Integrating Configuration Management with Model-driven Cloud Management based on TOSCA

Johannes Wettinger[1], Michael Behrendt[2], Tobias Binz[1], Uwe Breitenbücher[1], Gerd Breiter[2], Frank Leymann[1], Simon Moser[2], Isabell Schwertle[2] and Thomas Spatzier[2]
[1]*Institute of Architecture of Application Systems, University of Stuttgart, Universitätsstraße 38, Stuttgart, Germany*
[2]*IBM Germany Research & Development GmbH, Schönaicher Straße 220, Böblingen, Germany*

Keywords: Model-driven Management, Configuration Management, Service Management, Cloud Services, DevOps.

Abstract: The paradigm of Cloud computing introduces new approaches to manage IT services going beyond concepts originating in traditional IT service management. The main goal is to automate the whole management of services to reduce costs and to make management tasks less error-prone. Two different service management paradigms are used in practice: configuration management and model-driven Cloud management. The latter one aims to be a holistic management approach for services in the Cloud. However, both management paradigms are originating in different backgrounds, thus model-driven Cloud management does not cover all aspects of configuration management that are key for Cloud services. This paper presents approaches for integrating configuration management with model-driven Cloud management and how they can be realized based on the OASIS Topology and Orchestration Specification for Cloud Applications and Chef, a popular configuration management tool. These approaches enable the creation of holistic and highly portable service models.

1 INTRODUCTION

Today, many providers in the field of Cloud computing (Leymann, 2011; Mell and Grance, 2011) attract people to create highly scalable applications and services using the providers' proprietary offerings. These can be infrastructure offerings such as Amazon Web Services[1] or platform offerings such as Google App Engine.[2] Many people get attracted by these offerings because low requirements are put on developers when starting to create services and applications. The corresponding meta models are simple and the provided tooling is easy to use. This results in a very flat learning curve.

However, when services get more complex, their management becomes hard (Günther et al., 2010). It may be even impossible to move the service to another Cloud provider because of missing standards that ensure portability. The results are vendor lock-in and bad manageability. Thus, the following main contributions of this paper are focused on improving the manageability of services in the Cloud without abandon portability: (1) the *direct integration* of lower-level configuration definitions such as scripts with higher-level service models. (2) The *transparent integration* of configuration definitions with higher-level service models, so the type of the configuration definitions is completely transparent to the underlying runtime environment that is responsible for the actual service deployment and management. (3) *Combining the direct and the transparent integration approach* to bring together the strength of both of them and to reduce their individual shortcomings. To maintain portability, the actual concepts described in this paper do not stick to specific Cloud offerings.

The remainder of this paper is structured as follows: Section 2 further refines the actual problem that is the motivation for this paper. The Topology and Orchestration Specification for Cloud Applications (OASIS, 2012), hereafter referred to as TOSCA, is presented as a representative to realize model-driven Cloud management in Section 3; in addition, Chef is introduced as an example for configuration management tooling. We describe the main contributions of this paper in Section 4, namely the integration concepts. Section 5 shortly describes the implementation that was created to evaluate the integration concepts and to show their

[1]http://aws.amazon.com
[2]http://developers.google.com/appengine

Wettinger J., Behrendt M., Binz T., Breitenbücher U., Breiter G., Leymann F., Moser S., Schwertle I. and Spatzier T..
Integrating Configuration Management with Model-driven Cloud Management based on TOSCA.
DOI: 10.5220/0004376204370446
In *Proceedings of the 3rd International Conference on Cloud Computing and Services Science* (CLOSER-2013), pages 437-446
ISBN: 978-989-8565-52-5
Copyright © 2013 SCITEPRESS (Science and Technology Publications, Lda.)

realization. Related work is presented in Section 6. Section 7 concludes this paper and describes limitations and future work.

2 PROBLEM STATEMENT

As of today, the so called DevOps methodologies (Humble and Farley, 2010; Humble and Molesky, 2011; Shamow, 2011) represent the leading paradigm for efficiently managing services and applications in a highly automated manner. The original goal of these methodologies is to bring together developers and the operations personnel. This goal is mainly achieved by automating all the deployment and management tasks. The actual automation is realized by configuration management tooling (Günther et al., 2010; Nelson-Smith, 2011; Delaet et al., 2010) such as Chef[3] or Puppet.[4] As a result, the corresponding management processes are much more reliable and cost less in contrast to performing these processes manually. Consequently, it is much easier to deploy and re-deploy services into different environments such as development, test, and production. The philosophy behind the DevOps movement is to bring agile methodologies into the world of IT infrastructure and service management (Humble and Farley, 2010). This is achieved by implementing the concept of "Infrastructure as Code" using configuration management tooling. The concept is based on the assumption that almost any action on the infrastructure management level can be automated programmatically (Smith, 2011). Consequently, products such as Chef or Puppet that implement this concept provide a scripting language or a domain-specific language to create and maintain platform-independent configuration definitions for deploying and managing an IT service (Günther et al., 2010). Of course, configuration management is not limited to implementing DevOps scenarios. The concept of Infrastructure as Code targets automation and cost reduction of service management in general. These are essential parts of the Cloud computing paradigm (Leymann, 2009; Vaquero et al., 2008).

