2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing

Standards-based DevOps Automation and Integration Using TOSCA

Johannes Wettinger, Uwe Breitenbücher, Frank Leymann

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

***Abstract*—DevOps is an emerging paradigm to tightly integrate developers with operations personnel. This is required to enable fast and frequent releases in the sense of continuously delivering software. Users and customers of today's Web applications and mobile apps running in the Cloud expect fast feedback to problems and feature requests. Thus, it is a critical competitive advantage to be able to respond quickly. Beside cultural and organizational changes that are necessary to implement DevOps in practice, tooling is required to implement end-to-end automation of deployment processes. Automation is the key to efficient collaboration and tight integration between development and operations. The DevOps community is constantly pushing new approaches, tools, and open-source artifacts to implement such automated processes. However, as all these proprietary and heterogeneous DevOps automation approaches differ from each other, it is hard to integrate and combine them to deploy applications in the Cloud. In this paper we present a systematic classification of DevOps artifacts and show how different kinds of artifacts can be transformed toward TOSCA, an emerging standard in this field. This enables the seamless and interoperable orchestration of arbitrary artifacts to model and deploy application topologies. We validate the presented approach by a prototype implementation, show its practical feasibility by a detailed case study, and evaluate its performance.**

***Keywords*—DevOps; Deployment Automation; Transformation; TOSCA; Chef; Juju; Cloud Standards; Cloud Computing**

I. Introduction

The traditional split between developers and operations found in many organizations today is a major obstacle for fast and frequent releases of software. This is due to different goals, contrary mindsets, and incompatible processes owned by these two groups. For instance, developers want to push changes into production as fast as possible, whereas the operations personnel's main goal is to keep production environments stable [1]. For this reason, collaboration and communication between developers and operations personnel is mainly based on slow, manual, and error-prone processes. Consequently, it takes a significant amount of time to put changes, new features, and bug fixes into production. However, especially users and customers of Web applications and mobile apps expect fast responses to their changing and growing requirements. Thus, it is a competitive advantage to implement automated processes to enable fast and frequent releases. But this is only possible by closing the gap between development and operations. *DevOps* [2] is an emerging paradigm to bridge this gap between these two groups, thereby enabling efficient collaboration. Beside organizational and cultural challenges to eliminate the split, the *deployment process* needs to be highly automated to enable continuous delivery of software [3]. The constantly growing DevOps community supports this by providing a huge variety of individual approaches such as tools and artifacts to implement holistic deployment automation. Reusable DevOps artifacts such as scripts, modules, and templates are publicly available to be used for deployment automation. Juju charms and bundles[1] as well as Chef cookbooks[2] are examples for these [4], [5]. In addition, Cloud computing [6], [7] is heavily used to provision the underlying resources such as virtual servers, storage, network, and databases. DevOps tools and artifacts can then configure and manage these resources. Thus, end-to-end deployment automation is efficiently enabled by using the DevOps approaches in Cloud environments.

However, DevOps artifacts are usually bound to certain tools. For instance, Chef cookbooks require a Chef runtime, whereas Juju charms need a Juju environment to run. This makes it challenging to reuse different kinds of heterogeneous artifacts in combination with others. Especially when systems have to be deployed that consist of various types of components, typically multiple of management tools have to be combined: they typically focus on different kinds of middleware and application components. Thus, there is a variety of solutions and orchestrating the best of them requires to integrate the corresponding tools, e.g., by writing workflows or scripts that handle the individual invocations, the parameter passing, etc. However, this is a difficult, costly, and error-prone task as there is no means to do this integration in a standardized manner, supporting interoperability of the orchestrated artifacts. Therefore, the goal of our work is to enable the seamless integration of different kinds of DevOps artifacts based on the emerging OASIS standard *TOSCA* (Topology and Orchestration Specification for Cloud Applications) [8]. In this paper we present the major contributions of our work:

- We present an initial classification of DevOps artifacts and outline their usage
- We show a generic methodical framework to transform DevOps artifacts into standards-based TOSCA models that can be orchestrated arbitrarily to model and deploy new applications

[1] Juju Charm Store: http://jujucharms.com
[2] Chef Supermarket: https://supermarket.getchef.com

978-1-4799-7881-6/14 $31.00 © 2014 IEEE
DOI

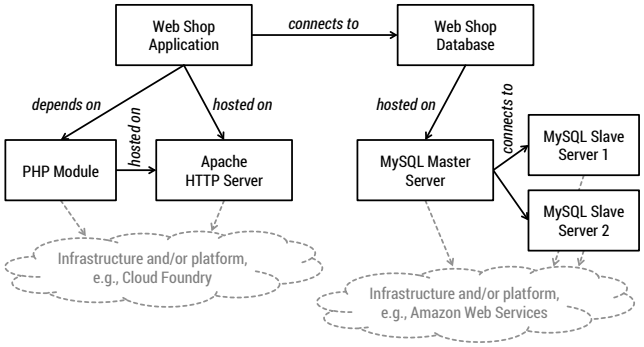

Figure 1. Web shop application and database topology

- We apply the presented framework to implement transformation methods that generate reusable and interoperable TOSCA models for two different DevOps approaches: Juju charms and Chef cookbooks
- We evaluate our framework and the implementations of the transformation methods based on the motivating scenario

The remainder of this paper is structured as follows: Section II describes the problem statement and presents a motivating scenario that is used as a running example. The fundamentals to understand our work is shown in Section III, including a classification of DevOps artifacts and the explanation of Chef, Juju, and TOSCA. The core of our work, namely the transformation framework and method as well as the technical transformations for Chef and Juju are shown in Section IV. Based on the motivating scenario, Section V presents an evaluation of our framework and our implementation. Section VI and Section VII present related work and conclude the paper.

II. Problem Statement and Motivating Scenario

The DevOps community actively shares open-source artifacts such as scripts, modules, and templates to deploy middleware and application components. Their portability and community support make them a predestined means to reuse them to automate the deployment of different kinds of applications, especially Web applications and Web-based back-ends for mobile apps. As long as only a single type of artifacts is used, the artifacts are typically interoperable and a single corresponding tooling may be utilized, e.g., a Chef runtime for Chef cookbooks. However, using and combining artifacts of different kinds such as Chef cookbooks and Juju charms in a seamless manner is a major challenge. Extra effort is required to learn and integrate all their peculiarities such as invocation mechanisms, state models, parameter passing, etc. In addition, the orchestration of different artifacts and tools must be implemented using workflows or scripts that integrate them on a very low-level of abstraction. This requires deep technical insight in the corresponding technologies and the overall orchestration approach. Typically, a lot of glue code is required that makes the overall orchestration hard to understand and maintain for non-experts. In the following we show a motivating scenario as a concrete example to confirm the necessity of integrating different kinds of artifacts.

