A Method and Programming Model for Developing Interacting Cloud Applications Based on the TOSCA Standard

Michael Zimmermann, Uwe Breitenbücher, Frank Leymann

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

```
@inproceedings{Zimmermann2018_ProgrammingModel,
  Author    = {Michael Zimmermann and Uwe Breitenb{\"u}cher and Frank Leymann},
  Title     = {{A Method and Programming Model for Developing Interacting Cloud
               Applications Based on the TOSCA Standard}},
  Booktitle = {Enterprise Information Systems},
  Series    = {Lecture Notes in Business Information Processing},
  Volume    = {321},
  Publisher = {Springer International Publishing},
  Pages     = {265--290},
  Month     = {June},
  Year      = {2018}
}
```

A Method and Programming Model for Developing Interacting Cloud Applications Based on the TOSCA Standard

Michael Zimmermann, Uwe Breitenbücher, and Frank Leymann

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

Abstract. Many cloud applications are composed of several interacting components and services. The communication between these components can be enabled, for example, by using standards such as WSDL and the workflow technology. In order to wire these components several endpoints must be exchanged, e. g., the IP addresses of deployed services. However, this exchange of endpoint information is highly dependent on the (i) middleware technologies, (ii) programming languages, and (iii) deployment technology used in a concrete scenario and, thus, increases the complexity of implementing such interacting applications. In this paper, we propose a programming model that eases the implementation of interacting components of automatically deployed TOSCA-based applications. Furthermore, we present a method following our programming model, which describes how such a cloud application can be systematically modeled, developed, and automatically deployed based on the TOSCA standard and how code generation capabilities can be utilized for this. The practical feasibility of the presented approach is validated by a system architecture and a prototypical implementation based on the OpenTOSCA ecosystem. This work is an extension of our previous research we presented at the International Conference on Enterprise Information Systems (ICEIS).

Keywords: Development Method, Programming Model, Orchestration, Interaction, Communication, Automated Deployment, TOSCA

1 Introduction

Cloud computing is of vital importance for realizing modern IT systems by enabling automated deployment and management of applications [18]. Cloud properties, for example, scalability, pay-on-demand pricing, or self-service enables developers building flexible and automated cloud applications. These cloud applications typically consists of multiple components, which need to be able to communicate with each other. Therefore, one of the most important issues from an application developer's perspective is to orchestrate and wire these different components. Also regarding connected sensors and actuators in the field of Internet of Things (IoT), the services and devices need to be wired—often IoT integration middleware technologies are used for this purpose [11].

However, this orchestration and wiring mainly depends on the technologies used to realize the application as well as its components. Thus, dependent on these technologies, different information need to be exchanged during the automated deployment of the overall application in order to enable the communication between the components. For example, consider a component which is hosted on a public cloud platform implementing a graphical user interface that presents data from physical devices, such as measured temperature data. In order to enable this component presenting any data from a device, it needs to be wired with this device measuring the data as well as the software running on this device. Therefore, the endpoint information of the component implementing the graphical interface needs to be exchanged with the devices during the automated deployment of the overall application. Such endpoint information required for the communication between components of such *composite applications* are, for example, URLs of services, IP addresses, and required credentials. [37]

Unfortunately, the exchange of these endpoint information mainly depends on technologies, such as the middleware, the programming languages, and the deployment technology used to implement and deploy the application and its components. Therefore, the exchange of these information requires the usage of custom written code, which limits the portability of the application and increases the complexity of implementing the components. Although available technologies, for example, WSDL [32], service buses [9], or orchestration and deployment technologies such as Docker Compose[1] enable to describe and abstract the communication between different components, the composition of multiple heterogeneous technologies is still an open issue. Furthermore, a programming model easing the implementation of interacting components or a method describing the systematically development of such applications is missing.

In this paper, we tackle these issues by presenting a TOSCA-based programming model and a corresponding development method that ease the implementation of interacting components of automatically deployed applications. The main idea of our approach is to abstract endpoint handling of interacting components by using the identifiers and interface descriptions from TOSCA models and by utilizing a service bus, which is integrated in the deployment runtime. This work is an extension of our previous research [37] presented and published at the *International Conference on Enterprise Information Systems (ICEIS)*. While our previous work already covered the TOSCA-based programming model and the required TOSCA extension, we extend these concepts in this work by a systematic development method that supports developers in applying the programming model. Moreover, we show how code generation capabilities can be used to automate some steps of the method. The practicable feasibility of our approach is validated by providing a system architecture and prototypical implementation following our programming model and supporting the presented method.

Before we present our extension and new contribution in Sect. 7, we first recap our previous research [37] to provide a comprehensive overview: Section 2 discusses different state-of-the-art approaches for automating the orchestration and wiring

[1] https://docs.docker.com/compose/

of components and illustrates the existing problems and limitations we tackle in this work. We introduce the TOSCA standard for modeling and managing cloud applications in Sect. 3. Section 4 presents our TOSCA-based programming model, which enables abstracting the communication between components and the endpoint handling during deployment. Our TOSCA extension to enable the modeling of operations implementing business operations is presented in Sect. 5, which we also adapted in this extended version. In Sect. 6 the corresponding communication concepts implemented as service bus are presented. Section 7 presents the new contribution of this extended paper in the form of a method for modeling and developing TOSCA-based cloud applications following the presented programming model. The validation of our approach by implementing a prototype is presented in Sect. 8. Finally, Sect. 9 discusses related work and Sect. 10 presents our conclusion as well as planned future work.

2 Problem Statement

In this section, different state-of-the-art approaches for automating the orchestration and wiring of components using existing technologies are discussed. Moreover, based on the discussed approaches the problems taking place when utilizing them are discussed, such as the required exchange of endpoint information.

Of course, the issue of automatically wiring components of applications in which all components are deployed and operated using only one technology can be solved by using a single composition technology, such as Docker Compose[2] or Kubernetes[3]: Typically such technologies provide built-in wiring and orchestration capabilities that must be considered when implementing a component. For example, by propagating environment variables to containers or by placing and sharing configuration files, which are used by a component to connect to another one [8]. However, in composite cloud applications consisting of multiple heterogeneous components typically multiple technologies have to be combined, especially if physical devices are involved in IoT scenarios [7]. Unfortunately, this also requires to combine multiple invocation mechanisms, protocols, and endpoint exchange mechanisms. Thus, this leads to custom code binding a component to an invoked component as well as its implementation if no *service bus* [9] or—in case of cyber-physical scenarios—*IoT middleware* [11] is used for abstraction.

Accordingly, for the interaction of (micro)services the calling service needs to know the endpoint of the other service to enable their communication. The service bus concept solves this issue from a communication layer perspective. However, if a concrete target service shall be invoked, at least its unique identifier (*ID* in the following) is required and must be contained in the message sent to the bus. In case of an IoT middleware, such as a message broker, typically the ID of the topic to which a device publishes must be known by sender and receiver. However, the exchange of such IDs is technically similar to the exchange of endpoints of the invoked components, for example, URLs of the deployed components. Thus, an

[2] https://docs.docker.com/compose/
[3] http://kubernetes.io/

appropriate exchange mechanism is required nevertheless which approach is used. Typically such information is required during deployment time of a component to tell it to which other components (or to which service bus) it shall connect[4]. However, a standardized approach for (i) automatically exchanging arbitrary kinds of endpoint information between components which they require to communicate with each other and (ii) exchanging IDs to enable components to invoke a certain component via a service bus is missing. Therefore, typically this kind of information is handled in an application-specific manner during the deployment time of the overall application by using manually created configuration scripts and similar approaches. For instance, if a component is implemented as script, environment variables are typically used to pass this kind of endpoint information. This kind of exchange is used, e.g., in a work of Wettinger et al. enabling the unified invocation of scripts implementing management operations [34]. Furthermore, often configuration files need to be updated, for example, as shown by da Silva et al. in an IoT deployment scenario [28]. However, all these issues are reflected in the implementations of components, thus, limiting the application's portability since the used technologies and their exchange mechanisms need to be considered.