However, this approach is not appropriate for managing complex services. Managing a large and sophisticated service topology consisting of different machines using plain configuration management tooling can get cumbersome and time-consuming. Even the infrastructure for providing a usual Web application can become complex very quickly: there are several technologies required to realize load balancing, caching, and a full-text index. To specify such an infrastructure a lot of "infrastructure code" is being created. As a result, it is hard to keep the code structure clean because there is no holistic service model defined including the service topology that prescribes the structure of a service that can be instantiated. Every single change of the "infrastructure code" becomes a risk because it is hard to estimate the consequences of that particular change (Nelson-Smith, 2011).

Thus, the paradigm of model-driven Cloud management provides a more holistic approach for managing a service, but it does not completely replace the DevOps approach. The model-driven approach does not target many aspects of DevOps methodologies because both paradigms are originating in different backgrounds. As an example, model-driven Cloud management does not directly specify how to perform the lower-level actions on a virtual machine to install and configure a particular software component. This is why the main contributions of this paper described in Section 4 are focused on bringing together the strengths of both worlds by integrating configuration management with model-driven Cloud management to minimize the shortcomings of the individual approaches.

3 FUNDAMENTALS

The following Section 3.1 outlines the fundamentals of TOSCA because the concepts presented in this paper are mainly described using the emerging standard to realize model-driven Cloud management. Several examples that are given in this paper are based on Chef, a popular configuration management product. This is why we shortly introduce Chef's basic concepts and terminology in Section 3.2.

3.1 Topology and Orchestration Specification for Cloud Applications

From a perspective of plain configuration management, a node is always a physical or virtual machine. However, the actual topology of an infrastructure to run a particular service is more than a set of virtual machines connected to one another. Software components are running on each machine. These components can be connected to components running on other machines, such as an application that connects to a database. Figure 1 and Figure 2 outline two sample service topologies for a Web application. They illustrate a much broader understanding of nodes and relationships. This understanding leads to a higher-level model-driven approach for defining the service topology. Such a

[3]http://www.opscode.com/chef
[4]http://www.puppetlabs.com

model is not limited to expressing relationships between different software components hosted on virtual machines as shown in Figure 1. Nodes can also be services offered by a Cloud provider as outlined in Figure 2. Furthermore, additional aspects of the service topology such as networking can be modeled.

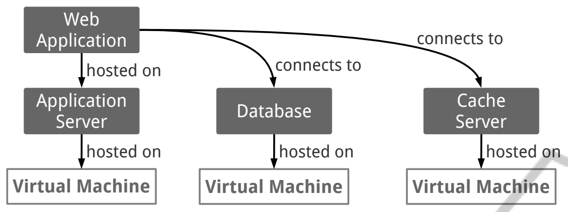


Figure 1: Service topology based on virtual machines.

These kinds of service topologies can be modeled using TOSCA. The result is a self-contained, portable, and executable service model that can be used to deploy and manage service instances in the Cloud (Binz et al., 2012). Before TOSCA was introduced, research focused on migrating services from one Cloud environment to another without taking the portability of management aspects into account (Leymann et al., 2011; Binz et al., 2011).

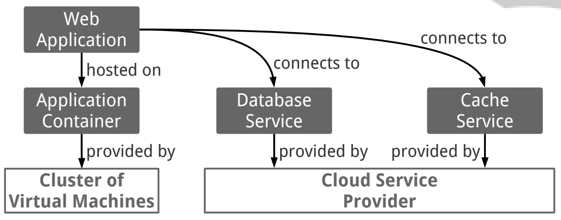


Figure 2: Service topology based on existing services.

The meta model of TOSCA is technically specified by an XML schema definition. It prescribes the structure of a service template. The key parts to describe a service topology inside such a service template are: (1) node types that represent components to be used inside the topology template. A single node type such as "database server" can be instantiated once or several times as a node template inside the topology template. (2) Relationship types such as "hosted on" can be instantiated as relationship templates inside the topology template to express any kind of relation between two particular node templates. (3) The topology template defines the actual topological structure of an IT service such as shown in Figure 1 and Figure 2; it consists of node templates and relationship templates.

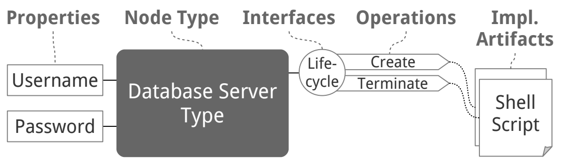


Figure 3: Example for a TOSCA node type.

Figure 3 presents an example for a node type that defines a database server component. A node type can own arbitrary property definitions such as "username" and "password" in case of a database server. These properties are explicitly defined and attached to a particular node type. Node templates can include concrete values for these properties. Furthermore, a node type can own interfaces. A particular interface provides operations, which define the possibilities of interaction of a node of the given node type. We assume that in future versions of TOSCA an arbitrary node type owns a "lifecycle" interface providing at least two operations: "create" and "terminate" the node of a particular node type. As yet, the node type definition is abstract and does not say anything about how an operation is implemented. Thus, one or more concrete implementation artifacts can be linked to an operation. Such an implementation artifact is created by defining an artifact template inside the service template. Then, the artifact template can be attached to at least one operation as an implementation artifact. For instance, an implementation for the "create" operation can be a Unix shell script to install the database server. In addition, another script can be attached to the same operation that performs the equivalent actions on Windows-based systems. Attaching several implementation artifacts to a particular operation improves the portability of the service template because the operation can be performed on different platforms. The definition of relationship types is similar to the definition of node types (OASIS, 2012).

