Lego4TOSCA: Composable Building Blocks for Cloud Applications

Florian Haupt, Frank Leymann, Alexander Nowak, Sebastian Wagner

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{haupt, leymann, nowak, wagner}@iaas.uni-stuttgart.de

BibTeX:

```
@inproceedings{INPROC-2014-24,
  author    = {Florian Haupt and Frank Leymann and Alexander Nowak and
               Sebastian Wagner},
  title     = {Lego4TOSCA: Composable Building Blocks for Cloud Applications},
  booktitle = {Proceedings of the 7th IEEE International Conference on Cloud
               Computing (CLOUD 2014)},
  pages     = {160-167},
  year      = {2014},
  doi       = {10.1109/CLOUD.2014.31},
  publisher = {IEEE}
}
```

Lego4TOSCA: Composable Building Blocks for Cloud Applications

Florian Haupt, Frank Leymann, Alexander Nowak, Sebastian Wagner
Institute of Architecture of Application Systems, University of Stuttgart
Universitätsstr. 38, 70569 Stuttgart, Germany
{firstname.lastname}@iaas.uni-stuttgart.de

***Abstract*—The Topology and Orchestration Specification for Cloud Applications (TOSCA) enables the description, provisioning, and management of complex cloud applications in a portable way. TOSCA, therefore, provides a comprehensive yet complex set of mechanisms that may hinder users from unleashing its power due to misusing or neglecting parts of those mechanisms. TOSCA has just been standardized and, although it seems to be highly adopted in industry, there is a lack of systematic research of its features and capabilities. In this work we discuss the design of basic building blocks for cloud applications, called node types, and show how they can benefit from a deep integration with TOSCA. We developed a generic architecture for the realization of TOSCA node types, show an implementation of this architecture and validate it based on a sample cloud application. Our work gives an insight into the capabilities of TOSCA with respect to enable the creation of portable cloud services based on a set of composable building blocks.**

Keywords—Cloud Application; Cloud Service; Cloud Management; Service Management; TOSCA

I. Introduction

Cloud Computing is still one of the hottest topics in strategic IT development of organizations [15]. While most organizations have identified proper IT operations that are suitable candidates for cloud environments, the migration, deployment, and management of those components and applications in cloud environments is still an open issue. The *Topology and Orchestration Specification for Cloud Applications* (TOSCA) [1] is a standard that addresses those issues and allows stakeholders to describe applications and their management operations in a portable fashion [16]. While TOSCA provides a very precise language to describe the high-level topology components, i.e. the different building blocks of an application as well as their relations, and the management operations that should be provided by each node, there is a lack of description and support on how their implementations should look like. A proper implementation of those components, however, is crucial to benefit from all capabilities of TOSCA.

To address that drawback this work presents a comprehensive design framework to develop composable building blocks that are used to describe applications and their management in TOSCA. Consequently, the contribution of this work is twofold: (i) we present a design framework that describes how building blocks, i.e. *node types* and their related *implementation artifacts*, for applications described in TOSCA should be implemented. This framework not only describes the implementations of single building blocks, but also mechanisms that allow communication between multiple building blocks to resolve given dependencies between them. (ii) we describe, how the developed building blocks are used in TOSCA in an imperative and declarative fashion.

The remainder of this work is structured as following: Section II provides an overview on TOSCA. Next, section III describes the design-framework for implementing building blocks for TOSCA and section IV describes how to use them for deployment and management. Subsequently, section V presents the validation of the work based on a sample application, section VI summarizes the related work in this field and section VII concludes this work and provides an overview on future work.

II. TOSCA

The OASIS standard *Topology and Orchestration Specification for Cloud Applications* (TOSCA) was developed to create portable descriptions for automated management and provisioning of cloud applications [5][7]. The centerpiece of a TOSCA application description is the *application topology* which basically consists of a set of nodes and edges. Each node represents an individual component of a cloud application. This can be software components such as operating systems, application servers, virtual machines, etc. but also physical resources such as servers or network nodes. Edges define arbitrary dependencies between nodes, e.g. *hostedOn* or *uses*.

The example topology presented in Fig. 1 describes the components and their dependencies required for hosting and running a Web application with the name *NoteApp_L4T*. This Web application is *hostedOn* an Apache Web server with name *ApacheWS_L4T* which is installed on an *Ubuntu* Linux operating system that is running on an *Amazon EC2* virtual machine. The relations between all nodes, except between the nodes *NoteApp_L4T* and *MySQL_L4T*, are defined as *hostedOn*, i.e., the source of each *hosted on* relation serves as "runtime container" for the target node of the relation. The relationship *connectsTo* between the nodes *NoteApp_L4T* and *MySQL_L4T* indicates that the node *NoteApp_L4T* accesses the functionality of the node *MySQL_L4T*.