To sum up, despite service-orientation, standards such as WSDL, service buses and the workflow technology, providing common means for enabling the interaction between components, their *automated deployment and wiring* is still a technology-dependent issue. Furthermore, this issue itself highly depends on the used (i) middleware technologies, (ii) programming languages, and (iii) deployment technologies. Thus, it results in an increase of the complexity of implementing components as well as orchestrating them leading to custom written code. The problems occurring when using state-of-the-art wiring approaches, for instance, establishing a direct communication between two components or applying a service bus instead, are illustrated by means of an exemplary IoT-Cloud scenario in the next section.

2.1 Motivating Scenario

In Fig. 1 a typical IoT-Cloud scenario describing the wiring of components is depicted. In the illustrated scenario, the *Python 3 App* running on a Raspberry Pi measures temperature data that shall be sent to the *Java 7 App*, which is responsible for storing and displaying this data. To enable the *Python 3 App* to send the measured temperature data to the *Java 7 App*, after the automated provisioning of all shown components the *Python 3 App* requires additional endpoint information. The figure illustrates two possibilities to connect the components: (i) a direct communication and (ii) a communication via a central service bus. However, both variants require exchanging endpoint information: Either the *Python 3 App* needs to know (i) an endpoint (e.g. an URL) of the *Java 7 App* in case of a direct communication, or (ii) some kind of ID specifying the *Java 7 App* in case of using the service bus. Furthermore, in case of the

[4] This is a general requirement for deploying composite applications. Of course, this does not apply to hard-wired scenarios, which are not the focus of this work.

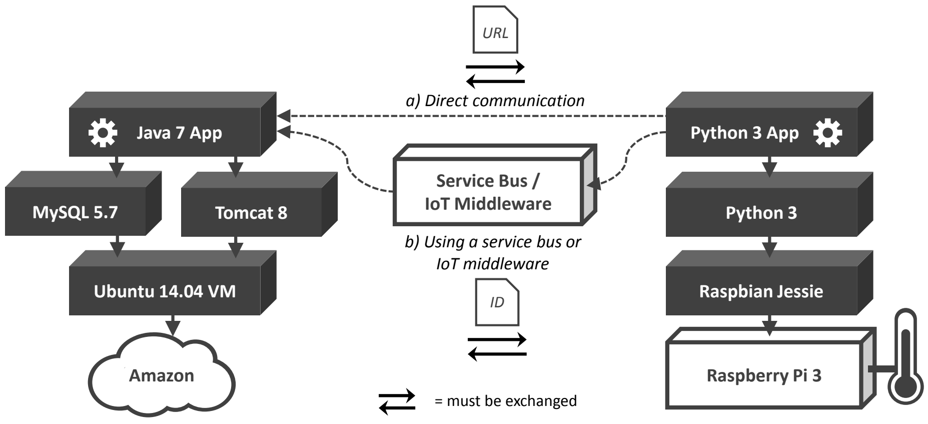

Fig. 1. Two state-of-the-art orchestration variants of an IoT-Cloud scenario [37].

service bus, the *Java 7 App* must be first registered at the bus to make itself known. Even when using a standard such as WS-Addressing[5], some information requires to be exchanged before a connection can be established initially. Thus, resulting in custom code written for each component in order to accomplish the initial exchange of the required endpoint information. However, since this binds the components to the used orchestration technology, in particular, to its endpoint exchange mechanism, this limits the portability of components. Furthermore, because of multiple error sources, additional effort and expertise is required for implementing and debugging components. In order to address these issues, we present a standards-based programming model to abstract the communication between heterogeneous components and proprietary endpoint exchange mechanisms in this paper.

3 The TOSCA Standard

Because the following concepts are based on TOSCA, we introduce the TOSCA standard in this section to provide a comprehensive background. The OASIS standard *Topology and Orchestration Specification for Cloud Applications (TOSCA)* [3, 22, 23] enables to describe the required infrastructure resources, the components, as well as the structure of a cloud application in an interoperable and portable manner. Moreover, TOSCA supports to define the operations required for managing an application. Thus, TOSCA enables the automated provisioning as well as management of cloud applications. The structure of a cloud application is defined in a *topology template*. Figure 2 shows such a template modeling the motivating scenario described in Sect. 2.1 following the visual notation Vino4TOSCA [6]. A topology template is a graph consisting of nodes and directed edges. The nodes of the graph are called *node templates*

[5] https://www.w3.org/TR/ws-addr-core/

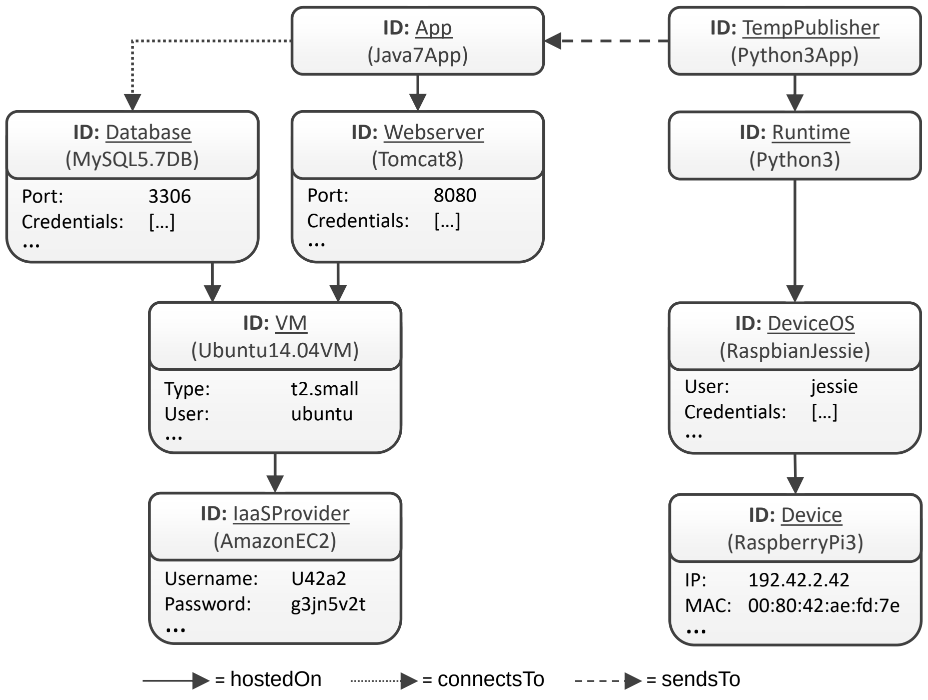

Fig. 2. Exemplary TOSCA Topology Template (Extension of [37]).

and represent components of the application, for example, an Apache Tomcat, a MySQL-Database, a virtual machine, or a cloud provider. Edges connecting the nodes are called *relationship templates* and allow to model the relationships between the components. For example, *"hosted on"* is a relation specifying that a component is hosted on another component, *"depends on"* specifies that a component has dependencies to another component, and *"connects to"* specifies that a component needs to connect to a database, for instance.

In order to support reusability, TOSCA enables the specification of *node types* and *relationship types* defining the semantics of the node and relationship templates. For example, *properties*, such as passwords, user names, or the port of a web server, as well as available *management operations* of a modeled component are defined within the types. Management operations are bundled in *interfaces* and enable the management of components. For example, usually a component node provides an "install" operation in order to install the component, while a hypervisor or cloud provider node typically provides a "createVM" operation in order to create a new virtual machine. The artifacts, which implement the management operations, are called *implementation artifacts* and are implemented, for instance, as a web service packaged as a WAR file or just as a simple SH script. Besides implementation artifacts, additionally TOSCA defines *deployment artifacts* representing the artifacts implementing the business logic of the nodes.

For example, a deployment artifact could be a WAR file implementing the Java application that should be provisioned on the VM of our motivating scenario.