TOSCA is not only focused on specifying the service topology. Management plans can be defined to support the whole lifecycle of an application, including deployment, maintenance, and termination (Breitenbücher et al., 2013). These plans can be defined using languages such as the Business Process Model and Notation (OMG, 2011) or the Web Services Business Process Execution Language (OASIS, 2007).

All the files that are referenced inside the service template such as scripts, binaries, and plans get packed together with the service template into a Cloud service archive (CSAR). The CSAR is completely self-contained, that is, it contains everything to deploy and manage the Cloud service that is specified by the included service template. The software that is able to process CSARs is referred to as a TOSCA runtime environment.

In terms of Cloud management, there are products and approaches available beside TOSCA such as VMware's virtualization and Cloud management,[5]

[5]http://www.vmware.com/solutions

OpenNebula (Milojičić et al., 2011), OpenStack,[6] and others (Han et al., 2009). But these are primarily focused on the lower-level infrastructure. Their goal is to simplify the management of virtualized resources such as virtual machines or storage resources. This paper is referring to TOSCA because of its holistic and portable approach to implement model-driven Cloud management. The emerging standard is supported by a number of prominent companies in the industry such as IBM, SAP, and Hewlett-Packard as well as research institutions.

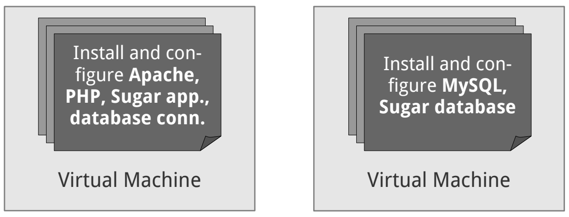
3.2 Chef

Chef is a popular configuration management product. Its open source variant is publicly available and can be used for free. Configuration definitions in Chef are called recipes. These are basically scripts written in a domain-specific language to express the target state of a system (Günther et al., 2010). One or more recipes are bundled in a cookbook. In addition to cookbooks, roles can be defined. A particular node can have zero or more roles. Recipes can be associated with one or more roles. This mechanism allows to link recipes to nodes without assigning them directly. The Chef server stores all the cookbooks and role definitions. In addition, the server component manages a "run list" for each node that is registered. The Chef client runs on each of these nodes to connect to the Chef server. A particular run list assigns recipes and roles to a node. To execute the recipes, there is no Chef server required: in addition to the Chef client/server mode, there is a Chef solo mode available. Using this mode, recipes can be directly executed using the Chef client without a Chef server (Nelson-Smith, 2011).

Puppet's architecture is similar to the architecture of Chef, so the concepts and examples we describe can be adapted to Puppet. The Puppet master is the equivalent of the Chef server; the Puppet agent runs on each node and is thus the equivalent of the Chef client. Puppet's manifests, written using a domain-specific language and bundled in catalogs, are similar to Chef's recipes bundled in cookbooks (Loope, 2011; Nelson-Smith, 2011). Further, there may be even more configuration management tools owning a similar architecture such as CFEngine.[7] These can be used to realize the concepts presented in this paper, too.

4 INTEGRATION CONCEPTS

The main contributions of this paper are outlined in this section: the goal is to bring together the strengths of configuration management and model-driven Cloud management in order to minimize their individual shortcomings. TOSCA is used to realize model-driven Cloud management by specifying the service template on a higher level. Such a specification includes the topology template, the node types, and the relationship types. In addition, lower-level implementation artifacts need to be attached to node type operations and relationship type operations in order to realize the operation's functionality such as installing and configuring a particular software component. Model-driven Cloud management does not focus on these lower-level aspects. The straightforward approach is to implement shell scripts to install and configure software components. Because shell scripts are intended to be used to perform simple tasks on a specific platform, using a platform-independent scripting language such as Python or Ruby would be a better choice to create portable artifacts. Otherwise, many platform-specific implementation artifacts would have to be attached to a single operation. However, another two difficulties still remain: (1) it is hard to maintain the scripts because these scripting languages are general-purpose languages and do not directly support the domain of software installation and configuration. (2) It is hard to design the scripts in a way so that they can be ported to additional platforms without much effort.


Install and configure **Apache, PHP, Sugar app., database conn.**
Virtual Machine
Install and configure **MySQL, Sugar database**
Virtual Machine
Figure 4: Deployment using configuration management.