Figure 1 shows a part of the topological structure of the Web shop application inspired from [9]. The application itself is hosted on an Apache HTTP server and depends on the PHP module; the database of the application is hosted on a MySQL master/slave environment to improve the application's scalability and to enable high availability of the database: data that are written to the master instance are consistently replicated to the slave instances, so reading requests can be load-balanced between slave instances. In case the master instance breaks, a slave instance can be selected to be the new master instance. The underlying infrastructures and/or platforms could be chosen depending on certain requirements or preferences. For instance, the Apache and MySQL servers could be hosted on virtual machines provided by Amazon Web Services[3]. To implement deployment automation for this application we want to reuse existing DevOps artifacts, especially to deploy the middleware components. For instance, Chef cookbooks may be used to deploy the Apache HTTP server and the PHP module, assuming that these are running on a single virtual machine. However, there is no Chef cookbook to deploy a complete MySQL master/slave environment out of the box. Consequently, we may better use the MySQL charm[4] shared by the Juju community to deploy and dynamically scale such a MySQL setup. This adds another kind of artifact to the deployment automation implementation, implying the learning, usage, and orchestration of additional tooling to handle and execute corresponding artifacts. Finally, we may have implemented custom Unix shell scripts to meet specific deployment and operations requirements of our Web shop application and its database. These scripts are used to deploy the application-specific parts of the topology. Consequently, there is yet another kind of artifact involved in the deployment automation process that needs to be integrated, too. Thus, three different kinds of deployment tools must be combined to deploy the application efficiently by reusing existing artifacts that are optimally suited.

Cloud standards such as TOSCA tackle this challenge of seamlessly integrating and combining different kinds of artifacts by introducing a unified meta model. TOSCA modeling artifacts can be used and composed seamlessly to create application models that can be deployed automatically. TOSCA is an emerging standard, but it still lacks an ecosystem of communities and reusable artifacts. However, an ecosystem based on open-source communities is key to establish TOSCA in practice [10], [11]. Because the DevOps community provides such an ecosystem but is mostly based on individual and proprietary approaches, the goal of our work is to *transform existing DevOps artifacts toward TOSCA to make them reusable and composable in a seamless and interoperable manner.* To provide all required information for understanding our approach, the following section provides a classification of DevOps artifacts and explains the basic modeling constructs of TOSCA.

III. Fundamentals

In this section we discuss the fundamentals on which our work is based. Section III-A provides an initial classification of DevOps artifacts to clarify their conceptual and technical differences. We consider a representative of each class in our further discussions. TOSCA's most important concepts

[3]Amazon Web Services: http://aws.amazon.com
[4]MySQL charm: http://jujucharms.com/precise/mysql-46

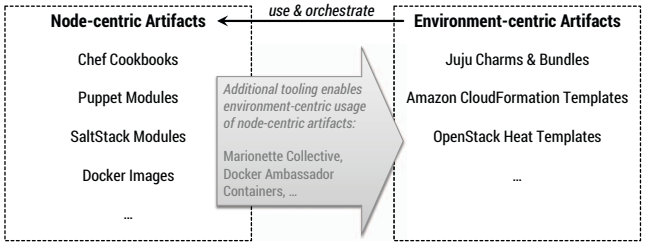

Figure 2. Initial classification of DevOps artifacts

and modeling constructs are presented in Section III-D as a foundation to discuss the transformation in Section IV.

A. Classification of DevOps Artifacts

As discussed in Section II, there is a huge and ever-growing amount and variety of artifacts shared by the DevOps community to deploy middleware and application components. Because these artifacts differ in how they are designed and how they are used, Figure 2 presents an initial classification of DevOps artifacts based on two major classes:

1) *Node-centric artifacts (NCAs)* are scripts, images, modules, declarative configuration definitions, etc. that are *executed on a single node* such as a physical server, a virtual machine (VM), or a container [12], [13]. Cross-node relations such as an application component connecting to a database running on another node are not explicitly expressed and implemented. Consequently, NCAs are not meant to be used to deploy complete application topologies.

2) *Environment-centric artifacts (ECAs)* are scripts, bundles, templates, etc. that are *executed in an environment, potentially consisting of multiple nodes.* Cross-node relations are explicitly expressed and implemented. Consequently, ECAs can be used to deploy complete application topologies.

Unix shell scripts, Chef cookbooks, Puppet modules[5], SaltStack modules[6], and Docker images[7] are a few prominent examples for NCAs. In terms of ECAs, Juju charms and bundles, Amazon CloudFormation templates[8], and OpenStack Heat templates[9] are representatives of this class of artifacts. However, node-centric and environment-centric artifacts are not meant to be used exclusively. In fact, ECAs typically use and orchestrate NCAs. For instance, Juju charms utilize Unix shell scripts to install, configure, and wire software components on VMs. Moreover, Amazon CloudFormation templates[10] can utilize Chef cookbooks to implement deployment logic. Beside the orchestration of NCAs using ECAs, additional management tooling can be utilized to enable the environment-centric usage of NCAs. As an example, Marionette Collective[11] to manage the distribution and execution of Chef cookbooks in large-scale environments, providing consistent, environment-specific context information to the nodes and NCAs involved. Docker ambassador containers[12] are another means to distribute and host containers based on Docker images [14] in a multi-node environment, e.g., consisting of several VMs.

The presented classification based on NCAs and ECAs is the foundation for our transformation approach. There are several other aspects that may be considered when choosing corresponding artifacts to implement deployment automation. These aspects include:

- *Level of dependencies:* some artifacts are *provider-dependent* such as CloudFormation templates, i.e., they can only be used in combination with a certain provider such as Amazon in this case. Other artifacts are *tooling-dependent* such as Chef cookbooks or Juju charms: they can be used in conjunction with different providers, but require certain tooling such as a Chef runtime or a Juju runtime.

- *Level of virtualization:* artifacts may depend on certain virtualization solutions such *hypervisor-based virtualization* (e.g., Amazon machine images) or *container virtualization* (e.g., Docker images) [13].

- Especially environment-centric artifacts can be distinguished in *infrastructure-centric* vs. *application-centric* artifacts. Infrastructure-centric artifacts such as CloudFormation templates focus on the configuration and orchestration of infrastructure resources such as VMs, storage, and network. Juju bundles are much more application-centric by focusing on the configuration and orchestration of middleware and application components and transparently managing the underlying infrastructure.