In TOSCA, an application topology is described by a *Service Template* that is created by an *Application Architect* (a person that knows the overall cloud application). Such a *service template* is represented as a XML file where the nodes of a topology are represented by *Node Templates* and their relations by *Relationship Templates*. These templates may specify property values and policies that are required for the management and provisioning of the components they represent. Moreover, to enable the reusability of implementations and management operations of a component, they refer to a so called *Node Type* or *Relationship Type*, respectively. In Fig. 1, the name of the *node type* that a *node template* refers to is shown in brackets. Consequently, a *node/relationship type* is a model for an arbitrary number of *node/relationship templates*. The management operations as well as the property schemes that are part of *node/relationship types* are defined by *Type Architects* that are experts for certain components and possible relations between other components [2]. A *node type* for an Apache Web server may, for instance, define management operations for starting and stopping the server, installing Web applications, etc. A *node type* may also define suitable properties to specify the URL and the port where the Web server can be accessed. Using an associated *node template*, concrete values can be assigned to these properties. In our example in Fig. 1 the *node template ApacheWS_L4T* assigns the *'http://lt4server.com'* URL to the property *BaseURL* and the value *'80'* to the property *Port*.

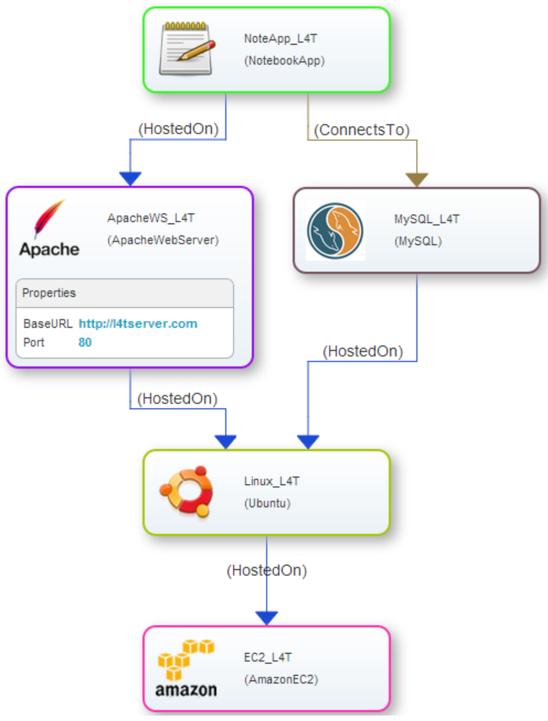


Fig. 1. Topology of the Notebook Web Application

Moreover, the *Type Architect* may also define requirements and capabilities for a *node type*. The requirements specify what features the *node type* requires from the *node type* it is related to. The capabilities on the other hand specify the features offered by a *node type*. All *node types* whose requirements and capabilities match can be related to each other in the topology. In our example in Fig. 1 the *node type NoteApp_L4T* may, for instance, define the requirement that the database must support data encryption. If the *node type MySQL_L4T* specifies in its capabilities that it offers encryption the *node templates* that are based on these *node types* can be interconnected.

So far we just discussed how a topology can be described in an abstract way by creating *service templates*. However, to operate and manage a cloud application and its components executable software artifacts are needed. The TOSCA specification, therefore, distinguishes between *Deployment Artifacts* and *Implementation Artifacts* that have to be provided by *Artifact Developers*. *Deployment artifacts* (DAs) are the executables that represent the single components of a cloud application, i.e., for each *node type* at least one DA has to be specified. The DAs can be physical files such as virtual operating system images, Web archive (WAR) files, etc. However, a DA can be also a running service such as Amazon EC2. In this case the *node type* does only provide a reference (usually an URL) to a service. In our example topology the *Ubuntu node type* is implemented by an Ubuntu Linux Amazon Machine Image (AMI).

An *Implementation Artifact* (IA) is an executable that implements the management operations specified by a *node-* or *relationship type*. An IA makes also use of the properties defined by the *node type*. Like DAs, IAs can be either provided as physical files along with the topology or as remote services. An IA can be implemented in different ways, e.g., as script or as Web service. Each *node-* or *relationship type* can also provide multiple IAs. For the *node type MySQL_L4T* there may be, for instance, two IAs defined, one is used for managing a MySQL database running on Windows and another is used for managing a MySQL database running on Linux.

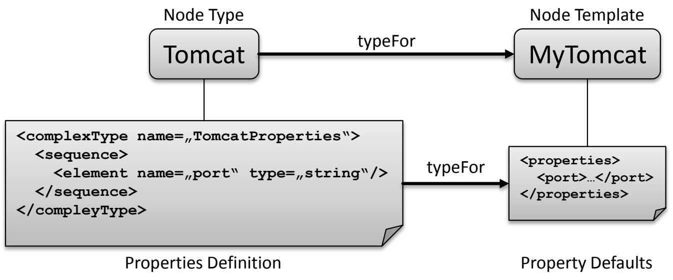
To provide an easy way to handle and exchange the definition files and the artifacts they can be packaged into a *Cloud Service Archive (CSAR)*. As the structure of a CSAR file is also defined by the TOSCA specification it can be deployed on any TOSCA compliant runtime environment (TOSCA container). Note, that the same cloud application (i.e., *service template*) may be provisioned multiple times (e.g., on different Amazon EC2 machines). This implies that the same *node-* and *relationship templates* are instantiated multiple times. Furthermore in this paper, we refer to an instance of a *node template* as *node instance* and to an instance of a *relationship template* as *relationship instance*.

III. Node Type Design

One goal of this work is to evaluate the design of the TOSCA language regarding node types. We are examining the main modeling constructs provided by TOSCA, we show how they can be used and what benefits they provide. To achieve this, we created a set of node types that are tightly integrated into TOSCA. The design of TOSCA node types comprises several fundamental design decisions. In this section, we will discuss the basic design decisions of the Lego4TOSCA Node Types.