This is why approaches originating in configuration management are a great fit to further enhance the value of model-driven Cloud management. As an example, Figure 4 presents how Sugar[8] can be deployed as a service in the Cloud using plain configuration management tooling. Sugar is a Web-based customer relationship management system; its "community edition" is publicly available as open source software. For the actual deployment of Sugar, there are two virtual machines provisioned. On the one machine, the Apache Web server, the PHP module, and the Sugar application get installed and configured by automatically running artifacts. The MySQL server and the Sugar database get installed and configured on the other machine. To finish the deployment, the application gets connected to the database.

[6]http://www.openstack.org
[7]http://www.cfengine.com
[8]http://www.sugarcrm.com

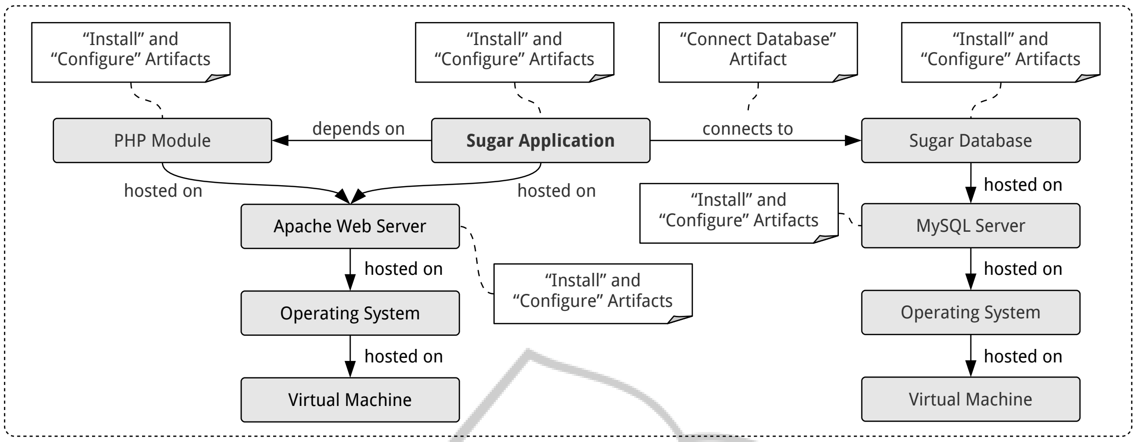


Figure 5: Service topology for deploying and managing Sugar in the Cloud.

All these actions are performed by processing corresponding configuration definitions. The artifacts are portable, configurable, and customizable (Günther et al., 2010). Using the higher-level service model based on TOSCA including the service topology model, the relation of these configuration definitions to one another can be defined. Figure 5 outlines the topology model for the Sugar Cloud service with artifacts attached including configuration definitions to install and configure the software components that are running on top of the virtual machines' operating system.

In addition to the advantages of using configuration definitions regarding portability and maintainability, configuration management products offer tools and mechanisms to ease the management of service topologies. As an example, additional configuration definitions can be assigned to nodes and roles owned by at least one node in the topology. Typical examples for activities realized by additional configuration definitions are updating an application, doing a database backup, or customizing the configuration of a deployed application.

For the popular configuration management tools such as Chef or Puppet, there are configuration definitions already available to install, configure, and manage many software components such as cookbooks for the Apache Web server and the MySQL database server. Consequently, they can be easily reused and combined by creating a service topology model with configuration definitions attached.

The following sections discuss two different approaches to include configuration definitions into a service topology model. These are the direct integration (Section 4.1) and the transparent integration (Section 4.2). Furthermore, Section 4.3 outlines how to combine these two approaches to benefit from the strengths of both.

4.1 Direct Integration

The direct way of embedding configuration definitions into a service template is to define an artifact type, which fits the structure of the artifacts including configuration definitions that are created based on a particular configuration management tool. For instance, a Chef artifact type needs to be defined to embed Chef cookbooks into a topology model. We assume that in future versions of TOSCA a set of standard types of implementation artifacts such as "script artifact" will be available. Implementation artifacts of these types can be processed by an arbitrary TOSCA runtime environment. However, arbitrary types of implementation artifacts can be defined by creating an appropriate XML schema definition. A TOSCA runtime environment that processes a service template including custom implementation artifacts then has to understand the corresponding artifact type.

Artifact templates are defined inside a TOSCA service template and can then be referenced as an implementation artifact for a particular operation. The following XML snippet represents an example for a Chef-specific artifact template that installs and configures a MySQL database server:

```
<ArtifactTemplate id="mysql-chef-artifact"
                  type="chef:ChefArtifact">
 <Properties>
  <chef:ChefArtifactProperties
        xmlns:chef="http://.../Chef"
        xmlns="http://.../Chef">
   <Cookbooks>
    <Cookbook name="build-essential"
     location="files/build-essential.zip"/>
    <Cookbook name="openssl"
     location="files/openssl.zip"/>
    <Cookbook name="mysql"
     location="files/mysql.zip"/>
   </Cookbooks>
```

```
15     <Roles>
16      <Role name="db-server"
17       location="files/db-server.json"/>
18     </Roles>
19     <Mappings>
20      <PropertyMapping mode="input"
21       propertyPath="/RootPassword"
22       cookbookAttribute="mysql/root_pw"/>
23     </Mappings>
24     <RunList>
25      <Include>
26       <RunListEntry roleName="db-server"/>
27      </Include>
28     </RunList>
29    </chef:ChefArtifactProperties>
30   </Properties>
31   <ArtifactReferences>
32    <ArtifactReference reference="files">
33     <Include pattern="build-essential.zip"/>
34     <Include pattern="openssl.zip"/>
35     <Include pattern="mysql.zip"/>
36     <Include pattern="db-server.json"/>
37    </ArtifactReference>
38   </ArtifactReferences>
39  </ArtifactTemplate>
```