- There are *definition-based* artifacts such as Chef cookbooks and Puppet modules, defining the configuration of resources such as VMs or containers. On the other hand *image-based* artifacts such as Docker images capture the state of a certain resource to create new instances by restoring the persisted state on demand.

- Definition-based artifacts can be created in a *declarative,* in an *imperative,* or in a combined manner. For instance, Chef cookbooks typically define the desired state of a resource using a declarative domain-specific language [4], [15]. However, imperative statements can also be part of such artifacts. Unix shell scripts typically consist of a set of imperative command statements.

In the following we explain Chef and Juju in more detail to highlight how NCAs (Chef cookbooks) and ECAs (Juju charms) are used in practice. Chef and Juju are representatives for their corresponding class. Moreover, we refine the positioning of these two approaches regarding the classification aspects discussed in this section. Our technical transformation implementations are based on Chef and Juju as we will discuss in Section IV-A and Section IV-B.

[5]Puppet Forge: https://forge.puppetlabs.com
[6]SaltStack modules: http://goo.gl/l1mcnr
[7]Docker Hub Registry: http://index.docker.io
[8]Amazon CloudFormation templates: http://goo.gl/NzOe3
[9]OpenStack Heat: http://wiki.openstack.org/wiki/Heat
[10]Integration of Amazon CloudFormation with Chef: http://goo.gl/pOsU0
[11]Marionette Collective: http://docs.puppetlabs.com/mcollective

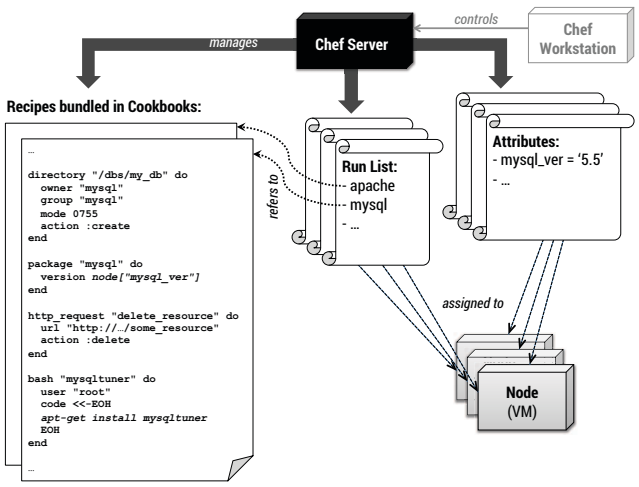

Figure 3. Chef overview: artifacts, components, and interrelations

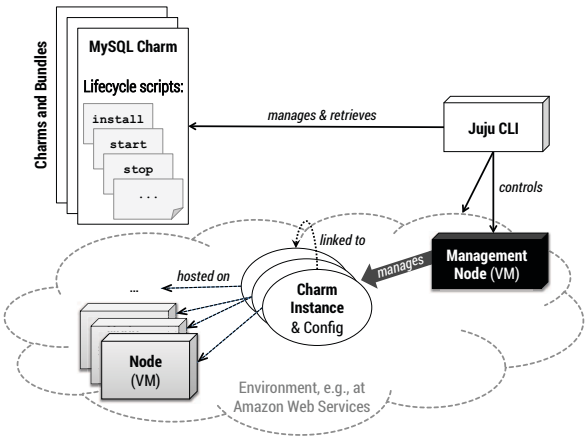

Figure 4. Juju overview: artifacts, components, and interrelations

B. Chef

Chef [4], [5] is a configuration management framework that provides a domain-specific language *(Chef DSL)* based on Ruby. The Chef DSL is used to define configurations of resources such as VMs. These configuration definitions are called *recipes.* Multiple recipes are bundled in *cookbooks.* For instance, a *MySQL* cookbook may provide the recipes *install_server, install_client,* and *install_all* to deploy the corresponding components of MySQL. The definition of declarative expressions such as *ensure that MySQL server is installed* is the recommended way to define portable configurations in recipes. However, imperative expressions such as system command statements (e.g., "`apt-get install mysql-server`") can be used, too. Figure 3 provides an overview of the core concepts of Chef: the Chef server acts as a central management instance for the recipes, run lists, and attributes. Each node has a run list and a set of attributes assigned. The run list of a particular node specifies which recipes have to be executed on this node. Because most recipes are created to be used in different ways, they have some variability points such as the version number of the *mysql* package as shown in the sample recipe in Figure 3. To resolve these variability points at execution time, attribute definitions such as *mysql_ver = '5.5'* can be assigned to a node. These attributes can be read during execution time. Finally, a Chef workstation runs *knife*[13] to control the Chef server as well as all nodes and data that are registered with the Chef server. To sum it up: according to our classification aspects presented in Section III-A, Chef is a *tooling-dependent* but not provider-dependent solution that supports *different levels of virtualization:* Chef recipes can be executed on physical servers, VMs, and containers. Chef is *infrastructure-centric* because it focuses on the distribution and execution of recipes on infrastructure resources such as VMs. Because Chef recipes are typically *declarative* scripts, although they may include *imperative* statements as well, Chef follows the idea of creating and maintaining *definition-based* artifacts. Chef recipes are *node-centric artifacts* because they run in the scope of a single node.

[12]Docker ambassador containers: http://goo.gl/PzGfcH

[13]Chef knife: http://docs.opscode.com/knife.html

C. Juju

In contrast to Chef, Juju[14] follows an environment-centric approach to implement configuration management and deployment automation. Charms contain scripts to implement a well-defined lifecycle of a certain component such as a MySQL server. For instance, scripts to *install, start,* and *stop* a MySQL server are contained in the MySQL charm. The lifecycle scripts can be implemented in an arbitrary language (Python, Ruby, Unix shell, etc.), as long as the resulting scripts are executable on the target nodes. Juju does not prescribe the usage of any domain-specific language to create these scripts. Figure 4 denotes the environment-centric focus of Juju: charm instances and their configurations live in a certain environment such as an interconnected set of VMs hosted at Amazon Web Services. Charm instances can be linked such as an application component that is connected to a database. These links are explicitly expressed and can be configured. In contrast to Chef, charms and charm instances are the main entities; the underlying nodes and the scripts that are executed on them provide the required infrastructure to host the charm instances. It is important to denote that a charm instance is not limited to the scope of a single node. For instance, to deploy a multi-node MySQL database server in a master/slave setup, a single charm instance can be used that is scaled in and out by adding and removing *units*. An example is shown in Figure 5, where the MySQL charm instance is hosted on three VMs. By default, one unit is one VM. When a new charm instance is created, it initially consists of one unit.

