Unified Invocation of Scripts and Services for Provisioning, Deployment, and Management of Cloud Applications Based on TOSCA

Johannes Wettinger, Tobias Binz, Uwe Breitenbücher, Oliver Kopp, Frank Leymann and Michael Zimmermann

Institute of Architecture of Application Systems, University of Stuttgart, Universitätsstraße 38, Stuttgart, Germany

Keywords: Management Plans, Management Operations, Script Invocation, Unified Interface, TOSCA, DevOps.

Abstract: There are several script-centric approaches, APIs, and tools available to implement automated provisioning, deployment, and management of applications in the Cloud. The automation of all these aspects is key for reducing costs. However, most of these approaches are script-centric and provide proprietary solutions employing different invocation mechanisms, interfaces, and state models. Moreover, most Cloud providers offer proprietary Web services or APIs to be used for provisioning and management purposes. Consequently, it is hard to create deployment and management plans integrating several of these approaches. The goal of our work is to come up with an approach for unified invocation of scripts and services without handling each proprietary interface separately. A prototype realizes the presented approach in a standards-based manner using the Topology and Orchestration Specification for Cloud Applications (TOSCA).

1 INTRODUCTION

Automated provisioning, deployment, and management of Cloud applications are key enablers to reduce costs of their operation (Leymann, 2009). However, it is hard to implement automated processes for managing applications in production after the actual development is finished. The main reason is that key aspects for operating an application in production were not considered during development. The idea of *DevOps* (development & operations) aims to eliminate the barrier between developers and operations personnel (Humble and Molesky, 2011). Consequently, a highly automated management environment has to be established during development already to continuously deploy and manage new iterations of an application to different environments (e.g., development, test, and production).

When automating such deployment and management processes there are typically a number of building blocks involved: (i) different Cloud providers expose proprietary APIs to *provision* and *manage* resources such as storage and virtual machines. (ii) Several tools such as Chef (Nelson-Smith, 2011) originating in the DevOps community provide a means to automatically *deploy* application components on top of the Cloud resources provisioned before. The DevOps community also publicly offers reusable artifacts to be used for the deployment of particular application components. (iii) Custom scripts (e.g., Unix shell scripts) may be implemented to *manage* the application during runtime (e.g., adding new users).

However, it is hard to combine these building blocks because the invocation mechanisms of Web services/APIs, scripts, and other plans involved in these automated processes differ significantly and the majority of them is not standards-based (Breitenbücher et al., 2013). The main goal of our work is to address this deficit by creating a management bus to provide a unified invocation interface hiding all the technical details of the employed management technologies. This interface is used by management plans implementing the logic required to automate a particular management task. The main contributions of our work are outlined in the following. We present:

- Concepts for unified invocation of different kinds of scripts (e.g., used for deployment) and different APIs (e.g., used for provisioning).
- Architecture and prototype of a unified invocation bus and interface to be used by management plans based on the Topology and Orchestration Specification for Cloud Application (TOSCA) (OASIS, 2013).
- Evaluation based on an end-to-end open source toolchain to show that plans get simpler and the

Wettinger J., Binz T., Breitenbücher U., Kopp O., Leymann F. and Zimmermann M..
Unified Invocation of Scripts and Services for Provisioning, Deployment, and Management of Cloud Applications Based on TOSCA.
DOI: 10.5220/0004859005590568
In *Proceedings of the 4th International Conference on Cloud Computing and Services Science* (CLOSER-2014), pages 559-568
ISBN: 978-989-758-019-2

number of specific "wrappers" used for invoking certain scripts and services are reduced.

The remaining of this paper is structured as follows: 2 outlines the currently existing issues and limitations we address in our work. We present TOSCA as an emerging standard for managing Cloud applications in 3 because major parts of our work is based on it. 4 discusses the notion of a management plan and the orchestration of management operations using plans. Our concepts regarding the unified invocation approach are presented in 5, followed by an evaluation of this approach in 6. Finally, we discuss some related work in 7 and conclude this paper in 8.

2 PROBLEM STATEMENT

Regarding the state of the art of provisioning and deployment of Cloud applications, several approaches emerged in the last few years. Some of the most popular approaches today are based on tools originating in the DevOps community such as Chef (Nelson-Smith, 2011), Puppet (Loope, 2011), CFEngine (Zamboni, 2012), and Juju[1]. (Delaet et al., 2010) survey and compare some of the technical aspects of these tools in more detail. The main purpose of these tools is automating the installation and configuration of application components on physical and virtual machines. This is the reason why they are often called *configuration management* tools. To implement deployment logic for application components such as a Web server or a database, scripting languages are used. As an example, Chef uses a domain-specific language (Günther et al., 2010) based on Ruby to implement deployment and management operations. In case of Chef, these operation implementations are called *recipes,* bundled in *cookbooks*[2]. Since each tooling uses different concepts for developing, managing, and invoking management operations, it is hard to combine them in order to automate the deployment of a certain application stack. Such a stack typically consists of several components, some of which are hosted on others (e.g., a PHP application *hosted on* on an Apache HTTP Server) and connected to each other (e.g., an application is *connected to* a database). We further refer to such a stack as an *application topology.*