To create or terminate instances of a topology template and to enable the automated management of applications, so-called *management plans* can be specified in TOSCA models. Management plans are executable workflow models implementing a certain management functionality. For example, they define which management operations need to be executed in order to achieve a higher level management goal, such as to provision a new instance of the entire application or to scale out a component. TOSCA does not specify a particular process modeling language for the definition of plans, however, recommends to use a workflow language such as the standardized *Business Process Execution Language (BPEL)* [21] or the *Business Process Model and Notation (BPMN)* [24][6].

Furthermore, TOSCA also specifies a portable and self-contained packaging format, which is called *Cloud Service Archive (CSAR)*. All artifacts, type definitions, the topology template, management plans, as well as all additional files required for automating the provisioning and management are packaged into the CSAR. Such a CSAR can be processed and executed automatically by all standard-compliant *TOSCA Runtime Environments*, such as OpenTOSCA [2], and thus ensuring the application's portability as well as interoperability.

4 TOSCA-based Programming Model

In this section, our TOSCA-based programming model is presented as described in our previous work (Zimmermann et al. [37]). The main goal of the programming model is to completely abstract (i) the communication between components as well as (ii) any endpoint handling during deployment. Therefore, it allows to program the invocation of operations provided by other components in almost the same manner as they would be available locally.

In Fig. 3 the concept of the programming model is illustrated. The upper half of the figure shows a simplified deployment model of the motivating scenario as TOSCA topology template. The left side of the template shows the *Java 7 App* component with ID *App* and its underlying stack, which is hosted on the Amazon cloud. Moreover, the description of the interface *TempManagement* and its operation *updateTemp* to update a temperature value with the input parameter *val* is illustrated. The right side of the template shows the stack of the *Python 3 App* component with ID *TempPublisher*, which shall be hosted on a physical *Raspberry Pi 3*. The main function of the *TempPublisher* component is to send the measured temperature data to the *Java 7 App* component by invoking its operation *updateTemp*. The lower half of the figure illustrates the physical deployment of this template. For example, the temperature sensor connected to the *Raspberry Pi 3* is depicted in this physical deployment view. The left side outlines an exemplary pseudo code implementation of the *updateTemp* operation, while the right side illustrates the simplified implementation of the *TempPublisher*.

[6] We also developed a TOSCA-specific workflow modeling extension called BPMN4TOSCA [14,16] that eases developing management plans.

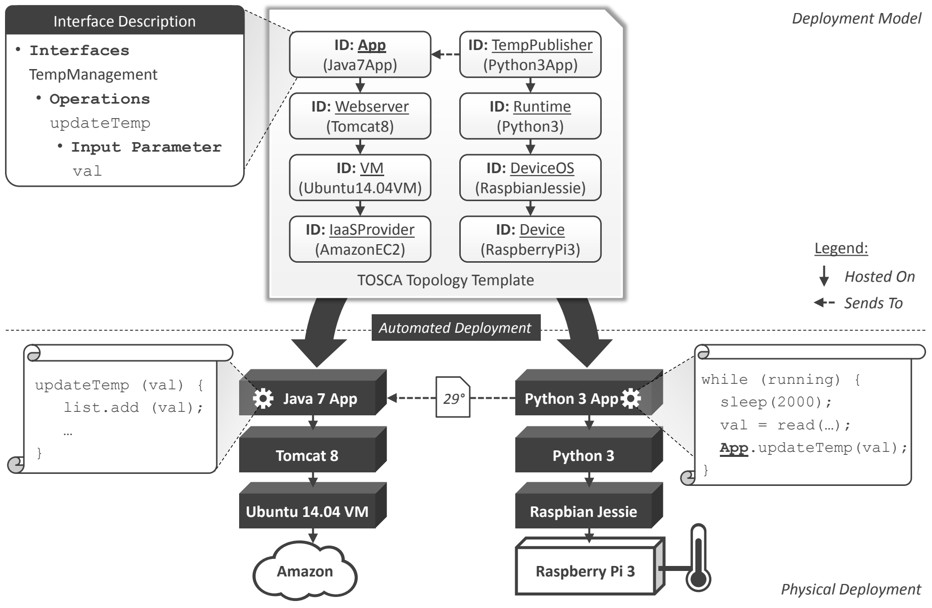

Fig. 3. TOSCA-based Programming Model based on the simplified motivating scenario [37].

The main idea of our proposed TOSCA-based programming model is to enable the invocation of operations offered by other components only based on information contained in the TOSCA topology template. Therefore, to program an invocation, the TOSCA ID of the component to be invoked is used as object in the code while the desired operation is called as usual in object-oriented programming. For instance, the code of the *TempPublisher* component contains the invocation of the operation *updateTemp* of the TOSCA node template having the ID *App* (`App.updateTemp(val)`). Thus, although the component *App* is hosted on the Amazon cloud and the component *TempPublisher* is hosted on a physical device, the operation *updateTemp* can be used within the *TempPublisher* component as it would be a locally available method. Therefore, all relevant wiring aspects are abstracted and no programming for endpoint handling or to connect to a service bus is required. Moreover, all TOSCA IDs of the components are specified within the topology template and are, therefore, well-known. Thus, discovering components and establishing a connection to enable the communication between components requires no exchange of IDs at all.

5 TOSCA Extension for the Programming Model

In order to realize our programming model, the operations implementing the business logic of an application must be defined in the corresponding TOSCA model. However, since this is not supported by TOSCA out of the box, in this section, a TOSCA-extension to define *business operations* of applications modeled using TOSCA is presented. Thus, we extend TOSCA by an *Interface Definition Language (IDL)* for business operations in this section[7]. We first differentiate business operations from management operations, which can be already modeled in TOSCA, in order to ease the understanding.

5.1 Application Interfaces

In Sect. 3 the fundamentals of the TOSCA standard were presented. Moreover, we outlined that management operations provided by node templates are implemented by implementation artifacts. They can be realized using any arbitrary technology such as a simple shell script, a WAR file exposing a web service, or more sophisticated technologies such as Chef recipes [30] or Ansible playbooks [19]. These management operations enable to automate arbitrary management tasks of cloud applications and are orchestrated by management plans. However, nodes of course can also have operations implementing the business logic of the corresponding component. In our motivation scenario, for instance, the Java component *Java 7 App* provides the operation *updateTemp* in order to update the temperature data to be displayed (cf. Sect. 2.1). However, this business operation cannot be modeled using standard TOSCA elements. But since they are required to realize our new programming model, we extend TOSCA node types by a modeling schema for business operations. Listing 1.1 shows how application interfaces and business operations can be defined using the extension.

```
<NodeType name="xs:NCName">
  <ot:ApplicationInterfaces>
    <Interface name="xs:NCName">
      <Operation name="xs:NCName">
        <documentation/> ?
        <InputParameters>
          <InputParameter name="xs:string" type="xs:string"
                          required="yes|no"?/> +
        </InputParameters> ?
        <OutputParameters>
          <OutputParameter name="xs:string" type="xs:string"
                          required="yes|no"?/> +
        </OutputParameters> ?
      </Operation> +
    </Interface> +
  </ot:ApplicationInterfaces> ?
</NodeType>
```

Listing 1.1. TOSCA extension for specifying application interfaces containing business operations.

[7] This section is based on our previous work but extends it by a schematic overview of the extension and a schematic description of the generated code-skeletons [37].

We extended the TOSCA metamodel of node types by an *ApplicationInterfaces* element following the schema of the TOSCA *ManagementInterfaces* element [23]. Thus, within the *ApplicationInterfaces* element, the elements originally specified for defining management operations can be reused: *Operation*, *InputParameter*, and *OutputParameter*. However, an *Operation* contained in an *ApplicationInterfaces* element specifies a business operation and not a management operation. Our extension enables the communication between components contained (i) in one topology template as well as the communication between components contained (ii) in different templates, and thus also enables other applications to utilize the provided operations.

```
class [interfaceName] {

  /**
   * [documentation]
   */
  static [operationName]([InputParameter1],[InputParameter2],...) {
    // TODO generated method stub
    return [OutputParameter]
  }
}
```

Listing 1.2. Abstract generated code-skeleton.