Beside the VMs that host units for the charm instances, a management node runs in each environment to deploy, manage, and monitor the charm instances. Conceptually, the management node can be compared to a Chef server. However, each environment requires its own management node, so different environments are independent of each other. Interconnections between management nodes are currently not supported, so multi-Cloud applications are difficult to deploy and manage by Juju. A Juju command-line interface (CLI) is provided to control both the management node and the environment itself, e.g., to configure security aspects such as firewalls. Different environments can be managed based on different infrastructures

[14]Ubuntu Juju: https://juju.ubuntu.com

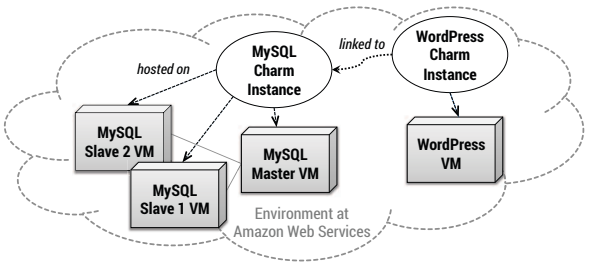

Figure 5. Juju sample: MySQL and WordPress running at Amazon

(OpenStack [16], Amazon Web Services, Microsoft Azure, HP Cloud, etc.). Moreover, the Juju CLI tool is used to manage and retrieve charms from public and private charm stores. This is the foundation for creating charm instances in any environment. Based on our classification aspects outlined in Section III-A, Juju is a *tooling-dependent* solution, currently mostly focused on *hypervisor-based virtualization* because VMs are the unit of currency. Contrary to Chef, Juju can be considered as *application-centric* because the underlying infrastructure and scripts are abstracted by charms and charm instances. Furthermore, charms are *environment-centric artifacts* because links between them are explicitly expressed and can be configured. Charms are *definition-based* because of their lifecycle scripts that are typically implemented using *imperative* scripting languages. However, links between charms are expressed in a *declarative* manner by defining requirements that can be met by certain capabilities provided by other charms.

In terms of integration, Chef recipes could be reused as NCAs to implement the lifecycle scripts of Juju charms, which are ECAs. However, the integration requires deep technical knowledge of both approaches (state models, parameter passing, invocation mechanisms, etc.) and couples them tightly. If even further approaches and technologies come into play, all these have to be integrated separately. Consequently, a common, standardized intermediate meta model is required to efficiently and seamlessly integrate different approaches and technologies. In the following Section III-D we introduce the Topology and Orchestration Specification for Cloud Applications (TOSCA) [8] that fulfills these meta model requirements.

D. TOSCA

The Topology and Orchestration Specification for Cloud Applications (TOSCA) [8] is an emerging standard, supported by several companies in the industry[15]. Its main goal is to enhance the portability and management of Cloud applications. Technically, TOSCA is specified using an XML schema definition. *Topology templates* are defined as graphs consisting of nodes and relationships to specify the topological structure of an application as, for instance, shown in Section II (Figure 1). As a foundation for defining such templates, *node types* and *relationship types* are defined as shown in Figure 6. These are used to create corresponding *node templates* and *relationship templates* based on them in the topology template. A complete type system can be introduced because types may be derived from other existing types in the sense of inheritance as it is used, for instance, in object-oriented programming. As an

[15]TOSCA tech. committee: https://www.oasis-open.org/committees/tosca

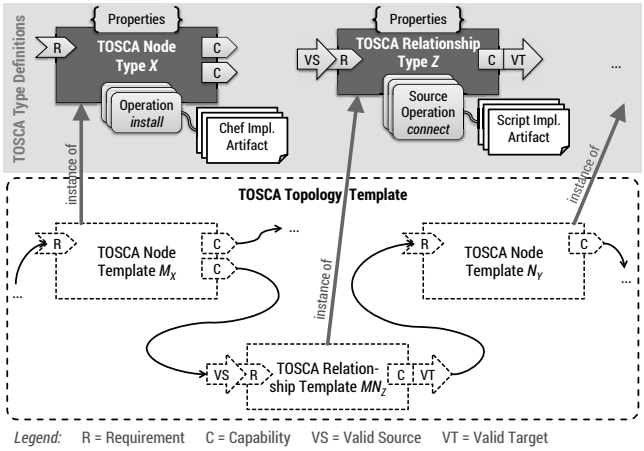

Figure 6. Type definitions and templates in TOSCA

example, an abstract *Java Servlet Container* node type can be defined, which has a child node type *Apache Tomcat*. There could be further node types derived from this one, such as *Apache Tomcat 6.0, Apache Tomcat 7.0,* etc.

Types consist of further sub-elements: *operations* are attached to nodes and relationships, for instance, to cover their lifecycle (*install, start, stop,* etc.). In addition, further management operations may be defined such as *backup_database* and *restore_database*. These operations are implemented by *implementation artifacts (IAs),* which could be, for instance, Chef recipes or Unix shell scripts. An IA is executed when the corresponding operation is invoked, e.g., by the TOSCA runtime environment. Operations belonging to relationships are distinguished in *source operations* and *target operations* because a relationship links a source node with a target node. Source operations are executed on the source node, target operations on the target node. To enable the linking of nodes and relationships, they expose *requirements* and *capabilities* that are used for matchmaking purposes. For instance, a Java application node may expose a *Java Servlet Runtime* requirement, whereas the the *Apache Tomcat* node provides a matching *Java Servlet Runtime* capability. A relationship specifying the *Java Servlet Runtime* requirement as valid source and the *Java Servlet Runtime* capability as valid target can be used to wire the two nodes. *Properties* can be defined as arbitrary data structures in XML schema to make nodes and relationships configurable. As an example, the *Apache Tomcat* node may provide a property to specify the directory where the log files should be written to. All properties are exposed to the operations and their IAs, so they can be considered during execution. Finally, TOSCA specifies the structure of *Cloud Service Archives (CSAR)* as a portable, self-contained packaging format for Cloud applications. Not only the XML definitions of types and templates are part of a CSAR; it also contains all scripts and files that are referenced, e.g., as IAs. Consequently, a CSAR is self-contained, enabling a TOSCA runtime environment to process it by traversing the topology template to create application instances. Each node provides lifecycle operations such as *install* that are called when the topology template gets traversed to create instances of the nodes and relationships on a real infrastructure. TOSCA supports the deployment and management of applications modeled as