The hard-to-realize interoperability of these approaches would make a lot of sense because the communities affiliated with these tools provide plenty of reusable management operations such as cookbooks provided by the Chef community and charms[3] offered by the Juju community. These operations can be used to implement deployment plans automating the deployment of whole application stacks consisting of different components. Especially components that are commonly used as middleware such as the Apache HTTP Server can be deployed by parameterizing these operations. As an example for the deployment of a typical stack, a publicly available Juju charm can be used to create a MySQL database cluster, an already available Chef recipe may be used to create an instance of the Apache HTTP Server, whereas the deployment of a particular PHP application is implemented by a custom Unix shell script. As of today, however, the orchestration of these operations is limited to the boundary of a single ecosystem, including its operations, its tooling, and its community.

Another limitation of the approaches described before is their limited management view on application topologies. Since an application typically is not a monolithic block, there are several application components involved that are related to each other. Some of these tools provide a means to model topologies and stacks such as the Juju GUI[4] and Amazon Web Services' OpsWorks[5], but most of them do not. This limited management view, mostly focusing on single more or less isolated parts of a topology, makes it hard to manage large and distributed applications. Relations between components are only modeled implicitly and are typically hard-wired in the operations' implementation. However, especially the relations are important for management to understand the dependencies and possible side effects of a management action. What these tools miss anyhow is a portable and standards-based approach to specify such topologies. This is another reason why the orchestration of different operation types is hard to implement because if there is a means to model an application topology it is limited to a single ecosystem. Furthermore, different approaches own different state models where properties are stored following a certain structure. A central and overarching state model is required to access these data in a unified manner.

The provisioning of resources such as storage or virtual machines in the Cloud is another key aspect in this context. This is typically the first step of a deployment process because a machine is required to be running to deploy and manage application components on it. In addition, provisioning of resources is a fundamental part of many management tasks. For example, to scale an application, additional resources must be

[1]Juju: http://juju.ubuntu.com

[2]Cookbooks: http://community.opscode.com/cookbooks

[3]Juju Charms: http://jujucharms.com

[4]Juju GUI: http://juju.ubuntu.com/resources/juju-gui

[5]AWS OpsWorks: http://aws.amazon.com/opsworks

provisioned. There are standardization and unification efforts going on such as OpenStack (Pepple, 2011) and Deltacloud[6] to provide a unified API for provisioning tasks across different Cloud providers. However, in practice there are still lots of different APIs such as the Google Compute Engine API[7], the Amazon Web Services API[8], and Baremetalcloud's API[9] to name a few. Consequently, in a multi-cloud management scenario where application components are distributed across different Cloud providers (e.g., for resilience purposes), it is hard to combine them because of their different APIs.

The goal of our work is to address the issues described before by presenting an approach for creating overarching plans orchestrating different kinds of operations and APIs. Furthermore, plans, covering both deployment and runtime management of Cloud applications, should be able to access a holistic topology model covering the whole automatically deployed application stack.

We present the most important concepts of the Topology and Orchestration Specification for Cloud Applications in the next section. The purpose of using this emerging standard in our work is to specify application topologies and package them together with corresponding management plans in a portable manner.

3 TOSCA

The Topology and Orchestration Specification for Cloud Applications (TOSCA) (OASIS, 2013) enables the definition of application topologies including their application components, the relations between them, and the resources where they are hosted on such as VMs. It is an emerging standard supported by a number of prominent industry partners such as IBM and Hewlett-Packard to enable the portable definition of topology models. 1 shows a TOSCA-based *topology model* for SugarCRM[10], an open source customer relationship management software. The SugarCRM application topology is used by the TOSCA interoperability subcommittee[11] as an example for different vendors to check whether their TOSCA runtime environment is compliant to the specification. The topology model is a graph consisting of *nodes* that represent both application components (Apache HTTP Server, SugarCRM App, etc.) and infrastructure resources (VM 1, VM 2). These nodes are linked by *relationships* to express that a particular node is hosted on another one (e.g., SugarCRM App is hosted on Apache HTTP Server) and to specify a connection between two nodes (e.g., application connects to the database).

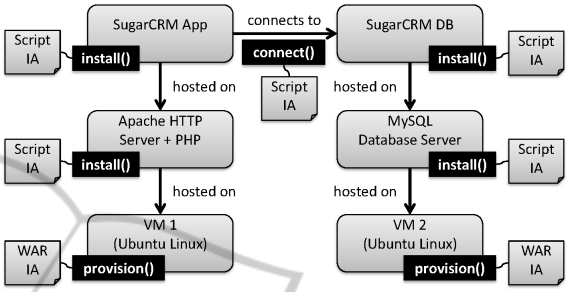

Figure 1: SugarCRM topology model.