Based on this extension, a code-skeleton (Listing 1.2) can be generated (We detail this in Sect. 6.2). Moreover, we show in Sect. 7.3 an example of such a definition and a generated code-skeleton for the motivation scenario.

5.2 Bindings

In order to technically enable callers, for example a service bus, invoking the specified business operation, binding information are required. Therefore, these information, for example, regarding the invocation style or application-specific invocation properties, need to be specified by the application developer in the TOSCA model of the corresponding operation so that they are available during runtime. The following XML listing (Listing 1.3) shows an example of such binding information based on the motivating scenario, thus, the schema is straight-forward.

```
<ot:ApplicationInterfacesBinding>
  <ot:Endpoint>/TempApp</ot:Endpoint>
  <ot:InvocationType>JSON/REST</ot:InvocationType>
  <ot:ApplicationInterfaceInformations>
    <ot:ApplicationInterfaceInformation name="TempManagement"
        class="org.temp.TempManagement"/>
  </ot:ApplicationInterfaceInformations>
</ot:ApplicationInterfacesBinding>
```

Listing 1.3. Binding information for an application interface [37].

The presented binding information need to be defined in the artifact template referenced by the deployment artifact implementing the business operations, which are defined in an application interface of the corresponding node template. Again, a node template represents a component of an application within a TOSCA topology template, whereas a deployment artifact represents an artifact implementing the business logic of such a component (cf. Sect. 3). For example, a WAR file, which implements the Java application that should be provisioned in the cloud. Therefore, these defined binding information together with our TOSCA extension described in Sect. 5.1 enable specifying the offered business operations of a component as well as how they have to be invoked in detail.

Of course, instead of using such a custom artifact template for binding business operations, accepted standards, such as the Web Services Description Language (WSDL) [32] can also be used to describe the provided functionality of web services. A WSDL file enables to bind the signature of an operation, i. e., the name and the input and output parameters, to information about how this operation can be invoked, such as the endpoint and the supported communication protocol. However, in [34] Wettinger et al. presented a similar TOSCA-based approach to define such binding information within an artifact template for management operations. Thus, for sake of consistency, we decided to additionally support this custom definitions of binding information within an artifact template, too. Therefore, our approach supports both, (i) a binding definition as already used within another TOSCA-based approach as well as (ii) standards such as WSDL. In case of using WSDL, the interface and operations specified within the TOSCA model should correspond to the information defined in the WSDL file. Thus, our presented approach enables to use all the proven and established tooling possibilities for WSDL, for example, automated top-down code generation.

6 System Architecture

Since there was no possibility to define operations implementing business logic using TOSCA without our extension, no tool support exists enabling the communication (i) between components within one TOSCA topology template as well as (ii) between components of different TOSCA topology templates. Thus, in this section, we present a system architecture for TOSCA runtimes that utilizes a service bus supporting our extension of the TOSCA standard. We already presented this system architecture in our previous work [37] and recap it to ease understanding the method introduced in Sect. 7.

6.1 Overview

Figure 4 illustrates our proposed system architecture in a simplified manner, only depicting components of TOSCA runtimes that are required for realizing our new programming model. Of course, several other components are also required, for instance, a component for interpreting the model, etc. A comprehensive overview on different TOSCA runtime architectures can be found in [22].

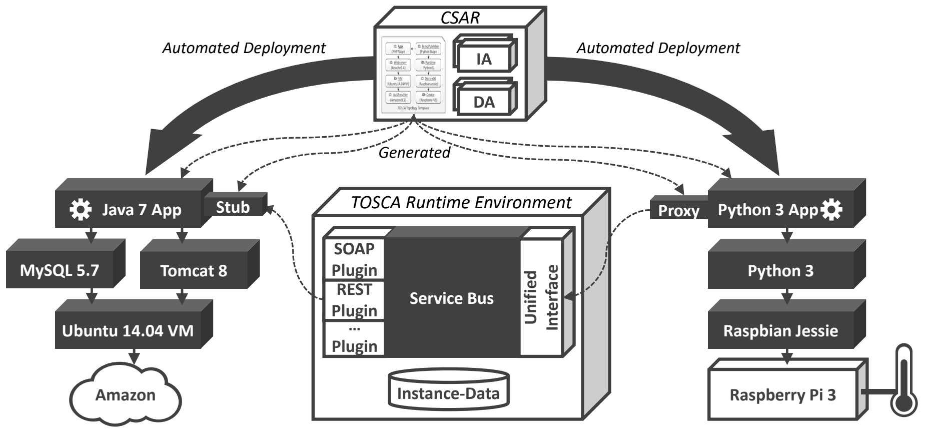

Fig. 4. Simplified system architecture of a TOSCA runtime supporting the presented programming model by a service bus [37].

The central component of the concept is a service bus, which is integrated in the TOSCA runtime[8]. This service bus provides a unified and generic interface for *incoming* invocation requests of business operations provided by components. This interface can be realized, for instance, as RESTful interface supporting synchronous operation invocations with a single HTTP request or asynchronous invocations via resource polling. But also other communication protocols, such as a SOAP interface supporting WS-Addressing [33] or a plugin-based implementation are possible. Depending on the implementation of the interface, also a proxy may be used for the implementation of the component in order to ease the communication with the service bus, e. g., to handle asynchronous callbacks.

In order to support the invocations of different types, such as SOAP/HTTP, the service bus also contains a plugin system for executing *outgoing* invocation requests. For enabling the invocation of the business operations of components, the service bus must determine invocation-relevant properties, for example, the IP address of a deployed component providing the corresponding operation. Thus, the service bus is integrated with other components of the TOSCA runtime in order to be able to access such stored information about application instances, for example, gathered information during the provisioning of the application.

To process incoming messages from the bus, the code implementing the communication part of the invoked component needs to understand these messages. This can be achieved by (i) manually programming against a communication protocol offered by the bus, (ii) using a generic stub for an existing plugin, or (iii) using a *TOSCA Interface Compiler* to generate compatible stubs and proxies out of the TOSCA model. While the first and second possibility require manual implementation, we introduce TOSCA Interface Compilers in the next subsection.

[8] Of course, other kinds of middleware may also be used similarly for realizing our programming model, e. g., a messaging middleware. This is part of our future work.

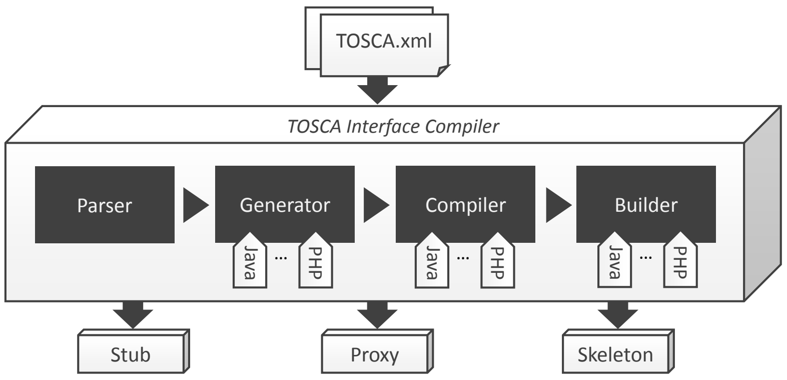

Fig. 5. Overview of the TOSCA Interface Compiler.

6.2 TOSCA Interface Compiler

In order to ease implementing the communication part of a component, our approach enables the generation of a client-proxy and a server-stub able to communicate with the service bus. Since all required information about the business operations, such as their parameters and binding information are contained in the TOSCA model, similarly to generating code out of WSDL files, our approach supports this, too, in order to ease the implementation of interacting components.