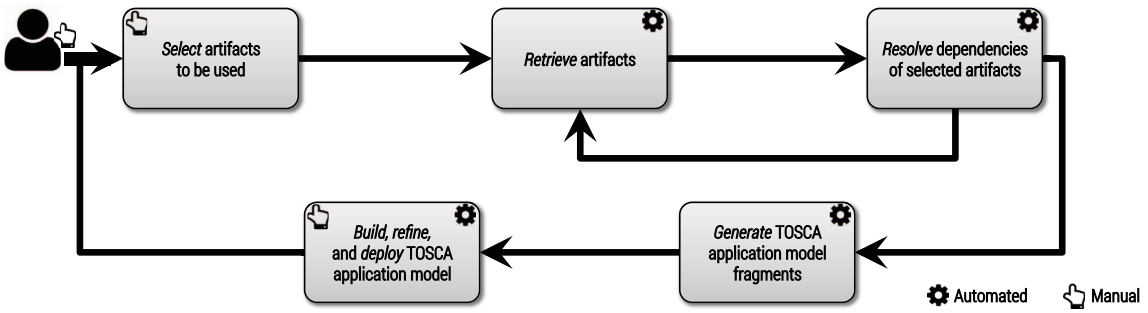

Figure 7. Overview of methodical framework for DevOps artifact transformation

topology templates by two different flavors: (i) imperative processing and (ii) declarative processing [17]. The *imperative approach* employs so called *management plans* that orchestrate the management operations provided by node and relationship types. Thus, they execute the implementation artifacts that are attached to the corresponding operation. These plans can be executed automatically and are typically implemented using workflow languages such as *BPEL* [18], *BPMN* [19], or the BPMN extension *BPMN4TOSCA* [20]. To enable completely self-contained CSARs, management plans can be stored directly in the corresponding CSAR. Thus, imperative TOSCA runtime engines run these plans to consistently execute management tasks. In contrast to the imperative approach, the *declarative approach* does not require any plans: a declarative TOSCA runtime engine derives the corresponding logic automatically by interpreting the topology template. We do not distinguish between the two flavors in the following as our approach is agnostic to this difference: it transforms DevOps artifacts to TOSCA implementation artifacts that can be either (i) orchestrated using management plans or (ii) used by declarative engines.

IV. Transformation Framework and Method

In order to transform arbitrary DevOps artifacts of different classes toward TOSCA to make them usable and composable in a seamless and interoperable manner, we present a transformation framework and method as shown in Figure 7. The initial step is the manual selection of artifacts such as Chef recipes, Juju charms, and Docker images to be used for a specific application. This selection may be driven by the individual middleware requirements of the application, e.g., a PHP-based Web application requires a PHP runtime and it may need a MySQL database. Based on the selection, the artifacts are automatically retrieved, e.g., from the public Juju charm store. Then, all dependencies of the retrieved artifacts are checked automatically. As a result, further artifacts may have to be retrieved to resolve these dependencies. These artifacts may have further dependencies that need to be resolved afterward. Therefore, the method iterates these two steps until all dependencies are resolved. In the next step, TOSCA node types and relationship types are generated based on the selected and retrieved artifacts. This is the technical transformation of DevOps artifacts toward TOSCA. The transformation logic is highly specific to the individual DevOps approaches. In the following Section IV-A and Section IV-B we explain the technical transformations for Chef and Juju in more detail.

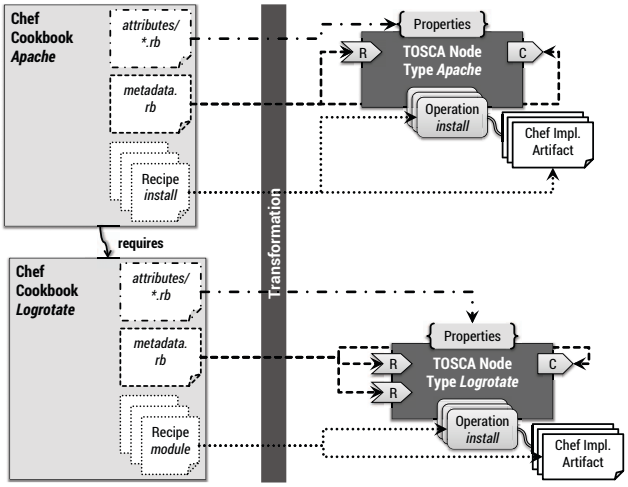

Figure 8. Fine-grained transformation: one node type per Chef cookbook

After the technical transformations have been performed and the TOSCA types were generated, these can be used in the context of appropriate modeling tooling such as Winery [21] to build TOSCA application models using topology templates. Such an application model can be packaged as CSAR to deploy it using a TOSCA runtime environment such as OpenTOSCA [22]. Alternatively, to refine the model, additional artifacts may have to be selected and retrieved, including dependency resolution to generate additional TOSCA types. This may be the case if the application requires additional components that cannot be covered by artifacts selected before. The usage of a TOSCA modeling tool in conjunction with the generated artifacts is not limited to building application models. For instance, the generated node types can be enriched by attaching additional management operations and corresponding IAs.

A. Technical Transformation of Chef Cookbooks

The technical transformation of DevOps artifacts to TOSCA types is the core of our transformation framework described before in Section IV. In this section we present the concepts of a technical transformation of Chef cookbooks that bundle Chef recipes. We provide two different alternatives to perform the transformation of Chef cookbooks: (i) the *fine-grained transformation* and (ii) the *coarse-grained transformation.* As shown in Figure 8 for the fine-grained approach, a node type is created for each selected cookbook. Each recipe is attached