Each node and relationship can have *management operations* attached. For instance, a VM node has a *provision* operation defined that is in charge of provisioning a new VM. Furthermore, nodes representing application components typically own an *install* operation to install and configure a particular application component. Operations can also be attached to relationships as it the case for the *connect* operation attached to the relationship wiring the application with the database. However, at this level an operation is still an abstract modeling construct. To make it executable, an *implementation artifact (IA)* needs to be attached to each operation that implements the operation. As an example, a WAR file exposing a Web service could be attached to the *provision* operation of a VM node. This WAR file is then deployed to a servlet container provided by the TOSCA runtime environment. Once the operation is invoked, a request is sent to the Web service performing the actions required to create a new VM. Another example is to attach a Unix shell script to the *install* operation of an application component node. When the operation is invoked, the script gets copied to the target VM and then is executed there. If several IAs are attached to a single operation, it is assumed that all of them implement the same logic based on different technologies. The TOSCA runtime environment is free to pick one of them to perform the operation.

In addition to defining the topology model and implementing the IAs, *plans* need to be written such as a build plan to create new instances of such a topology model. A plan basically defines the order of operation invocations to achieve a certain management goal (Binz et al., 2012). BPMN4TOSCA (Kopp et al., 2012) proposes TOSCA-specific modeling constructs to be used in plans based on BPMN (OMG, 2011). TOSCA

[6]Deltacloud: http://deltacloud.apache.org
[7]http://cloud.google.com/products/compute-engine
[8]AWS API: http://aws.amazon.com/documentation
[9]Baremetalcloud: http://www.baremetalcloud.com
[10]SugarCRM: http://www.sugarcrm.com
[11]SugarCRM Interop Topology: http://goo.gl/C6pqRd

further defines a packaging format called *Cloud Service Archive (CSAR)* to deliver all ingredients belonging to a particular topology model such as IAs, plans, and the topology model itself as a single file, a CSAR.

The following 4 outlines the notions of management plans and management operations as well as how they work together. These are the fundamentals for the concepts we describe later.

4 MANAGEMENT PLANS AND OPERATIONS

Management logic is implemented on different levels: script-centric artifacts provided by the DevOps community (Wettinger et al., 2013) typically implement the installation and configuration of application components such as an Apache HTTP Server or a PHP module on a physical or virtual machine (VM). Since the nature of these artifacts is relatively fine-grained, specifying low-level management tasks that need to be performed, we refer to these artifacts as *management operations.* They are typically designed in a way to perform a configurable "atomic" management action such as *installing an Apache HTTP Server* or *adding a new user account to an application.* These operations focus on the deployment of application components as well as their management at runtime such as changing the configuration of a particular component:

Management Operations *automate management actions that are considered "atomic", i.e., it does not make sense to split them up further to achieve a certain low-level management goal.*

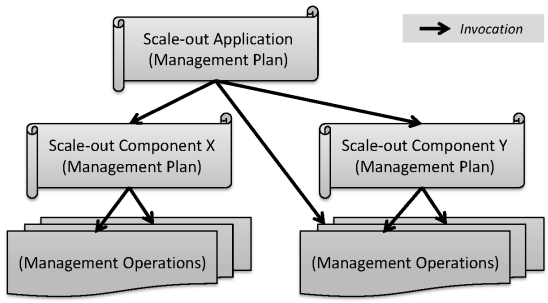

Figure 2: Plans orchestrate plans and operations.

A particular management operation may, for instance, represent a script performing a particular management action on a VM. However, it could also be an invocation of a service or an API offered by a Cloud provider, for instance, to provision new resources such as VMs. In order to enable the orchestration of several management operations, for instance, to deploy a complete application topology, more coarse-grained plans are required. We refer to these plans as *management plans:*

Management Plans *control the invocation of management operations and other management plans. They specify the proper ordering of the invocations as well as handling the input and output of each invocation.*

As mentioned in the definition before, plans may also be used to combine existing plans and operations, so plans can be positioned on different levels as shown in 2. As an example, a scalable application typically consists of several components that are deployed in a distributed manner. In order to scale such an application in or out, some of the components need to be scaled according to the current workload. There could be a management plan for each component that realizes its scale-out. A plan on this level (e.g., implemented as a Ruby script) may invoke operations (e.g., Chef recipes) to reconfigure deployed instances that belong to this particular component of the application as well as operations to create and configure new instances to enable the scale-out. Moreover, an overarching management plan (e.g., realized as a workflow) could be implemented to invoke individual management plans for each component involved in the scale-out process. Finally, management plans enable multi-cloud management scenarios: a particular plan may invoke different operations for resources distributed across different Cloud providers such as installing a database on a VM at provider A and installing an application server on a VM at provider B. Furthermore, such a management plan can use APIs and services (management operations) exposed by different providers to provision new resources such as storage or a VM.