A. Node Type Properties

A TOSCA node type may contain the definition of a properties document using a XML Schema Definition (XSD), as shown in Fig. 2 on the left side. A node template may then contain an instantiation of this properties document, shown in the right part of Fig. 2. This allows an application architect to configure the details of a node template as part of the application topology. In our work, we identified three different types of properties, each representing a different use case for the properties document.



```
<complexType name=„TomcatProperties“>
  <sequence>
    <element name=„port“ type=„string“/>
  </sequence>
</compleyType>
```

```
<properties>
  <port>…</port>
</properties>
```

Fig. 2. Node Type Properties

Configuration Properties are set by the application architect as part of the definition of a service topology. They allow the detailed configuration of a node template. The management operations, realized by the implementation artifacts, read these properties and then act accordingly. As an example, the Lego4TOSCA node type “Apache Tomcat” defines a property named “Port” defining which port it shall listen to. Another example for a configuration property is the machine type for an Amazon EC2 node template defining which EC2 instance type will be used.

Implementation Artifact Management Properties are used by the implementation artifact of a node template to persistently save data needed to realize its management operations. These properties are initially set by the implementation artifact of the corresponding node template and then read and changed during the lifetime of the corresponding Node Instance. As an example, the Lego4TOSCA Node Type “AWS EC2” defines a property named “Public DNS”. This property is initially set by the AWS EC2 implementation artifact when it creates and starts a new EC2 virtual machine. After that, the DNS name property is also accessed by other implementation artifacts. For example, the Apache Tomcat implementation artifact needs to know the DNS name of the underlying virtual machine to determine the public available URL of web applications hosted on a Tomcat server.

General Management Properties are used, similar to implementation artifact management properties, to persist data needed for management tasks. The main difference is that general management properties relate to the management of a whole service topology whereas implementation artifact management properties only relate to the management of a single node of a service topology. Global management tasks in TOSCA are typically modeled as management plans. Therefore, general management properties are typically written and read by management plans to realize complex management tasks. As an example, an application architect can define a management plan that initiates some cleanup operations on several nodes of a service topology. This plan may then, after each cleanup operation that successfully completed, write the current timestamp to the properties document of the respective node instance. If the same management plan is then executed again sometime later, it will read this property to decide, if another cleanup is needed or if the last cleanup still holds.

B. *Interface Design*

A TOSCA node type can offer one or more interfaces, each providing one or more operations defining input and output parameters. These interfaces describe the management operations offered by a node type. Management plans are then used to orchestrate these basic management operations into higher level management tasks. For example, the provisioning of a complex service topology comprises the ordered provisioning and configuration of multiple nodes like virtual machines, databases or application servers. The basic provisioning and configuration operations are offered by the corresponding node types. They can be orchestrated by a management plan in order to provision and configure a whole service topology.

The interfaces of the Lego4TOSCA node types basically comprise two types of parameters, *technical parameters* and *functional parameters*. Functional parameters are directly related to the management operation offered by a node type. In addition, the interfaces of the Lego4TOSCA node types contain two technical parameters. The *callback address* realizes asynchrony while the *node instance ID* uniquely identifies the target of the management operation. These two parameters are motivated and discussed in the following.

Management operations in general can be long running. For example, starting a virtual machine usually takes some minutes and backing up a huge database may even take some hours. Offering these operations over a synchronous interface would block the caller for a long time as well as typically introduce technical problems related to timeouts. Therefore, the interfaces of the Lego4TOSCA node types are consistently defined as asynchronous interfaces based on a generic callback mechanism.

The interaction with the asynchronous interface of the Lego4TOSCA node types is demonstrated in Fig. 3. The caller, for example a management plan, initially calls the needed management operation. In addition to the functional parameters, which are not explicitly shown in this case, the caller has to specify a callback address. This address will then later be used to return the result of the execution of the management operation to the caller. When called, the implementation artifact at first generates a Universally Unique Identifier (UUID) [8] identifying the current execution of the called management operation. After that, the implementation artifact starts the asynchronous execution of the management operation using a specific mechanism provided by Java, the e*xecutor service*. For a better understanding, the interaction with the executor service has been simplified in this example. Directly after initiating the execution of the management operation, the implementation artifact returns the UUID generated before to the caller and thereby acknowledges the successful start of the execution of the called management operation. At this point in time, the management operation is still being executed by the executor service. The caller however is not blocked anymore and can continue to, for example, call additional management operations. At the time the executor service finishes the execution of the management operation, it returns the result to the callback address initially provided by the caller. The result message always contains the UUID identifying the specific execution of the called management operation. This way, the caller is able to correlate the received result message to the initial operation call.