This example outlines the structure of an artifact template including the generic parts that are common for each artifact template as well as the Chef-specific parts. The *ArtifactTemplate* element (line 1) itself belongs to the generic parts and owns two attributes: the *id* attribute specifies a unique identifier for this particular artifact template inside the service template. The *type* attribute specifies the artifact type. The *ArtifactReferences* element (lines 31–38) including all its child elements is a generic mechanism for artifact templates to point to the files included in the CSAR that are necessary to process the artifact. In case of Chef artifact templates, the cookbooks and role definitions get referenced in this section. The content of the *Properties* element (lines 3–30) is Chef-specific. The structure of the *ChefArtifactProperties* element (lines 4–29) including all its child elements can be defined using an XML schema definition including the following parts:

Cookbooks. Each cookbook that is required to realize a particular artifact template is referenced using a *Cookbook* element (lines 8–13). This includes cookbooks that are directly referenced inside the *RunList* section as well as dependencies of these.

Roles. Each role definition that is required to realize a particular artifact template is referenced using a *Role* element (lines 16–17). This includes roles that are directly referenced inside the *RunList* section as well as dependencies of these.

Mappings. Values of node type properties can be mapped to cookbook attributes using a *PropertyMapping* element (line 20–22). The *propertyPath* attribute contains an XPath expression that points to a particular property. To refer to the corresponding cookbook attribute, the *cookbookAttribute* attribute is used. The same can be done for relationship type properties. Due to the fact that a relationship owns a source node and a target node, their property values can be mapped using a *SourcePropertyMapping* or a *TargetPropertyMapping* element. If the particular artifact template implements a script operation, input and output parameters can be mapped using an *InputParameterMapping* or an *OutputParameterMapping* element.

RunList. At its core, a Chef artifact template defines which recipes and roles have to be part of the corresponding run list. The *RunList* element (lines 24–28) can contain two child elements: the *Include* element consists of at least one *RunListEntry* element that points to recipes and roles. These have to be part of the run list in the given order. The *RunListEntry* child elements of the *Exclude* element denote recipes and roles that have to be explicitly excluded from the run list. A *RunListEntry* element can either point to a role as shown before or it can directly refer to a particular recipe:

```
<RunListEntry cookbookName="mysql"
              recipeName="install"/>
```

As an additional remark regarding the definition of property mappings, the *mode* attribute specifies the mapping direction: “input” means that the property value is mapped to the cookbook attribute before the Chef recipes are executed, “output” maps the cookbook attribute to the property after the Chef recipes were executed, and “input-output” combines both kinds of mapping.

This is how Chef artifact templates can be realized using the direct integration approach. The TOSCA runtime environment has to understand the content of the *ChefArtifactProperties* element to perform the corresponding actions. However, it is completely up to the TOSCA runtime environment how to perform these actions. Two options are: the Chef server’s REST API and Chef’s command line interface. Furthermore, the artifact template does not prescribe whether a Chef server is required to realize the artifact. Alternatively, the TOSCA runtime environment can just use a Secure Shell connection (SSH) to push the cookbooks and role definitions to the nodes and then use Chef solo to execute them.

4.2 Transparent Integration

Configuration definitions can also be embedded into a service topology model in a transparent way using the standard artifact type “script artifact.” Consequently,

SCIENCE AND TECHNOLOGY PUBLICATIONS

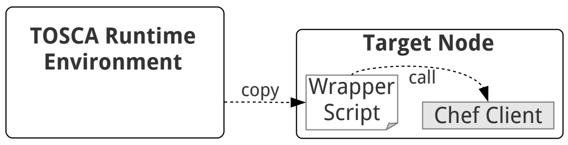


Figure 6: Transparent integration using a wrapper script.