Actually, management plans are very similar to workflows and business processes. (Binz et al., 2012) describe how to use existing workflow technology and standards such as the Business Process Model and Notation (BPMN) (OMG, 2011) and the Business Process Execution Language (BPEL) (OASIS, 2007) to implement and execute management plans. The recursive character of management plans, i.e., orchestrating existing management plans by another management plan, is similar to how BPEL deals with Web services: a BPEL workflow is not limited to invoking existing Web services. It can also be exposed as a new Web service that can then be invoked by another BPEL workflow, possibly as subprocess (Kopp et al., 2010). The distinction of management logic in plans and operations is similar to how workflows are typically distinguished: low-level workflows (similar to operations) are often referred to as *micro flows* (Manolescu and Johnson, 2001) that can be orchestrated by "real" workflows (similar to plans).

By implementing management plans as workflows based on established standards such as BPMN or BPEL, the plans' portability is improved. Existing general-purpose workflow engines can be used to execute these plans. Moreover, the plans inherit the properties of workflows such as recoverability and compensation as discussed by (Binz et al., 2012).

To enable the portability of management operations, domain-specific languages (DSLs) such as the Ruby-based DSL used by the Chef community (Günther et al., 2010) can be used. These languages provide platform-independent abstractions for typical low-level management and deployment actions such as installing a particular software package, writing a configuration file, or setting file permissions.

Again similar to workflows, there are different flavors of plans and operations: management logic can be implemented in an imperative or in a declarative manner. Whereas a declarative implementation only specifies *what* actions have to be performed, an imperative implementation additionally describes *how* these actions need to be performed. As an example, a declarative management operation may define a set of software packages that need to be installed. The imperative variant would further define how to install these packages. Imperative management plans based on workflows can be realized using BPMN or BPEL. However, there are also declarative workflow languages (Pesic and van der Aalst, 2006) available that could be used to implement declarative management plans. For management operations, the very same distinction exists: some of the DevOps tools such as Puppet provide a declarative domain-specific language, whereas Chef recipes, for instance, can be implemented using both a declarative (Ruby-based domain-specific language) and an imperative approach (plain Ruby). Furthermore, Unix shell scripts as for instance used by several Juju charms are a typical example for imperative management operations because their logic encapsulates several system commands that exactly define what action to perform and how to perform this action. Conceptually, there is no limitation to combining plans and operations of different flavors. As a result, an imperative management plan could invoke several declarative plans and operations or vice versa.

In the following section we present concepts for a standards-based approach to tackle the problems outlined in 2. Furthermore, we come up with an architecture and a prototype that realizes these concepts based on TOSCA.

5 UNIFIED INVOCATION INTERFACE FOR PLANS

As outlined in 2 it is difficult to combine different management operations such as Chef recipes, Unix shell scripts, or certain provider APIs because each of them has individual invocation mechanisms. To address this issue the goals of our concepts presented in this section are threefold:

1. Creating management plans has to be as simple as defining the flow logic (e.g., a sequence) of operations of particular nodes or relationships in a TOSCA topology model. The technical details of invoking a particular operation are completely hidden to such a management plan.
2. The number of implementation artifacts (IAs) created for a CSAR should be reduced as much as possible. As an example, the CSAR should not deliver an IA that implements the execution of scripts of a certain kind, because this logic does not immediately belong to the application's management logic and thus has to be provided by other means.
3. Different operations in a topology model can be implemented using different kinds of IAs such as Chef recipes, Unix shell scripts, or Juju charms. However, there should not be any dependency on additional IAs (e.g., an IA exposed as Web service to execute a Chef recipe) to enable the execution and orchestration of such operations.

All these goals enable *separation of concerns,* so that (i) a management plan focuses on the logic of combining operations properly, (ii) a CSAR contains artifacts and plans only related to the application it specifies, and (iii) different kinds of IAs can be used to implement a particular operation without changing higher-level management plans.

To achieve these goals a *unified invocation interface* is required that can be used by plans to invoke operations without knowing what kind of IA or technology is used in the background. The unified invocation interface is implemented based on OpenTOSCA (Binz et al., 2013), an open source TOSCA runtime environment. OpenTOSCA is one of the core components of an end-to-end toolchain to create and process TOSCA models. First, we outline how OpenTOSCA deals with plans, operations, and their invocation without such an interface. Then, we describe how we improved OpenTOSCA addressing the issues we discussed before.