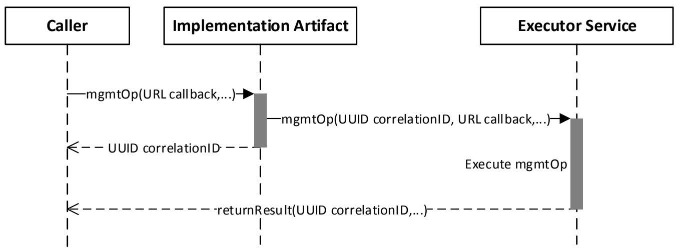


Fig. 3. Callback Mechanism

Another common approach to realize asynchronous communication is the use of a messaging system [9]. In this case, the interaction between participants is realized by sending messages to and receiving messages from so called channels, either queues or pub-sub topics. In contrast to the callback mechanism realized by the Lego4TOSCA node types, the messaging based approach requires the presence of a messaging middleware. By choosing the callback mechanism, the Lego4TOSCA node types do not rely on the presence of an appropriate messaging middleware and are therefore self-contained and portable.

The second technical parameter of each Lego4TOSCA interface is induced by the fact, that implementation artifacts are defined per node type. As there are in general multiple node templates for one node type and likewise multiple instances of one node template, one implementation artifact realizes the management operations for all instances of the corresponding node type. Therefore, when calling a management operation of an implementation artifact, the caller has to provide some data identifying the target of each management operation call.

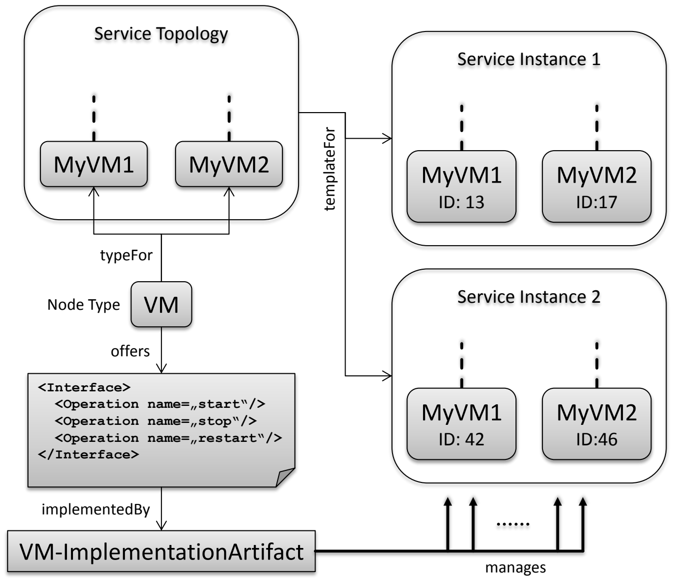


Fig. 4. Implementation Artifact managing multiple Node Instances

The relation between implementation artifacts and the components managed by them is depicted in Fig. 4. The node type for a virtual machine (VM), as shown on the left side, offers an interface to manage virtual machines. In addition to the interface definition, the node type also provides an implementation artifact that implements the management operations defined by the interface. This node type can then be used by an application architect to model a service topology, possibly containing multiple node templates of this node type (as shown in the upper left side of Fig. 4). This service topology can thereafter be used to create several instances of the modeled service. In the example shown in Fig. 4 there are two instances of the modeled service created. Nevertheless, there is only one implementation artifact available to manage all instances of the virtual machine node type.

Listing 1, line I, shows the signature of the restart operation of the virtual machine node type. The parameter "soft" is a functional parameter used to influence the detailed behavior of the restart operation. If defined this way, the call of this operation cannot be related to a specific virtual machine. An obvious solution would be to extend the interface of the virtual machine node type with additional parameters providing all data needed to identify and access a specific virtual machine (i.e. an instance of the corresponding node template). This is shown in line II of Listing 1. The parameter "vmid" identifies a specific virtual machine, the parameter "accesstoken" provides access credentials needed to access this virtual machine. However, defining the operation like this requires the caller to know and manage the details of how to identify and access a certain virtual machine. As discussed before (section III.A), this kind of data can also be stored in the properties document of a node instance. Taking advantage of this feature, the signature of the restart operation can be defined as shown in line III of Listing 1. The parameter "nodeID" identifies the node instance the operation call is related to. This ID can then be used by the implementation artifact to access the corresponding properties document. This document contains all data needed to access the targeted virtual machine, for example the identifier of the virtual machine and the access credentials.

```
  I. restart(Boolean soft)

 II. restart(Boolean soft, String vmid,
             String accesstoken)

III. restart(Boolean soft, String nodeID)
```

Listing 1. Operation Signatures

The Lego4TOSCA node types follow the approach shown in line III of Listing 1. The management operations of all node types contain a single parameter identifying the targeted node instance and therefore the properties document of the targeted node instance. This way, the signature of each operation is defined in a consistent way, providing a simple and intuitive way of identifying the target node. An application architect using this interface, for example when modeling management plans, can concentrate on the domain specific functional parameters and does not need to care about technical details how to identify and access node instances.

C. Composability

The Lego4TOSCA node types are a set of common used building blocks for cloud applications. On the infrastructure level, node types for the Amazon Web Services Elastic Compute Cloud (AWS EC2) and for the VMWare ESXi Hypervisor are provided. On the operating system level there are node types for Ubuntu Linux and Windows Server. The middleware level comprises node types for Apache Tomcat (a Servlet Container), the Apache Web server (a HTTP Server), the MySQL database and the WSO2 Business Process Server (an open source BPEL workflow engine). The complete set of the Lego4TOSCA node types is shown in Fig. 5.