The basic architecture of the TOSCA Interface Compiler component is depicted in Fig. 5. As input the TOSCA Interface Compiler gets the TOSCA files defining the operations implementing the business logic of an application. Then, the *Parser* component parses the TOSCA definition files and searches them for containing node types with specified application interfaces and operations. After that, the *Generator* component generates a client-proxy, a server-stub, or a code-skeleton, depending on the users selection. After the generation of the code, depending on the programming language, for example in case of Java the generated code needs to be compiled. This is done in the *Compiler* component. In the last step, again depending on the programming language, the *Builder* component finally builds the artifact to be ready to be provisioned. For example in case of Java, a JAR file is build based on the compiled *.class-files created by the Compiler component of the TOSCA Interface Compiler.

As a result, the TOSCA Interface Compiler assists the developer (i) during the implementation of an application with the generation of code-skeletons of the specified operations (cf. Sect. 7.4) and (ii) to generate a stub and a proxy enabling the communication with the service bus, as it is shown in Fig. 4. Before the generation, the TOSCA Interface Compiler can be customized, for example, to choose the programming language of the component for including required libraries, etc. If a separate WSDL file is referenced instead of using our binding definition (cf. Sect. 5.2), also the top-down approach for code generation using any WSDL tool can be used. Therefore, our approach complements existing code generation tools and enables their efficient usage during development.

7 A Method for Developing and Deploying Interacting TOSCA-based Cloud and IoT Applications

In this section, we present the major new contribution of this extended paper in the form of a method for systematically developing and deploying TOSCA-based applications following the presented programming model. Furthermore, we explain the general advantages of this new method and show how code generation can be utilized in order to speed up the development of TOSCA-based applications.

7.1 Motivation for the Method

Model-driven software development (MDSD) is an important factor in software engineering as it enables, for example, to speed up the development time of an application by generating code [29]. However, for the modeling of complex and distributed cloud or IoT applications, several different models are required. For example, models describing the single components of the application, models defining the relations and dependencies between different components, as well as models describing the required infrastructure resources for the application and its components. Therefore, either the modeler needs to be an expert in all these different model languages and model types, or several modelers with diverse expertise are required. Also, an automation of the entire development and deployment process – from the beginning of the modeling, to the implementation, to the deployment of the application – would be difficult to realize. Furthermore, the orchestration of a cloud application consisting of already existing cloud applications is cumbersome if different model types and languages or no models at all are used. Therefore, our approach allows to model all of the previously described aspects combining the development as well as the deployment steps.

7.2 Overview of the Method

As stated in Sect. 3, the TOSCA standard enables the modeling of the components, the structure, relations and management operations out of the box. As a reminder, the management operation are responsible for the management of a cloud application, for example to install an Apache HTTP Server on a virtual machine. For our approach, we extended the TOSCA standard by so called *application operations*, which are the business operations the cloud application itself implements and provides. Thus, the application operations are not invokable before the cloud application was successfully deployed (with help of the management operations). Altogether, our extension enables the modeling of cloud applications, their structure, the required components and infrastructure nodes, the installation procedure, the offered management as well as application operations uniformly with TOSCA. Thus, our approach not only enables the automated provisioning of the cloud application but also increases the development speed by means of code generation functionalities. Our novel method is composed of five steps to develop a TOSCA-based application and is illustrated in Fig. 6.

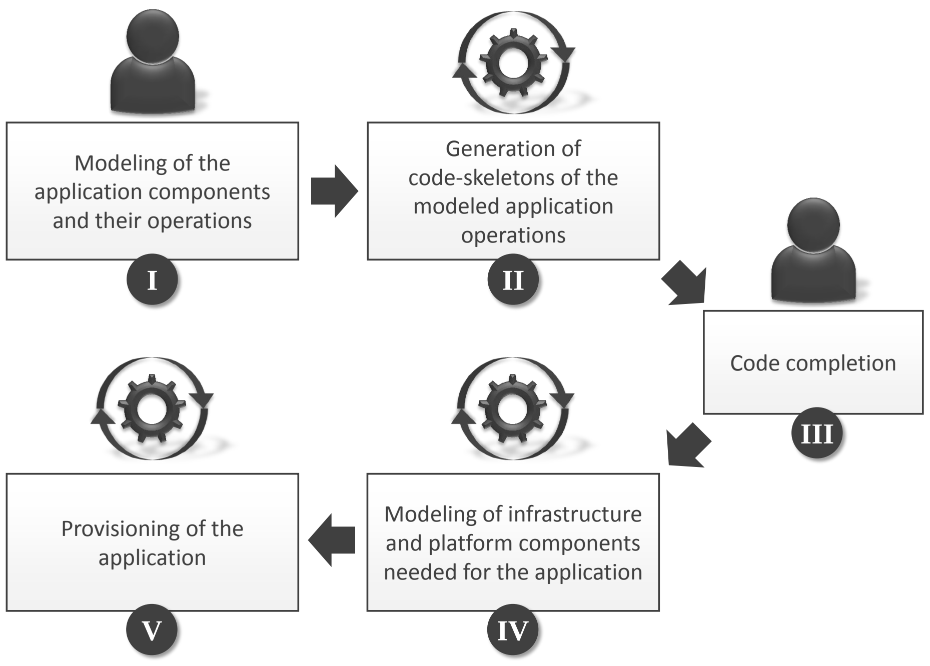
Modeling of the application components and their operations
I
Generation of code-skeletons of the modeled application operations
II
Code completion
III
Modeling of infrastructure and platform components needed for the application
IV
Provisioning of the application
V

Fig. 6. Method for the development of a TOSCA-based cloud application.

In the first step, the components of the application as well as the operations of the application are modeled by the application modeler using TOSCA. In the second step, the code-skeletons based on the previously modeled application operations are generated using the code-skeleton generator. In the third step, the code-skeletons generated in the previous step are completed manually with business logic by the programmer. In the fourth step, the infrastructure and platform components required for the execution of the application are modeled using default TOSCA elements. In the fifth and last step, the modeled and implemented application is deployed automatically by using a TOSCA runtime. The single steps of the method are explained in detail in the following.

7.3 Modeling of the Application Components and their Operations

In the first step, the application modeler models the components of the application as well as the application operations using TOSCA. The components can be modeled using the default TOSCA constructs. Also, already existing TOSCA-based topology models can be reused. Additionally, all application operations implementing business logic, which the application itself should provide, need to be defined within the new *ApplicationInterfaces* element introduced in Sect. 5. Since this is a manual task, using an existing TOSCA modeling tool, like for example Winery [15], can be very helpful for accomplishing this task.

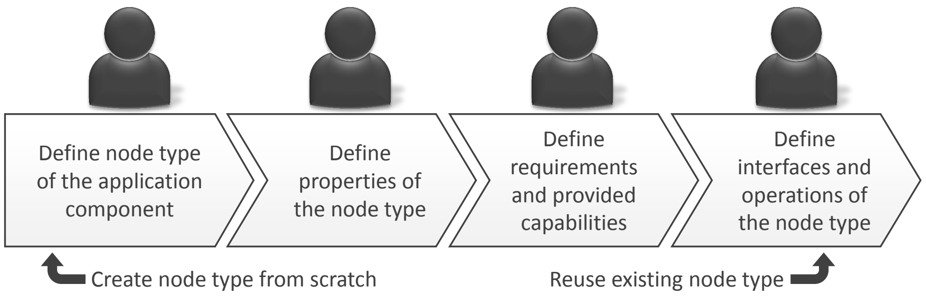
Define node type of the application component
Define properties of the node type
Define requirements and provided capabilities
Define interfaces and operations of the node type
Create node type from scratch
Reuse existing node type

Fig. 7. Required steps for modeling an application component.