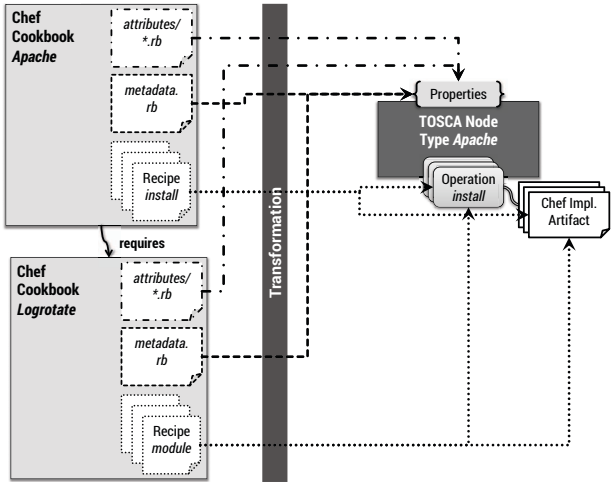

Figure 9. Coarse-grained transformation: dependencies packaged as IAs

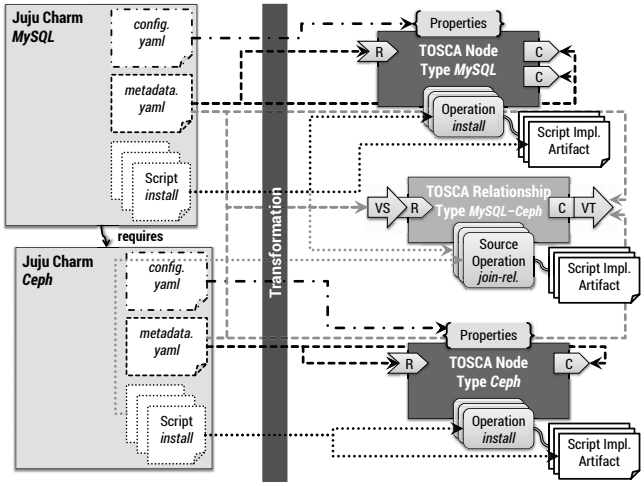

Figure 10. Transformation of Juju charms

to the generated node type as an IA that implements a certain operation as discussed in [23]. A single node type capability is generated for each cookbook consisting of the cookbook name. Each cookbook owns a `metadata.rb` file that contains information such as name, version, and maintainer of the cookbook. Moreover, all dependencies on other cookbooks are defined inside the `metadata.rb` file. For each dependency a corresponding requirement is generated. Relationship types are not required to be generated because Chef does not support modeling of relationships explicitly. Thus, a generic *depends on* relationship can be used to wire nodes derived from node types that were generated from cookbooks. Finally, the attributes definitions (`*.rb` files) stored in the `attributes` sub-directory of a cookbook is used to derive the node type properties. These are used to configure node templates derived from the node type, influencing the execution of the operations.

As an alternative, the coarse-grained transformation of Chef cookbooks is shown in Figure 9: node types are only generated for the selected cookbooks. The cookbooks that are dependencies are packaged inline as implementation artifacts. Consequently, the dependencies do not appear as separate node types and one node type may wrap multiple cookbooks. The main motivation for this is the fact that cookbooks and their dependencies may be very fine-grained. For instance, the *Apache HTTP server* cookbook[16] depends on the cookbooks *logrotate, iptables,* and *pacman*. These are operating system-level software packages that are not supposed to appear in the application topology (as individual nodes), which rather models more coarse-grained middleware and application components such as Web servers and databases. The coarse-grained transformation skips the creation of low-level node types, avoiding a pollution of topologies. We can also combine the two transformation approaches by specifying which cookbooks are too low-level to be exposed as separate node types. Consequently, these are transparently packaged as implementation artifacts without generating individual node types.

Other node-centric solutions and their artifacts such as Puppet [24] and CFEngine [25] work very similar to how Chef works. Consequently, the technical transformations discussed in this section can be transferred to be applied to these artifacts, too. In the following Section IV-B we discuss how Juju charms as environment-centric artifacts can be technically transformed to get TOSCA types.

[16]Apache cookbook: https://supermarket.getchef.com/cookbooks/apache2

B. Technical Transformation of Juju Charms

The transformation of environment-centric artifacts such as Juju charms is slightly more complex than what we discussed before for Chef cookbooks. This is because relationships have to be considered as separate entities with own operations and IAs to link different nodes in an application topology. In case of Juju, a separate node type is generated for each charm (Figure 10). The operations and their IAs are derived from the charm's lifecycle scripts. Each charm owns a `metadata.yaml` file describing its interfaces that it requires and provides. Based these interface descriptions, requirements and capabilities are attached to the generated node types. Furthermore, relationship types are generated for each requirement-capability pair because charms typically contain further lifecycle scripts that need to be executed when a relationship is established or changed (lifecycle of the relationship). Lifecycle scripts that need to be executed on the source node are mapped to source operations in the generated relationship type; scripts that are supposed to be executed on the target node result in target operations. Each relationship type defines the corresponding requirement (exposed by a source node) as valid source and the matching capability (exposed by a target node) as valid target. Consequently, the generated relationship type can be used to connect two nodes by matching the requirement of the source node with the capability of the target node. Similar to Chef's attributes the `config.yaml` file of each charm provides a description of its configuration options. These are used to attach corresponding property definitions to the generated types.

The discussed concepts to transform Juju charms to TOSCA types may be transferred to other environment-centric artifacts such as Amazon CloudFormation templates or OpenStack Heat templates. To confirm the added value of our transformation concepts, we present our evaluation of both technical transformations for Chef and Juju in the following Section V.

Table I. DERIVED TOSCA NODE TYPES AND RELATIONSHIP TYPES

Original Artifacts	Derived TOSCA Types	Attached Requirements	Attached Capabilities	Attached Operations
Web shop app. deployment script	Node Type: Web shop application	- Web server - PHP runtime - Web shop database	—	- deploy
Web shop database deployment script	Node Type: Web shop database	- MySQL database server	- Web shop database	- deploy - backup-db - restore-db
Web shop database connection script	Rel. Type: Web shop app. *connects to* database	- Web shop database *(valid source)*	- Web shop database *(valid target)*	- connect
Apache cookbook	Node Type: Apache HTTP server	- *Operating system*	- Web server	- deploy
PHP cookbook	Node Type: PHP module	- *Operating system* - Apache HTTP server *(derived from Web server)*	- PHP runtime	- deploy
MySQL charm	Node Type: MySQL server	- *Cloud infrastructure* - MySQL master server *(only for slave servers)*	- MySQL database server	- deploy
	Rel. Type: MySQL master *connects to* slave	- MySQL master server *(valid source)*	- MySQL database server *(valid target)*	- connect - disconnect
...				

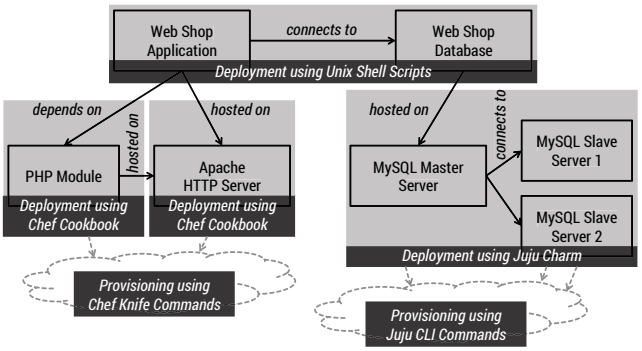

Figure 11. Web shop deployment automation using different kinds of artifacts

V. EVALUATION

In this section we present our evaluation of (i) the overarching transformation framework introduced in Section IV, (ii) the technical transformation of Chef cookbooks as discussed in Section IV-A, and (iii) the technical transformation of Juju charms (Section IV-B). We therefore implemented prototypes in Java as executable JAR files to perform the technical transformations presented before. The evaluation is based on the topology of the Web shop application outlined as motivating scenario in Section II.