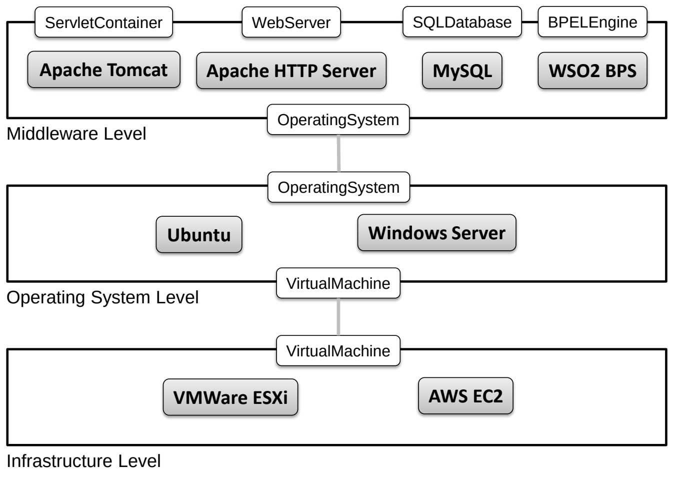


Fig. 5. Lego4TOSCA Node Types

In order to provide a simple and intuitive way to model service topologies, the Lego4TOSCA node types are built to be easily composable and interchangeable. Although the management of, for example, a MySQL database differs significantly depending on if it is hosted on a Linux or a Windows system, the Lego4TOSCA node type for MySQL can be combined with the Ubuntu node type as well as with the Windows Server node type. The possible combinations of different node types can be expressed in TOSCA by defining corresponding *requirements and capabilities*. As depicted in Fig. 5, the node types "VMWare ESXi" and "AWS EC" both provide the capability "VirtualMachine". On the other side, the node types "Ubuntu" and "Windows Server" require exactly this capability. This way, all possible combinations of the Lego4TOSCA node types are already part of their definition.

Most of the management operation provided by the node types operations can only be realized depending on the usage context of the node type. For example, how to install and start a Tomcat server heavily depends on whether it is hosted on a Linux or a Windows system. The Lego4TOSCA node types realize this kind of operation by dynamically interacting with each other. An example is shown in Fig. 6 as (2) using dotted lines. When the implementation artifact of the Tomcat node type is called in order to start a Tomcat server, the Tomcat implementation artifact interacts with the implementation artifact of the underlying operating system. Starting a Tomcat server can for instance be realized by a shell script. Therefore, the Tomcat implementation artifact calls the corresponding operation of the Ubuntu implementation artifact. This implementation artifact then connects to the targeted Ubuntu system using SSH and runs the given script.

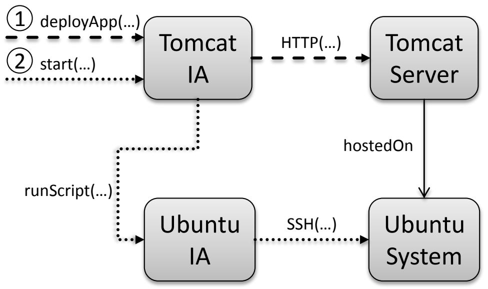


Fig. 6. Implementation Artifact Interaction

Following this design principle, the Lego4TOSCA node types *reuse already existing functionality* provided by other node types and realize the *separation of concerns* principle. In the given example, the implementation of connecting to a remote system using SSH and running scripts is encapsulated by the Ubuntu implementation artifact. The Tomcat implementation artifact simply reuses this functionality offered by the Ubuntu node type as a management operation.

D. Implementation Artifact Architecture

The common architecture of the implementation artifacts of the Lego4TOSCA node types is shown in Fig. 7. This architecture realizes the design decisions discussed before. In the following, the architecture will be explained using the Tomcat implementation artifact as an example.

In step 1 a *management operation* of the node type, realized by the corresponding implementation artifact, is called. The first parameter is the node instance ID identifying the target of the operation call. The second parameter is a functional parameter indicating that the Tomcat server should be started with an opened debug port 8000. The operation call is immediately acknowledged by returning the unique ID of the current operation execution ("897uhekfkj").

In step 2, the provided node instance ID is used by the *properties retrieval* component to fetch the properties document of the targeted node instance from the TOSCA container. This document contains all data needed to access and manage the Tomcat server.

In step 3, the *strategy selection* component selects the appropriate implementation of the called operation, also called a *strategy*. This approach follows the *strategy pattern*, a generic mechanism to determine appropriate behavior depending on a given context [10]. The context for strategy selection is the usage context of the corresponding node type. In the example shown, starting a Tomcat server differs depending on what operating system it is installed.

In step 4, the selected strategy component may interact with the implementation artifacts of other node types to realize the called management operation. In the example depicted in Fig. 7, the underlying operating system is called to execute a shell script starting the Tomcat server. This operation call immediately returns the unique ID of the initiated operation execution ("asdsd45543j").

In step 5, the result of the script execution is returned to the Tomcat implementation artifact using its *callback API*. The result message contains the unique ID of the operation execution, allowing correlating operation calls and resulting messages.

In step 6, the *callback handler* component processes the result message. It may either continue to interact with other node types, possibly using the *strategy selection* component again (6a) or it may also return a result message to the initial caller of the executed management operation (6b).

E. Analysis

In this section, we will discuss, how the presented design decision and the corresponding implementation artifact architecture influences the non-functional properties of the Lego4TOSCA node types.