the TOSCA runtime environment does not have to understand implementation artifacts of custom artifact types such as Chef-specific artifacts. Of course, we assume that an arbitrary TOSCA runtime environment can process artifacts of type "script artifact" as mentioned in Section 4.1. In practice, this approach of integrating configuration management can be realized by creating "wrapper scripts" that call the configuration management tool with corresponding parameters to point to the configuration definitions. Figure 6 shows how wrapper scripts are used to perform operations on a target node using Chef: first, the TOSCA runtime environment copies the corresponding wrapper script to the target node and then triggers its execution; additional files that are required for the execution of the wrapper script are copied to the target node, too. Second, the wrapper script calls the Chef client. These wrapper scripts can be attached to the topology model using implementation artifacts of type "script artifact" in TOSCA. However, some constraints such as the configuration of the agent that processes the configuration definitions are hard-wired inside the wrapper scripts. The result is a loss of flexibility to process configuration definitions because the TOSCA runtime environment does not understand what is happening inside a wrapper script. It cannot control how the wrapper script performs its work. In addition, it is hard to make the wrapper scripts portable. The following XML snippet presents an example for a wrapper artifact template of type "script artifact:"

```
 1 <ArtifactTemplate id="mysql-wrapper"
 2                   type="af:ScriptArtifact">
 3  <Properties>
 4   <af:ScriptArtifactProperties
 5    xmlns:af="http://.../StandardArtifacts"
 6    xmlns="http://.../StandardArtifacts">
 7    <ScriptLanguage>sh</ScriptLanguage>
 8    <PrimaryScript>
 9     scripts/mysql-install.sh
10    </PrimaryScript>
11   </af:ScriptArtifactProperties>
12  </Properties>
13  <ArtifactReferences>
14   <ArtifactReference reference="files">
15    <Include pattern="build-essential.zip"/>
16    <Include pattern="openssl.zip"/>
17    <Include pattern="mysql.zip"/>
18    <Include pattern="db-server.json"/>
19   </ArtifactReference>
20   <ArtifactReference reference="scripts">
21    <Include pattern="mysql-install.sh"/>
22   </ArtifactReference>
23  </ArtifactReferences>
24 </ArtifactTemplate>
```

This artifact template implements the very same operation as the Chef-specific artifact template presented before (Section 4.1). The content of the *ScriptArtifactProperties* element (lines 4–11) is limited to some generic meta data regarding the corresponding script. All the Chef-related information such as the mappings and the run list entries are hidden inside the wrapper script "mysql-install.sh." As a result, the Chef specifics of the artifact template are completely transparent to the TOSCA runtime environment: the runtime implementation does not have to know anything about Chef. However, all the actions necessary to realize the artifact are hard coded inside the wrapper script: the TOSCA runtime environment does not have much flexibility in performing the corresponding actions. Whereas for the direct integration approach the mapping of properties and parameters is explicitly defined inside the artifact template, these mappings are hidden inside the wrapper script for the transparent integration. The TOSCA technical committee did not establish a standardized convention yet how scripts can access properties of nodes and relationships. One possible convention for TOSCA runtime environments is to expose all properties as system environment variables to the script. As an example, the wrapper script can access the "$RootPassword" variable to get the value of the node type property "RootPassword." All the knowledge how to map these property values to the configuration management tool is part of the logic implemented inside the script.

4.3 Combined Integration

The transparent integration approach is not meant to be used solely because wrapper scripts are not intended to be created manually. The goal of the combined integration is to bring together the strengths of both integration approaches by including two alternative artifact templates for each operation as outlined in Figure 7: one that follows the direct integration approach and another one that follows the transparent integration approach. However, the artifact templates that are based on the transparent integration approach including wrapper scripts are not created manually. They are programmatically generated based on the artifact templates that follow the direct integration approach. Then, the TOSCA runtime environment can decide whether to process the standard artifacts or to process the custom artifacts if the corresponding artifact type is understandable for the particular TOSCA runtime environment. The actual decision of the TOSCA runtime

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Table 1: Comparison of direct, transparent, and combined integration.

	Direct Int.	Transparent Int.	Combined Int.
TOSCA runtime environment can process configuration definitions directly	+	−	+
TOSCA runtime environment does not have to understand custom artifact types	−	+	+
High degree of portability	−	−	+

environment which artifact template to process can be made based on certain compliance rules. Because wrapper scripts are "black boxes" for the TOSCA runtime environment and could possibly contain harmful content, the TOSCA runtime environment may always prefer artifact templates that follow the direct integration approach.

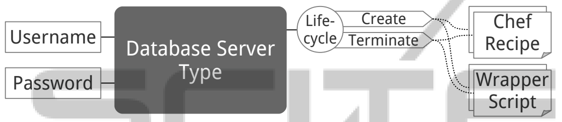


Figure 7: Combined integration results in two alternative implementation artifacts for each operation.

Beside the approach of programmatically generating individual wrapper scripts as mentioned before, a generic wrapper script can be built that can be parameterized and thus used to process arbitrary Chef recipes. This approach of building generic wrapper scripts can be extended by establishing a public community-driven repository to share generic wrapper scripts. These scripts enable an arbitrary TOSCA runtime environment that only understands implementation artifacts of a specific type such as "script artifact" to process any other type of artifacts if there is a corresponding wrapper script available. As a result, the creator of a CSAR can use his preferred configuration management tool or scripting language to implement node type operations and then include parameterized wrapper scripts to make the CSAR portable. Consequently, this approach enables separation of concerns because the CSAR creator can stick to his domain and include wrapper scripts to improve the portability of a CSAR.

Table 1 summarizes benefits and drawbacks of the integration approaches described in this section. It underlines that the combined integration approach brings together the strengths of both the direct and the transparent integration approach leading to a high degree of portability. This is achieved either by generating individual wrapper scripts for each operation or by using generic wrapper scripts. Both the direct and the transparent integration approach alone do not lead to a high degree of portability because either the TOSCA runtime environment has to understand custom artifact types or some constraints are hard-wired inside the wrapper scripts.

