

The script-centric DevOps community provides tooling such as Chef, Puppet, or Juju to simplify configuration management. However, integrating different kinds of provisioning technologies, e. g., proprietary management APIs, is not supported directly. Script-centric technologies operate on a deep technical level and manually orchestrating different scripts, recipes (Chef), charms (Juju), or manifests (Puppet) quickly degenerates to a hard and error-prone task. In contrast, our approach employs a high-level modeling language to describe provisionings declaratively and abstracts from most technical details. In addition, all these related works do not support TOSCA as modeling language and exchange format for Cloud applications.

VIII. Conclusion and Future Work

In this paper, we presented an approach to generate imperative Provisioning Plans based on declarative TOSCA models. We showed how dedicated, type-specific plugins are capable of analyzing and configuring provisioning of TOSCA Node and Relationship Templates and how reusable subprocesses can be combined to provision components also generically. The results are fully automatically executable plans that can be customized by application developers. In the future, we plan (i) to support non-functional requirements such as security policies, (ii) to extend the approach by concepts of the planning theory to handle complex use cases, and (iii) to integrate the plan generator into Winery and OpenTOSCA for reducing the number of required tools.

Acknowledgment

This work was partially funded by the BMWi project CloudCycle (01MD11023).

References

[1] F. Leymann, "Cloud Computing: The Next Revolution in IT," in *Proc. 52th Photogrammetric Week*, September 2009, pp. 3–12.

[2] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, and J. Wettinger, "Integrated Cloud Application Provisioning: Interconnecting Service-centric and Script-centric Management Technologies," in *CoopIS*, September 2013, pp. 130–148.

[3] OASIS, *Topology and Orchestration Specification for Cloud Applications Version 1.0*, May 2013. [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html

[4] *Topology and Orchestration Specification for Cloud Applications Primer Version 1.0*, OASIS, January 2013. [Online]. Available: http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.html

[5] T. Eilam, M. Elder, A. Konstantinou, and E. Snible, "Pattern-based composite application deployment," in *IFIP/IEEE IM*, May 2011, pp. 217–224.

[6] E. Maghraoui *et al.*, "Model driven provisioning: bridging the gap between declarative object models and procedural provisioning tools," in *Middleware*, November 2006, pp. 404–423.

[7] U. Breitenbücher *et al.*, "Pattern-based runtime management of composite cloud applications," in *CLOSER*, Mai 2013, pp. 475–482.

[8] T. Binz, G. Breiter, F. Leymann, and T. Spatzier, "Portable Cloud Services Using TOSCA," *IEEE Internet Computing*, vol. 16, no. 03, pp. 80–85, May 2012.

[9] U. Breitenbücher *et al.*, "Vino4TOSCA: A visual notation for application topologies based on tosca," in *CoopIS*, September 2012, pp. 416–424.

[10] R. Mietzner, "A method and implementation to define and provision variable composite applications, and its usage in cloud computing," Dissertation, University of Stuttgart, Germany, August 2010.

[11] OASIS, *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*, OASIS, Apr. 2007.

[12] OMG, *Business Process Model and Notation (BPMN), Version 2.0*, Object Management Group Std., Rev. 2.0, January 2011.

[13] van der Aalst *et al.*, "Workflow patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, July 2003.

[14] K. Képes, "Konzept und Implementierung eine Java-Komponente zur Generierung von WS-BPEL 2.0 BuildPlans für OpenTOSCA," Bachelor Thesis, University of Stuttgart, IAAS, 2013.

[15] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, "Winery – a modeling tool for TOSCA-based cloud applications," in *ICSOC*, 2013, pp. 700–704.

[16] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "OpenTOSCA – a runtime for TOSCA-based cloud applications," in *ICSOC*, 2013, pp. 692–695.

[17] H. Lu *et al.*, "Pattern-based deployment service for next generation clouds," in *SERVICES*, 2013, pp. 464–471.

[18] K. Levanti and A. Ranganathan, "Planning-based configuration and management of distributed systems," in *IFIP/IEEE IM*, June 2009, pp. 65–72.

[19] H. Herry, P. Anderson, and G. Wickler, "Automated planning for configuration changes," in *LISA*, 2011.

[20] A. Keller *et al.*, "The CHAMPS system: change management with planning and scheduling." in *NOMS*, April 2004, pp. 395–408.

[21] A. Keller and R. Badonnel, "Automating the provisioning of application services with the BPEL4WS workflow language," in *DSOM*, November 2004, pp. 15–27.