In this first step of our method, node type definitions are required to model the topology. For every component of the application, in general the following issues must be considered regarding existing node type definitions that can be reused directly and definitions that must be modified or created from scratch: Either (i) just an existing node type can be reused as it already specifies and implements all required application interfaces, (ii) an available node type already specifies everything required for deployment and only missing application interfaces and binding information must be added, or (iii) the node type needs to be modeled completely from scratch. All three possibilities are depicted in Fig. 7: If a suitable node type already exists, these steps can be ignored. If a complete new node type must be modeled, the modeler first needs to specify the type of the component, for example, that it is a Java application as well as the name of the application. Furthermore, all properties, such as port or password and username needs to be defined for the new component type. Moreover, defining requirements and provided capabilities enables the automatically completion of the application topology model in step 4 (cf. Sect. 7.6). Afterwards, the application interfaces and operations this component should provide are specified using the *ApplicationInterfaces* element. If an already available node type – which is modeled by only using the default TOSCA structures – should be reused in this method, only the interfaces and operations need to be added to the model. Thus, existing node types can be easily reused in the presented method by using the inheritance mechanism of TOSCA that allows to create subtypes of node types that inherit their semantics but allow to extend them.

Listing 1.4 shows the node type of the *Java 7 App* component from the motivation scenario offering the application operation *updateTemp* in order to store and display received temperature data. The temperature value is specified via the input parameter *val*[9]. Based on the shown example defining the provided operation, the input and optional output parameters, as well as documentations, a code-skeleton can be generated in the next step.

[9] Output parameters can be specified the same way

```
<NodeType name="Java7App">
  <ot:ApplicationInterfaces xmlns:ot="http://opentosca.org">
    <Interface name="TempManagement">
      <Operation name="updateTemp">
        <documentation>
          Updates the temperature
        </documentation>
        <InputParameters>
          <InputParameter name="val" type="xs:int"/>
        </InputParameters>
      </Operation>
    </Interface>
  </ot:ApplicationInterfaces>
</NodeType>
```

Listing 1.4. Example of the TOSCA extension for specifying application interfaces containing business operations [37].

7.4 Generation of Code-Skeletons of the Modeled Application Operations

The second step is the generation of code-skeletons based on the previously modeled topology, which includes all node type definitions as well as their application interfaces and operations. Depending on the code-skeleton generator implementation, code for any arbitrary programming language can be generated, for example, as Java or PHP application. Thus, no restriction regarding the used programming language are made for the application developer as our programming model enables an abstract and technology independent modeling of the application components. Since this step can be automated, the developer of the application is supported and, therefore, the development time can be significantly decreased. In Sect. 8, we describe our prototypical implementation of this step that enables the automated generation of code skeletons.

```
class TempManagement {

  /**
   * Updates the temperature
   */
  static void updateTemp(int val) {
    // TODO generated method stub
  }
}
```

Listing 1.5. Generated code-skeleton in Java [37].

```
procedure GenerateCodeSkeleton(ToscaDefinition)
 for every NodeType used in the TopologyTemplate
  for every ApplicationInterface
   if Code-Skeleton must be generated for this ApplicationInterface
    createClass(interfaceName)
    for every ApplicationOperation
     createMethod(operationName, inputParams, outputParams, doc)
    end for
   end if
  end for
 end for
end procedure
```

Listing 1.6. Pseudocode algorithm showing the basic functionality for generating code-skeletons.

Listing 1.5 shows an example of a generated code-skeleton based on the defined application operations presented in Listing 1.4. The basics of the TOSCA Interface Compiler component have been discussed already in Sect. 6.2. For generating the code-skeletons the TOSCA Interface Compiler checks for every found node type if it has defined application interfaces as well as operations. The simplified pseudocode-algorithm for this is shown in Listing 1.6.

Of course, the code-skeleton is generated depending on the defined programming language. Thus, the TOSCA Interface Compiler is based on a plugin-mechanism so that it can be easily extended in order to add support for additional programming languages. Moreover, the TOSCA Interface Compiler also enables the developer to specify for which application interfaces code-skeletons need to be generated as possibly code already exists in the form of a deployment artifact that implements the interface. Especially, if suitable node types are reused without any change in step 1 of the method, no code-skeletons must be generated for these node types that already provide complete implementations. We discuss more details about how this step can be realized in the scope of our prototypical validation based on the OpenTOSCA ecosystem presented in Sect. 8.

7.5 Code Completion

In the third step, the code-skeletons generated in the previously step need to be completed with the business logic. Since at this step business knowledge is needed, this step needs to be done manually by the application developer. However, because of the generated code-skeletons, the code is already prestructured and only the pure business logic needs to be implemented. Furthermore, since also stubs and proxies can be generated abstracting the communication between different components, the programmer can focus completely on the implementation of the business logic and does not need to take care of communication or messaging at all. Of course, this step needs to be repeated for every component a code-skeleton was generated and, thus, should be implemented.

7.6 Modeling of Infrastructure and Platform Components Required for Deploying and Executing the Application

In the fourth step, the infrastructure and platform components required for the execution of the application need to be modeled using the default TOSCA elements. Again, already existing TOSCA-based topology models can be reused in this step. Furthermore, there is other work [35] enabling to generate TOSCA modeling artifacts from various other existing technologies, such as DevOps artifacts, for example, Chef cookbooks or Juju charms. Thus, already proven solutions can be reused in TOSCA again significantly simplifying the development, modelling, and implementation of the required platform and infrastructure components.

Based on the requirements defined in step 1 (cf. Sect. 7.3) the application topology model can be automatically completed by finding existing suitable components providing matching capabilities [13]. For example, a Java application packaged as WAR might define that a web server is required in order to successfully provision it. Thus, if an Apache Tomcat is modeled with defined capabilities matching this requirement, these both components can be connected automatically—for example by using the approach presented by Hirmer et al., which has been integrated in Winery [13]. Likewise this can be done for other components too, for example, that a virtual machine node must be hosted on a infrastructure component such as a hypervisor or a cloud provider. Besides the fully automated completion, also a semi-automatically completion is possible: If several components are matching the requirements of a component, the modeler can decide which one should be used in the model. Also, instead of single components, entire topology fragments can be selected for completing the topology. The topology completion is depicted in Fig. 8. If this modeling task needs to be done manually – since no suitable components can be found automatically – existing modeling tools such as the Winery can support the modeler in this step. However, in this case the modeler needs some domain-specific knowledge about, for example, which components can and should be hosted on which components or what other dependent components are required in the model. Also the fragment-based completion is supported by Winery [36].

7.7 Provisioning of the Application

The fifth and last step is the provisioning of the modeled application with help of a TOSCA runtime environment. In this step, first the modeled infrastructure and platform components, such as virtual machines or web servers, are installed and configured and afterwards the application itself is installed. Since the corresponding topology model was created beforehand, this step only needs to be triggered manually but afterwards runs fully automatically. Thus, by using the integrated service bus, no components need to be configured or connected manually in order to be able to communicate with each other. In Sect. 8, the modeling tool Winery [15], the open-source TOSCA runtime environment OpenTOSCA [2], as well as the self-service portal Vinothek to start the provisioning [5] are introduced, which provide the basis for our prototype described in the next section.

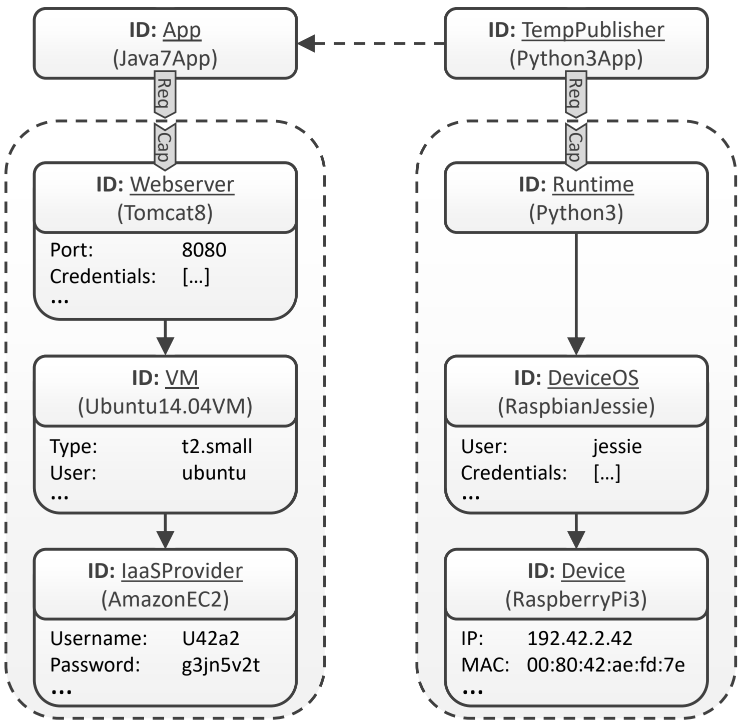

Fig. 8. Automatically completion of the application topology based on specified requirements and capabilities.