5 EVALUATION

To evaluate the concepts presented in Section 4 and to show that they are applicable in practice, we built two CSARs that realize the Sugar service topology presented in Figure 5. The first CSAR is based on the direct integration approach including one Chef-specific artifact for each operation. The second CSAR realizes the combined integration approach by additionally attaching equivalent implementation artifacts of type "script artifact" to each operation, including individually generated wrapper scripts. Both CSARs are completely self-contained including all Chef cookbooks and role definitions.

We did not include any management plans into the CSARs. As a result, a TOSCA runtime environment has to understand the declarative model that is defined by the topology template. In contrast to an imperative model, a declarative model does not include a "build plan" that explicitly specifies the steps required to be performed to deploy a service instance. To enable the declarative approach of deploying a CSAR, we assume that the lifecycle operation "create" is part of each node type. Implementation artifacts to install and configure the corresponding software component are attached to these "create" operations. The deployment of a declarative model is implemented by traversing the topology template and calling the "create" operations. All "create" operations of the middleware stack inside the service topology such as the Apache Web server and the MySQL database server are implemented using cookbooks that are publicly shared by the Chef community.[9] The only cookbook that we implemented manually is the Sugar cookbook. It includes recipes to install and configure the application, to set the Sugar database up, and to connect the application to the database.

In addition, we enabled the deployment of the CSARs in three different ways: (1) we implemented a lightweight TOSCA runtime environment that can process Chef-specific implementation artifacts attached to the "create" operations. These artifacts get processed by Chef solo. This is realized by copying the Chef cookbooks and role definitions to the target node using SSH and then calling the Chef client in Chef solo

[9]http://community.opscode.com

mode with the corresponding parameters. (2) We extended the current TOSCA runtime environment of the IBM SmartCloud Orchestrator[10] to process Chef-specific artifacts either using an existing Chef server or by Chef solo. The Chef solo variant is implemented similarly to the implementation outlined before using SSH. In case of the Chef server variant the Chef server's REST API is used to upload the cookbooks and role definitions and to assign recipes and roles to the nodes that are registered at the Chef server. Then, the Chef client running on each node retrieves and processes the assigned cookbooks and role definitions. In addition, we utilized the Chef server to perform simple management tasks regarding the deployed service instance. (3) A student team developed another lightweight TOSCA runtime environment during the "Extreme Blue 2012" summer internship at IBM. It understands implementation artifacts of type "script artifact" only. However, it is able to process the CSAR that was created based on the combined integration approach without any extension or modification. It basically executes the wrapper scripts, which then call the Chef client with the corresponding parameters.

6 RELATED WORK

Because large and complex service topologies are hard to manage using plain configuration management, the DevOps community establishes additional higher-level tooling such as Marionette Collective (Turnbull and McCune, 2011; Loope, 2011) and Spiceweasel.[11] These tools provide more convenience to manage clusters and collections of nodes. Marionette Collective provides a framework that can be used in conjunction with both Puppet and Chef. Spiceweasel is a Chef-based command line tool to ease the management of a set of nodes, but it is much simpler and more limited compared to Marionette Collective. The configuration of a collection of nodes can be described using a document written in "JavaScript Object Notation" (JSON). Then, several Chef commands are being generated based on this document to perform the actual deployment. However, these tools do not lift configuration management tooling to the level of model-driven management because they do not introduce a holistic meta model for managing complex service topologies as TOSCA does.

Similar to Spiceweasel, Amazon Web Services' CloudFormation[12] is a service offering to manage a set of resources in the Amazon Cloud. CloudFormation templates are written in JSON and can be used to instantiate and configure a set of resources such as virtual machines and object stores. In contrast to a TOSCA service template, a CloudFormation template does not include a holistic service topology model including relationships between nodes. Further, a CloudFormation template can only refer to resources that are available inside the Amazon Cloud.

Canonical Juju[13] is a service orchestration framework for sharing and reusing DevOps best practices in the form of self-contained units. These shareable and reusable units are called "charms." A single charm represents a service model that can be instantiated. In addition, relations can be established between service instances, so Juju's meta model has similarities to TOSCA. As a result, Juju enables model-driven Cloud management. However, there are limitations when it comes to portability because Juju is strongly focused on the ecosystem around the Ubuntu Linux distribution. The scripts included in a charm representing the implementation artifacts are usually shell scripts or Python scripts designed to be executed on a Ubuntu Linux system only.

7 SUMMARY AND OUTLOOK

We presented concepts to integrate configuration management with model-driven Cloud management. The goal was to bring together the strengths of both service management paradigms. This was achieved based on TOSCA, an emerging standard to realize model-driven Cloud management. Different concepts were described how to include configuration definitions into a CSAR: the direct integration of artifacts including configuration definitions requires the TOSCA runtime environment to understand the corresponding artifact type. However, there are no hard-wired constraints included in the artifacts, so the TOSCA runtime environment has a high degree of flexibility in processing the artifacts. The transparent integration approach can be used to enable the processing of configuration definitions in a TOSCA runtime environment that does not understand the corresponding artifact type. Both integration approaches can be combined to make a CSAR highly portable. These integration concepts are following the design principles of TOSCA and their implementation does not need any extension of TOSCA itself. The integration of configuration definitions is the foundation for creating portable CSARs.