The usage of properties documents has multiple effects. The use of *implementation artifact management properties* to persist management related data enables to build *stateless implementation artifacts*. All data needed to execute a called management operation can be read from the corresponding properties document. Implementation artifacts do not need to store any data. Stateless components in general *enable scaling* by instantiating them multiple times and also allow creating *more robust systems*, as failed component instances can simply be replaced by other ones. The usage of *configuration properties* allows to *ease the use of management operations* and to build *simpler management plans*. When using configuration properties, the configuration of nodes, and therefore also the configuration of single management operations, is already contained in the service topology. The caller of a management operation does not need to provide any, or at least most of, functional parameters. This allows domain experts using the management operations to concentrate on what to do and not on how to do it in detail.

The *requirements and capabilities* defined by the node types, in combination with the *strategy pattern* as the underlying implementation of this feature, enable *simple composability*. They hide the complexity related to technical dependencies between different node types from the application architect. Besides a *more intuitive modeling experience*, it is also possible to *exchange nodes of a topology*, as long they are compatible regarding their requirements and capabilities.

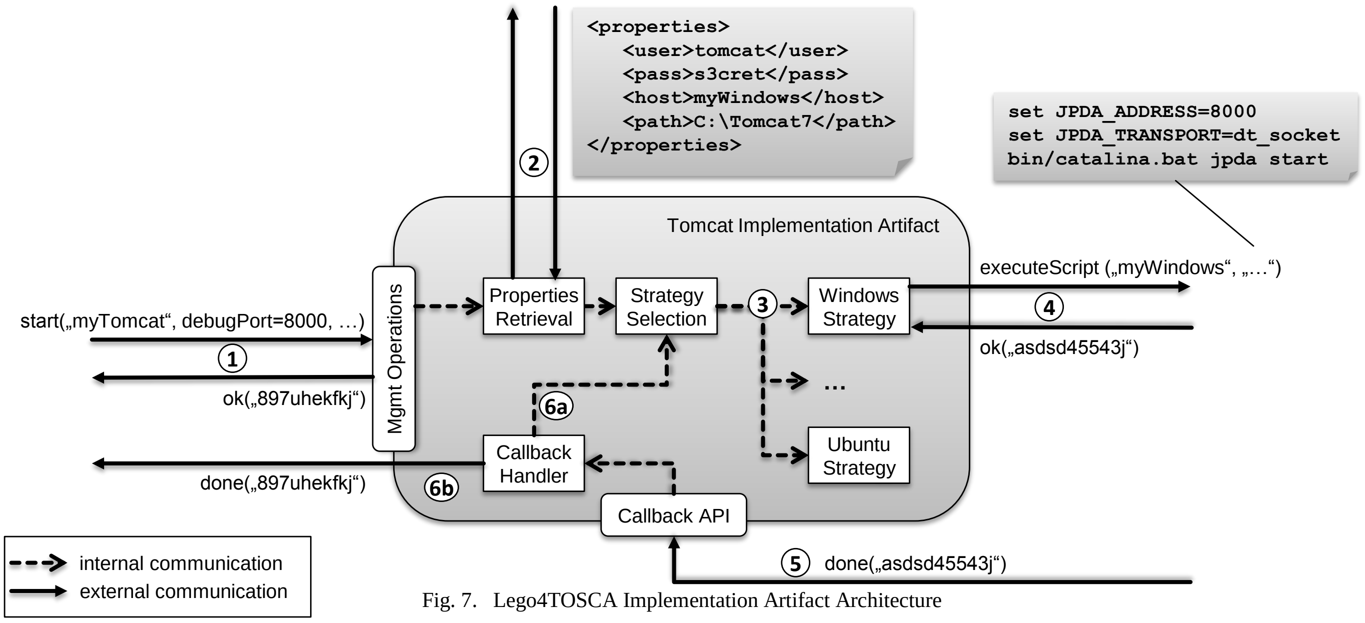
IV. Node Type Usage

In this section we discuss how the operations of node types can be used via their interfaces to perform management tasks and how the execution of these operations can be orchestrated by plans to manage the whole cloud application.

The node types we designed provide a set of common management functions for the respective components they are representing. The implementation artifacts of the Lego4TOSCA node types expose these abstract operations as Web service operations. The signatures of the operations consist of two different types of parameters: (i) functional parameters and (ii) technical parameters. To ease the use of the management operations, each operation accepts two signatures – with and without functional parameters. If the operation is called without functional parameters the implementation of the operation retrieves the values of the functional parameters from the properties document of the respective node. The technical parameters are mandatory for each management operation and have already been introduced and discussed in section III.B.

Recall that all operations may be invoked without a functional parameter if the expected value can alternatively be read from the node's properties. This frees the management plans from carrying too much redundant information but gives the flexibility to specify certain properties during instantiation or runtime of a plan.

The available management capabilities (represented by the available management operations) of a cloud application can be processed in two different ways: (i) *declaratively*, by using the TOSCA runtime environment or (ii) *imperatively*, by using


```
<properties>
   <user>tomcat</user>
   <pass>s3cret</pass>
   <host>myWindows</host>
   <path>C:\Tomcat7</path>
</properties>
```

```
set JPDA_ADDRESS=8000
set JPDA_TRANSPORT=dt_socket
bin/catalina.bat jpda start
```

Fig. 7. Lego4TOSCA Implementation Artifact Architecture

pre-defined management plans. Due to its strong dependencies to TOSCA runtime environments we will only briefly explain the declarative approach in this work.