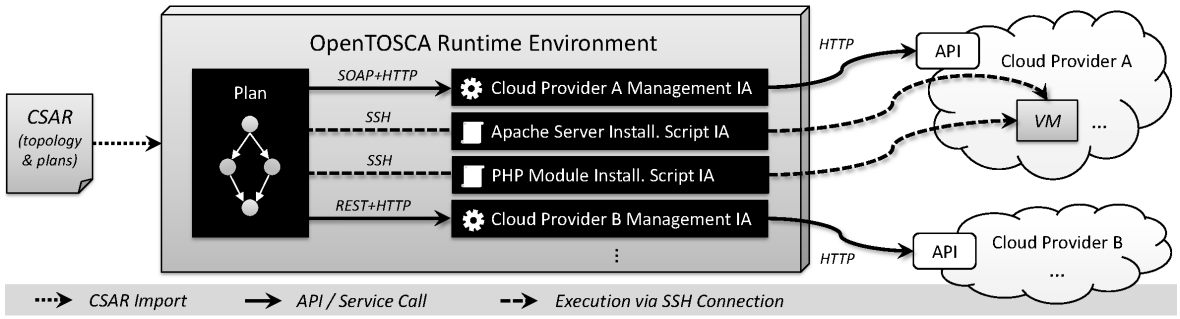

Figure 3: Original architecture without a unified invocation interface for plans.

5.1 Original Architecture of OpenTOSCA

3 shows a simplified overview of the architecture of OpenTOSCA. It further outlines the process of creating an instance of a particular application topology specified by a CSAR as it is currently implemented without the issues in mind we mentioned before. The initial step is to import a CSAR to the runtime environment, so that new instances can be created based on the model described in the CSAR. In addition to the actual topology model, the CSAR further contains all configuration and management plans as IAs and TOSCA management plans (Binz et al., 2012). We assume that each CSAR contains an overarching *build plan* specifying which operations of nodes and relationships in the topology have to be invoked in which order. This plan is executed by the OpenTOSCA runtime environment to build a new instance of the topology described in the model.

When the plan invokes a particular operation the corresponding implementation artifact (IA) is executed. Some IAs may be implemented as SOAP-based Web services in Java (WAR files), others may be implemented as Unix shell scripts or Chef recipes. When importing a CSAR, all implementation artifacts that have to run in the local OpenTOSCA management environment are deployed. For instance, the IAs providing Web services implemented in Java are hosted on a local Java servlet container that is part of the OpenTOSCA runtime environment. We refer to these IAs as *local IAs*. OpenTOSCA further supports local IAs providing RESTful Web services implemented in Java. All other IAs such as script IAs are obviously not deployed locally because they need to be executed on the target VMs remotely. Therefore, we refer to these IAs as *remote IAs*. Plans are bound to local IAs, so the corresponding plan and IAs are hard-wired. This fact decreases the portability and reusability of plans.

The sample plan outlined in 3 depicts several management operations that are performed in a multi-cloud management scenario. These represent a subset of operations required to create a new instance of the SugarCRM application topology described in 3. (i) The plan uses the local *Cloud Provider A Management IA* to interact with the API of provider A. The same applies to the *Cloud Provider B Management IA* for provider B. However, the invocation interfaces differ (SOAP+HTTP vs. REST+HTTP). Consequently, the plan needs to be aware of the interfaces' idiosyncrasies in terms of invocation, passing input data, and receiving the result of the execution. (ii) Furthermore, the plan invokes operations implemented by script IAs (Unix shell scripts) such as the *Apache Server Installation Script IA*. Since these scripts need to be executed remotely on a target VM at the provider, there are two options to deal with remote execution: the plan itself could implement the management logic for copying the scripts to the target machine using SSH, setting input and environment variables accordingly, performing the execution, and retrieving its output. Alternatively, a local wrapper IA such as an *SSH Unix Shell Script Invoker IA* could be implemented and packaged with a CSAR. This IA would expose the management logic for handling the execution of a certain type of script (Unix shell scripts) as Web service to be used by the plan. The second option is typically necessary in case the plan is implemented as workflow, e.g., based on BPEL, because such workflows cannot establish SSH connections without proprietary extensions.

Both options for executing remote IAs such as scripts pollute plans and CSARs with management logic that does not belong to the application's management logic and should therefore be provided by the management environment. In this case, such management logic can be reused by different CSARs. Furthermore, plans need to deal with different invocation mechanisms such as SOAP+HTTP, REST+HTTP, and SSH-based script execution. Thus, plans can get complex even if the application topology is relatively simple.

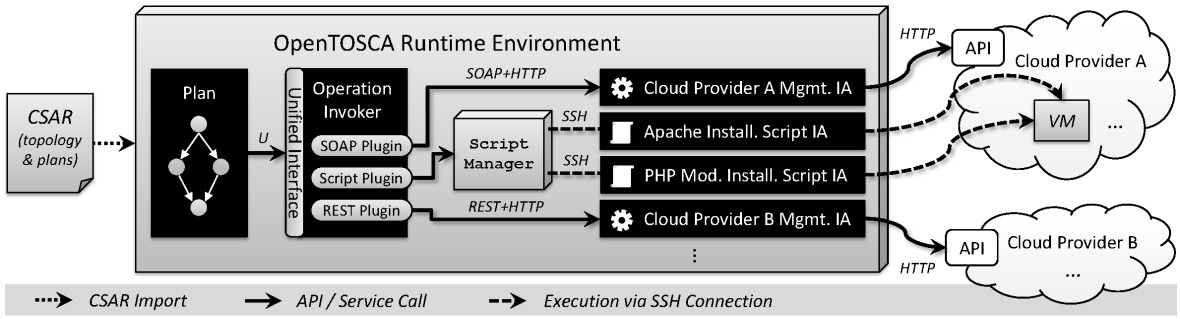

Figure 4: Improved architecture providing a unified invocation interface for plans.