A. Case Study & Performance Measurements

According to the method that we presented as part of our transformation framework (Section IV), the first step is the selection of DevOps artifacts to be used to deploy the Web shop application. To deploy the middleware required by the Web shop application, we reuse the Apache cookbook[17] and the PHP cookbook[18] as shown in Figure 11. For the database part we cannot just reuse the MySQL cookbook[19] because it does not support a distributed master/slave deployment out of the box. This is why we choose the MySQL charm[20] to deploy the MySQL database server in a distributed manner. The deployment of the application-specific parts (Web shop application, database, and the application-database connection) is implemented using custom Unix shell scripts. The underlying infrastructure is provisioned by associated tooling such as Chef knife and the Juju CLI tool. After retrieving all artifacts and resolving all dependencies we utilize our prototype implementations of the technical transformations to generate TOSCA node types and relationship types (Table I). In order to quantitatively evaluate the transformation we run the transformation for each artifact (Apache cookbook, PHP cookbook, and MySQL charm) *ten* times. The transformations were executed on a virtual machine (1 virtual CPU clocked at 2.8 GHz, 2 GB of memory) on top of the VirtualBox hypervisor, running a minimalist installation of the Ubuntu OS, version 14.04. The average transformation time and the standard deviation for each artifact based on our measurements is shown in Table II. Beside the actual model transformation the measured time includes the retrieval of the artifacts and all their dependencies from code repositories such as Git. Moreover, the packaging of the generated artifacts into Cloud Service Archives (CSARs) is also included in the measured time. The transformation of cookbooks should be faster because no relationship types are created. However, cookbook dependencies have to be retrieved separately because they are distributed across different repositories. This is why the transformation times are in the same order of magnitude. For the application-specific parts (Web shop application and database) we created additional types to cover the complete Web shop topology. In addition, two generic relationship types are available: *hosted on* and *depends on*. These are used to logically wire nodes in a topology if no specific implementation or operations are required. Finally, we use Winery [21] as a TOSCA modeling tool to build a topology template for the Web

[17]Apache cookbook: https://supermarket.getchef.com/cookbooks/apache2
[18]PHP cookbook: https://supermarket.getchef.com/cookbooks/php
[19]MySQL cookbook: https://supermarket.getchef.com/cookbooks/mysql
[20]MySQL charm: http://jujucharms.com/precise/mysql-46

Table II. TRANSFORMATION TIME MEASUREMENTS

	Apache cookbook	PHP cookbook	MySQL charm
Generated Node Types:	Apache HTTP server	PHP module	MySQL server
Generated Rel. Types:	—	—	MySQL master *connects to* MySQL slave
Average Transformation Time:	31.6 seconds	25.8 seconds	26.5 seconds
Standard Deviation:	0.7 seconds	0.5 seconds	0.2 seconds

shop application based on the generated TOSCA types. The complete topology template, including all type definitions and IAs is exported as a self-contained TOSCA-compliant Cloud Service Archive (CSAR). The Web shop CSAR can then be deployed using any TOSCA-compliant runtime environment, for instance, OpenTOSCA [22].

B. Standards-based Open-Source End-to-End Prototype

To prove the practical feasibility of the presented approach, we implemented the technical transformations presented in Section IV-A and Section IV-B as a prototype in Java (executable JAR files). This prototype can be used in conjunction with our open-source TOSCA ecosystem *OpenTOSCA*. This ecosystem consists of the (i) *OpenTOSCA runtime environment* [22], which can be used to run TOSCA-based applications, (ii) the modelling tool *Winery* [21], which provides an editor to model topologies, and (iii) *Vinothek* [26], a Web-based self-service portal for users to provision applications using the OpenTOSCA runtime environment. However, to create new node and relationship types using Winery, this must be done manually by (i) defining the types, management interfaces, operations, and properties, (ii) attaching the corresponding implementation artifacts, e.g., in the form of Juju charms, and (iii) linking the interfaces and operations with the implementation artifacts accordingly. This is a time-consuming, difficult, and error-prone task. The approach presented in this paper supports this task by generating the corresponding TOSCA elements fully automatically based on existing community artifacts such as Chef cookbooks. Thus, our approach significantly eases the development of new node and relationship types, which makes the whole development process of TOSCA-based applications very efficient. As a result, together with the existing ecosystem, the approach supports our open-source end-to-end TOSCA toolchain for modeling, deploying, and managing applications.

VI. RELATED WORK

Model transformations play a key role in different domains such as model-driven architectures and model-driven development [27]. According to [28] three major transformation strategies can be distinguished: (i) the *direct model manipulation* is a way to immediately modify an existing source model in order to transform it to the target model. (ii) Alternatively, an *intermediate representation* can be used to render models in a general-purpose markup language such as XML. (iii) A *transformation language* providing constructs to express and apply transformations may be used to drive such model transformations. In our work we focus on the intermediate representation of transformed models based TOSCA and rendered in XML.

In this paper we were mostly focusing on the modeling part of Cloud applications based on TOSCA. Related work [23], [29] presents approaches to deploy and manage Cloud applications that are modeled based on TOSCA and our transformation framework. There are several alternatives to TOSCA such as CloudML [30], Blueprints [31], and enterprise topology graphs [32]. Moreover, pattern-based deployment approaches [33] utilize a proprietary meta model based on Chef to transform patterns to deployable artifacts. Because TOSCA is an emerging standard and tooling support is improving as well, we decided to use TOSCA as an interoperable, intermediate meta model. Further orchestration approaches are originating in the DevOps community such as Amazon OpsWorks [34], Terraform[21], and DevOpSlang [35]. These can be utilized to orchestrate the transformed artifacts in a more seamless and interoperable manner. In addition to pure deployment automation approaches that are mainly targeting the level of infrastructure-as-a-service, different platform-as-a-service (PaaS) frameworks and solutions are emerging such as Heroku[22], IBM Bluemix[23], and Cloud Foundry[24]. These enable the packaging of middleware and application components on a higher level, mostly abstracting away infrastructure aspects such as virtual servers and networking. Consequently, artifacts that are packaged this way can be considered as environment-centric artifacts. Corresponding transformation concepts discussed in this paper may be transferred to these kinds of artifacts.