In the declarative approach there are no management plans provided. The TOSCA runtime environment determines the operations and their execution order to provision or manage a cloud application solely from the available information within the service template, i.e. from the node templates and the relationship templates [2]. This requires the TOSCA runtime to "know" about the functionalities of all node types and relationship types within a service template and how to use them to accomplish management tasks. The management knowledge is, at least partly, encoded in the runtime environment. In the declarative approach the data required to manage a cloud application (e.g. Ports, URLs, etc.) are solely read from the node's properties. As the declarative approach also utilizes the operations of node types, the LegoTOSCA node types presented in this work can also be used with this approach. From an application architect perspective the declarative approach has the advantage that he just needs to create and maintain the topology of the cloud application, but he does not need to care about the orchestration of the management operations. Our experiments revealed that a declarative approach is usually just suited for simple management tasks that can be inferred from the topology. For more complex management tasks on the cloud application we suggest using management plans.

Imperative processing using management plans is more flexible for the provisioning and management of a cloud application. It is called *imperative processing* because a plan defines precisely "how" a cloud application has to be managed [2]. Hence, a plan specifies what operations of the different node types have to be called, in which order and what data are required to setup and manage the cloud application. Thereby, for different management tasks different plans can be defined. For instance, the plan for provisioning the Notebook Application shown in Fig. 1 would first setup an EC2 instance with Ubuntu Linux as operating system by using the management operations of the Amazon EC2 node type. Then the management operations of the Apache Web server and MySQL node types would be used to install them on the Ubuntu system. The management operations of these node types would afterwards be used to deploy and setup the Notebook Application. Since the implementation artifacts are implementing the actual logic of the management operations the plans are kept simple. They are just defining the execution order of the management operations as well as the required information. The application of the strategy pattern that is part of our approach leverages this simplicity even more: Plans can focus on a single node type but are freed from required logic that distinguishes between different node type stacks. For instance, a plan does not need to implement different management logic for an Apache Web server depending on if it is deployed on Windows or on Linux. We used BPEL [11] as workflow language for implementing the management plans because it offers many language constructs for implementing asynchronous service interaction which enables an easy integration of our asynchronous operations in the plans.

V. Validation

For validation purposes, we used the Lego4TOSCA node types to model and provision a sample cloud service as shown in Fig. 1. The depicted *Notebook Application* is a simple PHP application allowing creating, editing and deleting text notes. These notes are persistently stored in a MySQL database.

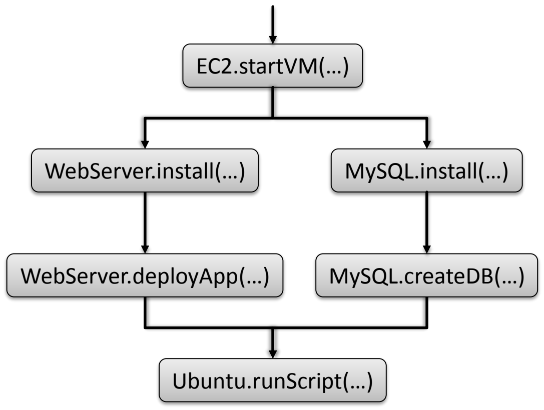


Fig. 8. Notebook Application Build Plan

In a first step, we modeled the service topology using the *Winery* application, an open source modeling tool for TOSCA cloud services [12]. Winery is an open source project hosted at the Apache Foundation[1] and is in addition available online[2] for testing purposes. After that, we created several BPEL plans realizing the provisioning, management and de-provisioning of the sample service topology. The topology model, the plans and the implementation artifacts are packed in one CSAR file.

As runtime environment for the created CSAR file we used the *OpenTOSCA container* [4], an open source TOSCA container[3]. The OpenTOSCA container deployed the Lego4TOSCA implementation artifacts on a Tomcat server and the contained BPEL plans on a WSO2 Business Process Server. After that, we were able to automatically create instances of the Notebook application by running the build plan depicted in Fig. 8.

For further validation and to extensively test the Lego4TOSCA node types we created some variants of the service topology described so far. To test the *composability feature* and the underlying *strategy pattern*, we exchanged the EC2 node with a VMWare ESXi node and also the Ubuntu node with a Windows Server node (in all possible combinations). The resulting CSAR files also worked as expected and the exchange of nodes in the service topology had only minor effect on the build plans.

VI. Related Work

Due to the fact that TOSCA has been published quite recently, there is only little work already conducted related to it. In [6] an approach is developed to integrate existing cloud management solutions based on Chef[4] with TOSCA. Similar to

[1] https://projects.eclipse.org/projects/soa.winery
[2] http://winery.opentosca.org/
[3] http://www.opentosca.org/
[4] http://www.opscode.com/chef/

our work, it is shown how build node types for TOSCA. In contrast, the focus of this paper is how to reuse and adapt an existing management approach including already existing artifacts. The deep integration with TOSCA and the possible benefits provided by it are not covered in this work.