5.2 Improved Architecture with Unified Invocation Interface

Based on the state of OpenTOSCA without a unified invocation interface presented before, 4 outlines how we improved OpenTOSCA's architecture to provide a unified invocation interface to plans. Thus, we implemented an additional component, namely the *Operation Invoker* acting as a bus in the sense of a service bus (Chappell, 2004). This component provides a unified interface to plans to invoke operations. In contrast to the original architecture outlined in 3, the plan does only communicate with the single unified interface of the Operation Invoker. Technically, the unified interface is implemented as SOAP-based Web service. However, it could alternatively or additionally make the interface accessible through a RESTful Web service or any other communication protocol.

The unified interface enables loose coupling between plans and IAs. When a particular operation is called through the Operation Invoker, it checks what kind of IA is available to execute the operation. Then, it checks whether there is a plugin registered that can execute IAs of the given type (e.g., an IA of type *SOAP* or of type *REST*). In case there is a corresponding plugin, this plugin gets invoked, which itself invokes the corresponding IA. The Operation Invoker is extensible by adding plugins to it enabling OpenTOSCA to process any kind of IA. These plugins are implemented using OSGi[12].

(Leitner et al., 2009) and (De Antonellis et al., 2006) present solutions similar to the Operation Invoker in order to dynamically invoke different kinds of Web services. However, these solutions are limited to service invocation and do not consider the execution of arbitrary executables such as scripts on a VM. In case of deploying and managing Cloud applications, service invocations do typically only cover provisioning tasks, e.g., creating a new VM. The actions to install and configure application components on the VM are typically implemented as scripts. We discussed this fact in 2, especially in the context of the tools such as Chef or Puppet originating in the DevOps community. Thus, it is essential for the Operation Invoker and the unified invocation interface that it provides a means to manage and monitor the execution of scripts. To be able to easily combine different kinds of scripts such as Chef recipes or Unix shell scripts the Operation Invoker has to deal with their individual invocation mechanisms and their way of handling input and output.

We decided to create the *Script Plugin* to enable the Operation Invoker to deal with IAs that are implemented as scripts. However, the actual logic for dealing with the idiosyncrasies of different types of scripts is realized by a separate reusable component, namely the *Script Manager,* which is used by the Operation Invoker's Script Plugin. When a plan invokes an operation that is implemented as a script, the Service Invoker calls the Script Manager and passes the references to the script files, some meta information such as the type of the script, and the context such as the properties of the node of which the particular operation belongs to. The following XML listing shows an example for a message that is created by the Script Plugin when a plan invokes the *install* operation of the *SugarCRM App* node:

```
<Definitions ...
 targetNamespace="http://www.opentosca.org/script"
 xmlns="http://.../tosca"
 xmlns:toscaBase="http://.../toscaBaseTypes"
 xmlns:s="http://www.opentosca.org/script">

 <ArtifactTemplate ... type="toscaBase:ScriptArtifact">
  <Properties>
   <toscaBase:ScriptArtifactProperties ...>
    <ScriptLanguage>sh</ScriptLanguage>
    <PrimaryScript>scripts/install.sh</PrimaryScript>
   </toscaBase:ScriptArtifactProperties>
  </Properties>
  <ArtifactReferences>
   <ArtifactReference reference="scripts/install.sh" />
  </ArtifactReferences>
```

[12]OSGi: http://www.osgi.org

```
17  </ArtifactTemplate>
18
19  <s:ArtifactContext>
20   <!-- OpenTOSCA's RESTful API to access CSAR content -->
21   <Files url="http://localhost/.../CSARs/SugarCRM/" />
22
23   <Operation>
24    <InputParameter name="licenseKey" type="string">
25     DUMMY_KEY
26    </InputParameter>
27    <OutputParameter name="stdout" type="string" />
28   </Operation>
29
30   <Node>
31    <!-- properties of SugarCRM App node -->
32    <NodeProperties nodeTypeName="SugarCRMApp">
33     <!-- admin password, app settings, etc. -->
34    </NodeProperties>
35
36    <!-- properties of server node on which the
37         SugarCRM App node is hosted -->
38    <HostProperties nodeTypeName="UbuntuLinuxServer">
39     <!-- public DNS address, SSH credentials, etc. -->
40    </HostProperties>
41   </Node>
42  </s:ArtifactContext>
43 </Definitions>
```

This message is then sent to the Script Manager. First, the script files (line 15) are retrieved from OpenTOSCA's RESTful API endpoint (line 21). Second, the Script Manager establishes, for instance, an SSH connection to copy the script files to the target VM, to set the corresponding environment variables, to execute the scripts, and to retrieve the result of the execution. The Operation Invoker's Script Plugin uses a RESTful API provided by the Script Manager to monitor the status of the execution and to finally retrieve the output to pass it back to the plan through the unified invocation interface. Such a message is structured as shown in the following XML listing:

```
1 <body>
2  <status>completed</status>
3  <output>
4   <param>
5    <name>stdout</name>
6    <type>string</type>
7    <value>SugarCRM installed successfully</value>
8   </param>
9  </output>
10 </body>
```

We evaluated our concepts based on the improved architecture of OpenTOSCA to show that our solution proposal can be implemented in practice to address the issues outlined in 2. The following section discusses the evaluation in more detail.