VII. CONCLUSION

The motivation of our work is based on the fact that a huge number and variety of reusable DevOps artifacts are shared as open-source software such as Chef cookbooks and Juju charms. However, these artifacts are usually bound to specific tools such as Chef, Juju, or Docker. This makes it hard to combine and orchestrate artifacts of different kinds in order to automate the deployment of Cloud applications consisting of different middleware and application components. In order to tackle this issue we decided to work on an automated transformation framework to generate TOSCA standard-based modeling artifacts from different kinds of existing DevOps artifacts. As a foundation for our work we presented an initial classification of DevOps artifacts, distinguishing between node-centric and environment-centric artifacts. We presented a generic transformation framework to be used to transform arbitrary DevOps artifacts into TOSCA node types and relationship types. These can then be used to create topology templates for Cloud applications. TOSCA enables the seamless and interoperable orchestration of arbitrary artifacts. Based on our transformation framework we implemented technical transformation methods for two different kinds of DevOps artifacts, namely Chef cookbooks and Juju charms. Finally, we evaluated our framework as well as the technical transformation implementations based on our motivating scenario. In terms of future work we plan to refine our proposed classification

[21]Terraform: http://www.terraform.io
[22]Heroku: https://www.heroku.com
[23]IBM Bluemix: http://bluemix.net
[24]Cloud Foundry: http://cloudfoundry.org

of DevOps artifacts by considering and evaluating further conceptual differences. Moreover, we plan to implement further technical transformations based on our framework to be able to extend the evaluation of our approach and to further verify that our framework is generic enough to deal with very different kinds of DevOps artifacts.

ACKNOWLEDGMENT

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

REFERENCES

[1] M. Hüttermann, *DevOps for Developers*. Apress, 2012.

[2] J. Humble and J. Molesky, "Why Enterprises Must Adopt Devops to Enable Continuous Delivery," *Cutter IT Journal*, vol. 24, 2011.

[3] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[4] S. Nelson-Smith, *Test-Driven Infrastructure with Chef*. O'Reilly Media, Inc., 2013.

[5] N. Sabharwal and M. Wadhwa, *Automation through Chef Opscode: A Hands-on Approach to Chef*. Apress, 2014.

[6] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," *National Institute of Standards and Technology*, 2011.

[7] B. Wilder, *Cloud Architecture Patterns*. O'Reilly Media, Inc., 2012.

[8] OASIS, "Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0, Committee Specification 01," 2013. [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html

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

[10] S. O'Mahony and F. Ferraro, "The Emergence of Governance in an Open Source Community," *Academy of Management Journal*, vol. 50, no. 5, pp. 1079–1106, 2007.

[11] S. Sharma, V. Sugumaran, and B. Rajagopalan, "A Framework for Creating Hybrid-Open Source Software Communities," *Information Systems Journal*, vol. 12, no. 1, pp. 7–25, 2002.

[12] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, 2007, pp. 275–287.

[13] M. J. Scheepers, "Virtualization and Containerization of Application Infrastructure: A Comparison," 2014.

[14] J. Fink, "Docker: a Software as a Service, Operating System-Level Virtualization Framework," *Code4Lib Journal*, vol. 25, 2014.

[15] S. Günther, M. Haupt, and M. Splieth, "Utilizing Internal Domain-Specific Languages for Deployment and Maintenance of IT Infrastructures," Very Large Business Applications Lab Magdeburg, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, Tech. Rep., 2010.

[16] K. Pepple, *Deploying OpenStack*. O'Reilly Media, 2011.

[17] OASIS, "Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0," 2013. [Online]. Available: http://docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.html

[18] ——, "Web Services Business Process Execution Language (BPEL) Version 2.0," 2007.

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

[20] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, "BPMN4TOSCA: A Domain-Specific Language to Model Management Plans for Composite Applications," in *Business Process Model and Notation*, ser. Lecture Notes in Business Information Processing, J. Mendling and M. Weidlich, Eds., vol. 125. Springer Berlin Heidelberg, 2012, pp. 38–52.

[21] 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*, ser. LNCS, vol. 8274. Springer Berlin Heidelberg, 2013, pp. 702–706.

[22] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "OpenTOSCA – a runtime for TOSCA-based cloud applications," in *Proceedings of the 11th International Conference on Service-Oriented Computing*, ser. LNCS. Springer, 2013.

[23] J. Wettinger, M. Behrendt, T. Binz, U. Breitenbücher, G. Breiter, F. Leymann, S. Moser, I. Schwertle, and T. Spatzier, "Integrating Configuration Management with Model-Driven Cloud Management Based on TOSCA," in *Proceedings of the 3rd International Conference on Cloud Computing and Services Science*. SciTePress, 2013.

[24] J. Loope, *Managing Infrastructure with Puppet*. O'Reilly Media, Inc., 2011.

[25] D. Zamboni, *Learning CFEngine 3: Automated System Administration for Sites of Any Size*. O'Reilly Media, Inc., 2012.

[26] U. Breitenbücher, T. Binz, O. Kopp, and F. Leymann, "Vinothek - A Self-Service Portal for TOSCA," in *ZEUS*. CEUR-WS.org, 2014, pp. 69–72.

[27] A. Kleppe, J. Warmer, and W. Bast, "MDA Explained - The Model Driven Architecture: Practice and Promise," 2003.

[28] S. Sendall and W. Kozaczynski, "Model Transformation the Heart and Soul of Model-Driven Software Development," Tech. Rep., 2003.

[29] J. Wettinger, T. Binz, U. Breitenbücher, O. Kopp, F. Leymann, and M. Zimmermann, "Unified Invocation of Scripts and Services for Provisioning, Deployment, and Management of Cloud Applications Based on TOSCA," in *Proceedings of the 4th International Conference on Cloud Computing and Services Science*. SciTePress, 2014.

[30] A. Rossini, N. Nikolov, D. Romero, J. Domaschka, K. Kritikos, T. Kirkham, and A. Solberg, "CloudML Implementation Documentation," PaaSage project deliverable, 2014.

[31] M. Papazoglou and W. van den Heuvel, "Blueprinting the Cloud," *Internet Computing, IEEE*, vol. 15, no. 6, pp. 74–79, 2011.

[32] T. Binz, C. Fehling, F. Leymann, A. Nowak, and D. Schumm, "Formalizing the Cloud through Enterprise Topology Graphs," in *Proceedings of the IEEE International Conference on Cloud Computing*. IEEE Computer Society Conference Publishing Services, 2012.

[33] H. Lu, M. Shtern, B. Simmons, M. Smit, and M. Litoiu, "Pattern-based Deployment Service for Next Generation Clouds," 2013.

[34] T. Rosner, *Learning AWS OpsWorks*. Packt Publishing Ltd, 2013.

[35] J. Wettinger, U. Breitenbücher, and F. Leymann, "DevOpSlang - Bridging the Gap Between Development and Operations," in *Proceedings of the 3rd European Conference on Service-Oriented and Cloud Computing (ESOCC)*, 2014.