There are many configuration definitions publicly

[10]http://ibm.co/CPandO

[11]http://wiki.opscode.com/display/chef/Spiceweasel

[12]http://aws.amazon.com/cloudformation

[13]http://juju.ubuntu.com

available that can be reused. These can be included into a CSAR without modifying them. Configuration definitions can also be used to perform simple management tasks. However, configuration definitions are created using a domain-specific language (Günther et al., 2010). Consequently, that specific language has to be learned when manually creating such an artifact.

The examples and the evaluation presented in this paper were strongly related to Chef. However, the underlying concepts are not Chef-specific. As a result, the integration concepts can be adapted to be used and implemented in conjunction with other configuration management tooling such as Puppet because they own a similar architecture as outlined in Section 3.2. This will be evaluated in future work.

OpenTOSCA[14] is an open source implementation that provides a TOSCA runtime environment. Currently, OpenTOSCA is being extended to process implementation artifacts of type "script artifact." Then, the integration concepts described in Section 4 can be further evaluated based on OpenTOSCA.

Moreover, it will be evaluated how existing configuration definitions can be reused even more easily in the world of model-driven Cloud management. This makes sense because of the existing and growing ecosystems of configuration management products.

The contributions described in Section 4 are mainly limited to service deployment. Future research will focus on utilizing configuration management for management tasks that are performed after the actual deployment of a service instance. The goal is to orchestrate configuration definitions using management plans as they can be defined in a TOSCA service template.

ACKNOWLEDGEMENTS

The research leading to these results has partially received funding from the 4CaaSt project from the European Union's Seventh Framework Programme (FP7/ 2007-2013). Further, this work was partially funded by the BMWi project CloudCycle (01MD11023).

REFERENCES

Binz, T., Breiter, G., Leymann, F., and Spatzier, T. (2012). Portable Cloud Services Using TOSCA. *Internet Computing, IEEE*, 16(3):80–85.

Binz, T., Leymann, F., and Schumm, D. (2011). CMotion: A Framework for Migration of Applications into and between Clouds. In *2011 IEEE International Conference on Service-Oriented Computing and Applications*. IEEE.

Breitenbücher, U., Binz, T., , Kopp, O., and Leymann, F. (2013). Pattern-Based Runtime Management of Composite Cloud Applications. In *Proceedings of the 3rd International Conference on Cloud Computing and Services Science (CLOSER)*.

Delaet, T., Joosen, W., and Vanbrabant, B. (2010). A Survey of System Configuration Tools. In *Proceedings of the 24th Large Installations Systems Administration (LISA) conference*.

Günther, S., Haupt, M., and Splieth, M. (2010). Utilizing Internal Domain-Specific Languages for Deployment and Maintenance of IT Infrastructures. Technical report, Very Large Business Applications Lab Magdeburg, Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg.

Han, H., Kim, S., Jung, H., Yeom, H., Yoon, C., Park, J., and Lee, Y. (2009). A RESTful Approach to the Management of Cloud Infrastructure. In *2009 IEEE International Conference on Cloud Computing (CLOUD'09)*.

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

Humble, J. and Molesky, J. (2011). Why Enterprises Must Adopt Devops to Enable Continuous Delivery. *Cutter IT Journal*, 24(8):6.

Leymann, F. (2009). Cloud Computing: The Next Revolution in IT. In *Photogrammetric Week '09*. Wichmann Verlag.

Leymann, F. (2011). Cloud Computing. *it – Information Technology*, 53(4).

Leymann, F., Fehling, C., Mietzner, R., Nowak, A., and Dustdar, S. (2011). Moving Applications to the Cloud: An Approach Based on Application Model Enrichment. *International Journal of Cooperative Information Systems*, 20(3):307.

Loope, J. (2011). *Managing Infrastructure with Puppet*. O'Reilly Media, Inc.

Mell, P. and Grance, T. (2011). The NIST Definition of Cloud Computing. *National Institute of Standards and Technology*.

Milojičić, D., Llorente, I., and Montero, R. (2011). OpenNebula: A Cloud Management Tool. *Internet Computing, IEEE*, 15(2):11–14.

Nelson-Smith, S. (2011). *Test-Driven Infrastructure with Chef*. O'Reilly Media, Inc.

OASIS (2007). Web Services Business Process Execution Language (BPEL) Version 2.0.

OASIS (2012). Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0, Committee Specification Draft 04.

OMG (2011). Business Process Model and Notation (BPMN) Version 2.0.

Shamow, E. (2011). Devops at Advance Internet: How We Got in the Door. *IT Journal*, page 14.

Smith, D. (2011). Hype Cycle for Cloud Computing, 2011.

Turnbull, J. and McCune, J. (2011). *Pro Puppet*. Apress.

Vaquero, L., Rodero-Merino, L., Caceres, J., and Lindner, M. (2008). A Break in the Clouds: Towards a Cloud Definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55.

[14]http://www.iaas.uni-stuttgart.de/OpenTOSCA