In [13] the architecture of a cloud management system is introduced. The operations to manage cloud resources are provided over a RESTful API and an additional graphical user interface based on this API. The caller of a management operation interacts with a manager component which in turn interacts with the managed elements, for example a virtual machine or a web server. Similar to the Lego4TOSCA node types, the interaction between the manager component and the managed elements is based on a callback mechanism. In contrast, it is required that each manageable component hosts a special agent component.

In [14], a method enabling the modeling and automated provisioning of application topologies is presented. Similarly to TOSCA, complex applications are modeled using a graph based approach. In addition, this work focuses on the definition and resolution of so called *variability points* describing needed configuration activities during the setup of an application. Another focus is the realization of an automated application topology deployment, therefore following a declarative approach.

VII. Conclusion and Future Work

As a core result, this paper presents a comprehensive design guide for composable building blocks for cloud applications based on TOSCA. We started with an extensive discussion of several TOSCA features like *properties documents* and *requirements and capabilities*. We showed how they are used by the Lego4TOSCA node types and what benefits a deep integration with TOSCA can provide. As a result of our design decisions, the Lego4TOSCA node types are an *easy composable and easy to use* set of modeling artifacts to create complex cloud service topologies. In addition, the corresponding implementation artifacts are realized as stateless components and therefore provide a *scalable and robust* implementation of the provided management operations. We were able to validate our work using existing open source tools from the TOSCA domain like *Winery* and *OpenTOSCA*. We created a sample application and conducted extensive tests.

One aspect currently not supported by our node types is policies. In TOSCA, policies provide a means to express non-functional requirements to the nodes of a service topology. To realize these non-functional requirements at runtime, we aim at extending the Lego4TOSCA node types to be policy aware. Basic concepts regarding policies in TOSCA in general and also in combination with Lego4TOSCA have already been published [3].

As a more technical aspect we are also interested in benchmarking the Lego4TOSCA node types. A first approach would be to determine, how many management operations can be handled in parallel by a single implementation artifact. As the implementation artifacts are hosted by a TOSCA container, for example OpenTOSCA, and the properties documents are also managed by this container, we believe that a meaningful benchmark also has to include the performance of the TOSCA container.

Acknowledgment

This work was funded by the BMWi project Migrate! (01ME11055). We thank Robin Boldt, Marcus Eisele, Olaf Hoffeld, Thomas Kosch, Fabian Merkle, Maximilian Pfeffer, and Eric Seiz for their help with the first implementation of the Lego4TOSCA node types.

References

[1] Topology and Orchestration Specification for Cloud Applications Version 1.0. 25 November 2013. OASIS Standard. http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html.

[2] Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0. 31 January 2013. OASIS Committee Note Draft 01. http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.html.

[3] Waizenegger, T.; Wieland, M.; et al: *Policy4TOSCA: A Policy-Aware Cloud Service Provisioning Approach to Enable Secure Cloud Computing*. In: OTM 2013 Conferences.

[4] Binz, T.; Breitenbücher, U.; Haupt, F.; Kopp, O.; Leymann, F.; Nowak, A.; Wagner, S.: *OpenTOSCA – A Runtime for TOSCA-based Cloud Applications*. In: ICSOC 2013.

[5] Lipton, P. 2013. *Escaping Vendor Lock-in with TOSCA, an Emerging Cloud Standard for Portability*. CA Technology Exchange 4, 1, 49–55.

[6] Wettinger, J.; Behrendt, M.; et al: *Integrating Configuration Management with Model-Driven Cloud Management Based on TOSCA*. In: CLOSER 2013.

[7] Binz, T.; Breiter, G.; Leymann, F.; Spatzier, T.: *Portable Cloud Services Using TOSCA*. In: IEEE Internet Computing. Vol. 16(03), 2012.

[8] Leach, P. J.; Mealling, M.; Salz, R.: *A universally unique identifier (uuid) urn namespace*. IETF RFC. http://tools.ietf.org/html/rfc4122

[9] Hohpe, G.; Bobby, W.: *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley.

[10] Erich Gamma, et al.: *Design patterns: elements of reusable object-oriented software*. Addison Wesley Publishing Company ,1995.

[11] Organization for the Advancement of Structured Information Standards (OASIS) (2007): *Web Services Business Process Execution Language Version 2.0. OASIS Standard*. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[12] Kopp, O.; Binz, T.; Breitenbücher, U.; Leymann, F.: *Winery - A Modeling Tool for TOSCA-based Cloud Applications*. In: ICSOC 2013.

[13] Hyuck Han; Shingyu Kim; Hyungsoo Jung; Yeom, H.Y.; Changho Yoon; Jongwon Park; Yongwoo Lee: *A RESTful Approach to the Management of Cloud Infrastructure*. In: CLOUD '09.

[14] Mietzner, Ralph: *A method and implementation to define and provision variable composite applications, and its usage in cloud computing*. (2010).

[15] Mell, Peter; Grance, Timothy (2011): The NIST Definition of Cloud Computing (Draft). http://www.nist.gov/itl/cloud/.

[16] Binz, T., Breitenbücher, U., Kopp, O., & Leymann, F. (2014). *TOSCA: Portable Automated Deployment and Management of Cloud Applications*. In Advanced Web Services. Springer New York.

All links were last followed on 14.04.2014.