6 EVALUATION

Based on the topology of SugarCRM described in 3 we show that the concepts and the improved architecture presented in the previous 5 can be implemented in practice to address the issues discussed in 2. The overarching build plan to create new instances of SugarCRM is implemented using BPEL (OASIS, 2007). The main reason for using standards-based workflow languages as proposed in TOSCA is to benefit from the workflows' advantages such as recoverability and to reuse existing workflow engines for plan execution. However, the plan's implementation does not rely on any unique feature of BPEL. If certain workflow properties such as recoverability or compensation are not required for a particular plan it could also be implemented, for instance, using a scripting language such as Ruby or Python.

For evaluation purposes we created two CSARs including a build plan for each of them: (i) the original one does not use the Operation Invoker at all, so the build plan directly invokes Web services provided by local IAs bound to certain operations. The IAs' implementation is part of the CSAR. Consequently, a local wrapper IA was created to copy scripts on VMs and execute these scripts because a BPEL plan cannot do this directly. (ii) The improved CSAR's build plan invokes all operations through the Operation Invoker, meaning there is no direct dependency to any IA. Furthermore, there is no IA needed to invoke scripts because this task is delegated to the Script Manager. By using the Operation Invoker for each operation invocation centralized monitoring can be implemented because all input and output is going through a central component. 1 summarizes the evaluation results showing that (i) the complexity (measured as the number of XML elements in the BPEL plan) of the improved CSAR's build plan is lower, so plan creation gets simplified by using the Operation Invoker. This is mainly because many low-level actions such as copying scripts to the target machine do not appear in the plan. (ii) In contrast to the original CSAR the improved one does not require a special IA for invoking scripts because the

Table 1: Evaluation Results.

CSARs:	**Original**	**Improved**
Complexity of build plan (number of XML elements)	563	161
Wrapper IA for script invocation	yes	no
IAs exchangeable without changing plans	no	yes

Operation Invoker uses the Script Manager for this purpose. (iii) IAs can be arbitrarily exchanged in case of the improved CSAR because the plan only points to the operations that need to be invoked. Then, the Operation Invoker deals with the IAs and invokes them accordingly.

7 RELATED WORK

Invoking Web services dynamically by using some kind of unified invocation interface was discussed before (Leitner et al., 2009; De Antonellis et al., 2006). These approaches do not consider the invocation of arbitrary management operations such as Chef recipes or Unix shell scripts. However, their invocation is of utmost importance in case of creating management plans because many operations such as installing and configuring application components are implemented as scripts. In the field of multi-cloud research there is related work (Petcu et al., 2011; Moscato et al., 2011; Sampaio and Mendonça, 2011) proposing concepts and architectures to talk to different Cloud providers using a unified API. Only some of them (Liu et al., 2011) marginally consider the invocation of scripts on VMs in the target Cloud. Most of the time it is assumed that application components are bundled and deployed as virtual image files, e.g., using OVF.

In terms of provisioning resources such as VMs, libraries (e.g., jclouds[13] and fog[14]) enable the abstraction of specific Cloud provider APIs. However, their functionality is limited to provisioning tasks and they can only be used in a programmatic manner. Consequently, a standards-based plan (e.g., based on BPEL or BPMN) cannot use such libraries directly without a wrapper exposing the libraries' functionality as Web services. Beside these libraries, approaches such as cloud-init[15] can be used to run scripts on a VM when it gets started. This may work for the initial deployment of application components. However, arbitrary management operations such as a database backup that need to be performed later cannot be covered by this approach.

Some DevOps approaches can also perform provisioning actions. However, they are either bound to specific Cloud providers such as AWS OpsWorks[16], or they are mainly used as a command-line tool such as Chef's knife[17] and Juju[18]. Beside the DevOps tools such as Chef or Puppet, there is another category of tools emerging. These are in particular platform-as-a-service (PaaS) frameworks such as Cloud Foundry[19]. Their goal is to provide an integrated environment, namely a PaaS environment (Mell and Grance, 2011), for deploying and running applications and all their management processes. However, standards such as TOSCA are not considered there.