8 Validation

In order to validate the practical feasibility of the presented concepts we implemented a prototype, which is integrated in the OpenTOSCA open-source toolchain. Furthermore, in this section, we show how the steps presented in the previous section can be realized using this toolchain.

In order to implement our prototype, we extended the *OpenTOSCA Ecosystem*[10], which consists of: (i) the graphical TOSCA modeling tool *Winery*[11] [15], (ii) the *OpenTOSCA container*[12] [2], and (iii) the self-service portal *Vinothek* [5]. An overview of the ecosystem is depicted in Fig. 9. Using Winery the topology template of the application can be modeled and all files can be packaged into a CSAR. The OpenTOSCA container can use the resulting CSAR as input, interprets the containing files, and deploys the modeled application[13]. The self-service

[10] For testing, instructions to automatically deploy the ecosystem can be found at http://install.opentosca.org

[11] https://projects.eclipse.org/projects/soa.winery

[12] https://www.github.com/OpenTOSCA

[13] Details about this deployment can be found in Breitenbücher et al. [4]

portal Vinothek is used in order to trigger the provisioning of the application. It provides a graphical, web-based end user interface. The tools mentioned in this section are all available as open-source implementations. Therefore, our developed and integrated prototype provides an open-source end-to-end toolchain, which supports the modeling, provisioning, management, orchestration, and communication of TOSCA-based cloud applications. More technical details about the prototypical implementation of the service bus and how the programming model has been realized can be found in the original paper [37]. In the following, we want to focus on how this toolchain supports our new method proposed in this extended work for modeling and developing TOSCA-based applications.

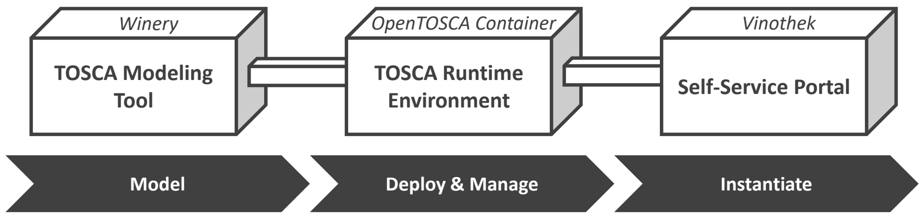

Fig. 9. Overview of the OpenTOSCA Ecosystem.

The first step of the method presented in the previous section is the modeling of the application components and their operations. This step can be realized using Winery, which is a standard-compliant modeling tool for TOSCA. Winery provides a graphical modeling editor for creating Node Types, Relationship Types, and other TOSCA-specified artifacts as well as entire topology models. Furthermore, Winery enables the export of CSARs containing all files required for deploying and managing the modeled application. Thus, Winery supports the modeler of the application components by providing a graphical modeling tool. Therefore, the modeler does not need to write TOSCA definition files in XML himself. For the second step, the generation of code-skeletons of the previously modeled application components, the TOSCA Interface Compiler (cf. Sect. 6.2) can be used. It supports the generation of code-skeletons for different programming languages. For completing the generated code-skeletons any preferred IDE can be used. The fourth step of the method – modeling of the infrastructure and platform components as well as completing the topology – can be realized by using Winery again. It supports the creation of new infrastructure or platform nodes as well as the completion of the application topology manually, semi-automatically, as well as fully automatically based on the concepts of Hirmer et al. [13] and Zimmermann et al. [36]. After that, the final CSAR containing the complete topology model as well as all required artifacts and files can be exported. The last step is the provisioning of the modeled application. Therefore, the exported CSAR can be consumed by the OpenTOSCA Container. After the CSAR was processed successfully by the OpenTOSCA Container, the application is available to be provisioned. In the Vinothek, all installed applications are offered. Therefore, using the Vinothek the application can be selected, required information, for example, credentials for a cloud provider added and the provisioning of a new instance of the modeled application be started.

9 Related Work

In this section, our discussion about related work, which we already discussed partially in Sect. 2 is completed. Regarding the dynamic and flexible invocation of web services, there is different work available [10, 17, 20]. However, their works do not consider topology modeling aspects using standards such as TOSCA. Regarding TOSCA-related work, a concept as well as a prototype enabling the invocation of operations through a unified interface was proposed by Wettinger et al. [34]. However, they only consider the invocation of management operations, which are already supported by the TOSCA standard, and do not consider the invocation of operations implementing the business logic of an application.

In [12] Happ et al. present limitations of the publish-subscribe pattern, for example implemented in the widely accepted IoT protocol MQTT, for the area of IoT. They argue, for example, that a potential publisher of sensor data respectively the used topic can not be easily discovered. Moreover, they argue that the standard is missing details about the messaging reliability. Therefore, this leads to custom solutions and implementations, which results in incompatible applications. Thus, in their work they provide a concept improving the discovery and reliability. However, standards to describe the structure of an application such as TOSCA are not considered in their work.

Occurring problems when integrating different custom components and technologies were already discussed in related work. In [7] Breitenbücher et al. argue, that because most of the available web services and APIs of vendors and cloud providers are not standardized, existing solutions cannot integrate them. Therefore, they provide an approach to integrate provisioning and configuration technologies. However, they do not consider the invocation of business operations through a unified interface, but only focus on management technologies.

In the field of container-based orchestration, there is related work [1, 27, 31] available. They discuss orchestration approaches using containers and advantages using container technologies such as Docker Compose[14], Docker Swarm[15] and Kubernetes[16] in the cloud in general. For example, these technologies enable to transfer and reuse containers between different cloud providers. However, they do not consider the orchestration of non-containerized components.

The general approach of generating a stub from an interface definition for enabling the invocation of a remote method as a local invocation is similar to other approaches such as Java-RMI [26] and CORBA [25]. However, since we use web service technologies, for example, HTTP and XML our approach is agnostic

[14] https://www.docker.com/products/docker-compose
[15] https://www.docker.com/products/docker-swarm
[16] http://kubernetes.io/

regarding the underlying technology. Furthermore, since we use HTTP in our prototype we have no issues with firewalls blocking the traffic.

10 Conclusion

In this paper, we presented a programming model for enabling the unified communication of components of automatically deployed applications (cf. Sect. 4). Therefore, we extended the Topology and Orchestration Specification for Cloud Applications (TOSCA) in order to define business operations of components in a technology-agnostic manner, as presented in (cf. Sect. 5). Moreover, we described a system architecture of an automated deployment and orchestration system utilizing an integrated service bus supporting our programming model (cf. Sect. 6). Furthermore, we presented a generic method for systematically modeling and developing TOSCA-based applications following our proposed programming model (cf. Sect. 7). In order to validate our concepts, we implemented a prototypical service bus and integrated it in the OpenTOSCA toolchain (cf. Sect. 8). Based on the motivation scenario illustrated in Sect. 2.1, we also showed how the proposed method can be applied using the OpenTOSCA toolchain.

In order to support a wider range of IoT scenarios following our programming model, we plan to integrate other middleware components, such as a message broker in future work. Furthermore, in order to improve the performance of our approach, we plan to eliminate the centralized service bus by realizing our approach in a decentralized manner. We also plan in future work to investigate other middleware technologies for enabling and coordinating the communication between components using TOSCA, for example, by utilizing a tuple space.