8 CONCLUSIONS AND FUTURE WORK

The unified invocation interface presented in 5 cleanly decouples overarching management plans (e.g., the application's build plan) from their underlying management operations (e.g., IAs implemented as Chef recipes and Unix shell scripts). The Script Manager used to invoke management operations implemented as scripts can be arbitrarily extended to support the execution of any kind of scripts. This enables reusing existing management operations published by the DevOps community that can be embedded into CSARs to implement certain operations. Finally, IAs exposing SOAP-based or RESTful Web services (e.g., to provision a new VM) can also be invoked by a plan through the Operation Invoker without binding the plan to a certain Web service. As a result, management plans focus on the actual orchestration of different operations and thus do not have any dependency on particular IAs or scripts. Our evaluation presented in 6 confirmed that this approach can be implemented in practice. Furthermore, the complexity of plans can be notably decreased.

In terms of future work we aim to extend our prototype in two dimensions: (i) extending the Script Manager to process a wide range of existing management operations (Puppet, Juju, etc.) and (ii) exposing the Operation Invoker's unified interface as RESTful Web service to make it directly usable for scripts implemented in Ruby, Python, etc. This enables the implementation of plans based on scripting languages as an alternative to workflows. Provisioning libraries such as jclouds mentioned in 7 can be wrapped as IAs to be reused in different CSARs for provisioning tasks.

ACKNOWLEDGEMENTS

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

[13] jclouds: http://jclouds.incubator.apache.org
[14] fog: http://fog.io
[15] cloud-init: http://launchpad.net/cloud-init
[16] AWS OpsWorks:http://aws.amazon.com/opsworks
[17] Knife: http://docs.opscode.com/knife.html
[18] Juju: http://juju.ubuntu.com
[19] Cloud Foundry: http://cloudfoundry.org

REFERENCES

Binz, T., Breitenbücher, U., Haupt, F., Kopp, O., Leymann, F., Nowak, A., and Wagner, S. (2013). OpenTOSCA - A Runtime for TOSCA-based Cloud Applications. In *Proceedings of 11th International Conference on Service-Oriented Computing (ICSOC'13)*, volume 8274 of *LNCS*, pages 694–697. Springer Berlin Heidelberg.

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

Breitenbücher, U., Binz, T., Kopp, O., Leymann, F., and Wettinger, J. (2013). Integrated Cloud Application Provisioning: Interconnecting Service-centric and Script-centric Management Technologies. In *Proceedings of the 21st International Conference on Cooperative Information Systems (CoopIS 2013)*.

Chappell, D. A. (2004). *Enterprise Service Bus*. O'Reilly.

De Antonellis, V., Melchiori, M., De Santis, L., Mecella, M., Mussi, E., Pernici, B., and Plebani, P. (2006). A layered architecture for flexible Web service invocation. *Software: Practice and Experience*, 36(2):191–223.

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.

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

Kopp, O., Binz, T., Breitenbücher, U., and Leymann, F. (2012). BPMN4TOSCA: A Domain-Specific Language to Model Management Plans for Composite Applications. In *Business Process Model and Notation*, volume 125 of *Lecture Notes in Business Information Processing*, pages 38–52.

Kopp, O., Eberle, H., Leymann, F., and Unger, T. (2010). The Subprocess Spectrum. In *Proceedings of the Business Process and Services Computing Conference: BPSC 2010*, volume P-177 of *Lecture Notes in Informatics*, pages 267–279. Gesellschaft für Informatik e.V. (GI).

Leitner, P., Rosenberg, F., and Dustdar, S. (2009). Daios: Efficient Dynamic Web Service Invocation. *Internet Computing, IEEE*, 13(3):72–80.

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

Liu, T., Katsuno, Y., Sun, K., Li, Y., Kushida, T., Chen, Y., and Itakura, M. (2011). Multi cloud management for unified cloud services across cloud sites. In *Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference on*, pages 164–169.

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

Manolescu, D.-A. and Johnson, R. E. (2001). *Micro-Workflow: A Workflow Architecture Supporting Compositional Object-oriented Software Development*. University of Illinois at Urbana-Champaign.

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

Moscato, F., Aversa, R., Di Martino, B., Fortis, T., and Munteanu, V. (2011). An analysis of mosaic ontology for cloud resources annotation. In *Computer Science and Information Systems (FedCSIS), 2011 Federated Conference on*, pages 973–980.

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 (2013). Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0, Committee Specification 01.

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

Pepple, K. (2011). *Deploying OpenStack*. O'Reilly Media.

Pesic, M. and van der Aalst, W. M. (2006). A Declarative Approach for Flexible Business Processes Management. In *Business Process Management Workshops*, pages 169–180. Springer.

Petcu, D., Craciun, C., Neagul, M., Lazcanotegui, I., and Rak, M. (2011). Building an interoperability api for sky computing. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 405–411.

Sampaio, A. and Mendonça, N. (2011). Uni4cloud: An approach based on open standards for deployment and management of multi-cloud applications. In *Proceedings of the 2Nd International Workshop on Software Engineering for Cloud Computing*, SECLOUD '11, pages 15–21. ACM.

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

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