Acknowledgements

This work was partially funded by the BMWi project *SmartOrchestra* (01MD16001F).

References

1. Bernstein, D.: Containers and Cloud: From LXC to Docker to Kubernetes. IEEE Cloud Computing 1(3), 81–84 (Sep 2014)
2. Binz, T., Breitenbücher, U., Haupt, F., Kopp, O., Leymann, F., Nowak, A., Wagner, S.: OpenTOSCA - A Runtime for TOSCA-based Cloud Applications. In: Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013). pp. 692–695. Springer (Dec 2013)
3. Binz, T., Breitenbücher, U., Kopp, O., Leymann, F.: TOSCA: Portable Automated Deployment and Management of Cloud Applications, pp. 527–549. Advanced Web Services, Springer (Jan 2014)
4. Breitenbücher, U., Binz, T., Képes, K., Kopp, O., Leymann, F., Wettinger, J.: Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA. In: International Conference on Cloud Engineering (IC2E 2014). pp. 87–96. IEEE (Mar 2014)

5. Breitenbücher, U., Binz, T., Kopp, O., Leymann, F.: Vinothek - A Self-Service Portal for TOSCA. In: Proceedings of the 6th Central-European Workshop on Services and their Composition (ZEUS 2014). pp. 69–72. CEUR-WS.org (Feb 2014)
6. Breitenbücher, U., Binz, T., Kopp, O., Leymann, F., Schumm, D.: Vino4TOSCA: A Visual Notation for Application Topologies based on TOSCA. In: On the Move to Meaningful Internet Systems: OTM 2012 (CoopIS 2012). pp. 416–424. Springer (Sep 2012)
7. Breitenbücher, U., Binz, T., Kopp, O., Leymann, F., Wettinger, J.: Integrated Cloud Application Provisioning: Interconnecting Service-Centric and Script-Centric Management Technologies. In: On the Move to Meaningful Internet Systems: OTM 2013 Conferences (CoopIS 2013). pp. 130–148. Springer (Sep 2013)
8. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, Omega, and Kubernetes. Communications of the ACM 59(5), 50–57 (2016)
9. Chappell, D.A.: Enterprise Service Bus. O'Reilly (2004)
10. De Antonellis, V., Melchiori, M., De Santis, L., Mecella, M., Mussi, E., Pernici, B., Plebani, P.: A Layered Architecture for Flexible Web Service Invocation. Software: Practice and Experience 36(2), 191–223 (2006)
11. Guth, J., Breitenbücher, U., Falkenthal, M., Leymann, F., Reinfurt, L.: Comparison of IoT Platform Architectures: A Field Study based on a Reference Architecture. In: Proceedings of the International Conference on Cloudification of the Internet of Things (CIoT 2016). IEEE (Nov 2016)
12. Happ, D., Wolisz, A.: Limitations of the Pub/Sub Pattern for Cloud Based IoT and Their Implications. In: 2nd Int. Conf. on Cloudification of the Internet of Things (CIoT'16). IEEE (November 2016)
13. Hirmer, P., Breitenbücher, U., Binz, T., Leymann, F., et al.: Automatic Topology Completion of TOSCA-based Cloud Applications. In: GI-Jahrestagung, GI, vol. P-251, pp. 247–258. GI (Sep 2014)
14. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F.: BPMN4TOSCA: A Domain-Specific Language to Model Management Plans for Composite Applications. In: Proceedings of the 4th International Workshop on the Business Process Model and Notation (BPMN 2012). pp. 38–52. Springer (Sep 2012)
15. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F.: Winery – A Modeling Tool for TOSCA-based Cloud Applications. In: Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013). pp. 700–704. Springer (Dec 2013)
16. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F., Michelbach, T.: A Domain-Specific Modeling Tool to Model Management Plans for Composite Applications. In: Proceedings of the 7th Central European Workshop on Services and their Composition, ZEUS 2015. pp. 51–54. CEUR Workshop Proceedings (May 2015)
17. Leitner, P., Rosenberg, F., Dustdar, S.: Daios: Efficient Dynamic Web Service Invocation. Internet Computing, IEEE 13(3), 72–80 (2009)
18. Leymann, F.: Cloud Computing: The Next Revolution in IT. In: Proceedings of the 52th Photogrammetric Week. pp. 3–12. Wichmann Verlag (Sep 2009)
19. Mohaan, M., Raithatha, R.: Learning Ansible. Packt Publishing (Nov 2014)
20. Nagano, S., Hasegawa, T., Ohsuga, A., Honiden, S.: Dynamic Invocation Model of Web Services Using Subsumption Relations. In: Proceedings of the International Conference on Web Services (ICWS 2004). pp. 150–156. IEEE (Jul 2004)
21. OASIS: Web Services Business Process Execution Language (WS-BPEL) Version 2.0. Organization for the Advancement of Structured Information Standards (OASIS) (2007)

22. OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0. Organization for the Advancement of Structured Information Standards (OASIS) (2013)
23. OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0. Organization for the Advancement of Structured Information Standards (OASIS) (2013)
24. OMG: Business Process Model and Notation (BPMN) Version 2.0. Object Management Group (OMG) (2011)
25. OMG: CORBA 3.3. Object Management Group (OMG) (2012)
26. Oracle: Java Remote Method Invocation - Distributed Computing for Java. `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html` (2010)
27. Pahl, C.: Containerisation and the PaaS Cloud. IEEE Cloud Computing 2(3), 24–31 (2015)
28. da Silva, A.C.F., Breitenbücher, U., Képes, K., Kopp, O., Leymann, F.: OpenTOSCA for IoT: Automating the Deployment of IoT Applications based on the Mosquitto Message Broker. In: Proceedings of the 6th International Conference on the Internet of Things (IoT 2016). pp. 181–182. ACM (Nov 2016)
29. Stahl, T., Voelter, M., Czarnecki, K.: Model-driven Software Development: Technology, Engineering, Management. John Wiley & Sons (Jul 2006)
30. Taylor, M., Vargo, S.: Learning Chef: A Guide to Configuration Management and Automation. O'Reilly (Nov 2014)
31. Tosatto, A., Ruiu, P., Attanasio, A.: Container-Based Orchestration in Cloud: State of the Art and Challenges. In: Ninth International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS 2015). pp. 70–75. IEEE (Jul 2015)
32. W3C: Web Service Definition Language (WSDL) Version 1.1. World Wide Web Consortium (2001)
33. W3C: Web Services Addressing (WS-Addressing). World Wide Web Consortium (2004)
34. Wettinger, J., Binz, T., Breitenbücher, U., Kopp, O., Leymann, F., Zimmermann, M.: Unified Invocation of Scripts and Services for Provisioning, Deployment, and Management of Cloud Applications Based on TOSCA. In: Proceedings of the 4th International Conference on Cloud Computing and Services Science (CLOSER 2014). pp. 559–568. SciTePress (Apr 2014)
35. Wettinger, J., Breitenbücher, U., Leymann, F.: Standards-based DevOps Automation and Integration Using TOSCA. In: Proceedings of the 7th International Conference on Utility and Cloud Computing (UCC 2014). pp. 59–68. IEEE (Dec 2014)
36. Zimmermann, M., Breitenbücher, U., Falkenthal, M., Leymann, F., Saatkamp, K.: Standards-based Function Shipping - How to use TOSCA for Shipping and Executing Data Analytics Software in Remote Manufacturing Environments. In: Proceedings of the 21th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2017). IEEE (Oct 2017)
37. Zimmermann, M., Breitenbücher, U., Leymann, F.: A TOSCA-based Programming Model for Interacting Components of Automatically Deployed Cloud and IoT Applications. In: Proceedings of the 19th International Conference on Enterprise Information Systems (ICEIS). pp. 121–131. SciTePress (Apr 2017)

All links were last followed on August 7, 2017.