Anything to Topology - A Method and System Architecture to Topologize Technology-specific Application Deployment Artifacts

Christian Endres, Uwe Breitenbücher, Frank Leymann and Johannes Wettinger
Institute of Architecture of Application Systems, University of Stuttgart, Stuttgart, Germany

Keywords: Application Deployment, Topology Crawling, TOSCA, Configuration Management, Chef.

Abstract: In recent years, many application deployment technologies have emerged such as configuration management tools, e.g., Chef and Juju, infrastructure and platform technologies, e.g., Cloud Foundry and OpenStack, as well as container-based approaches, e.g., Docker. As a result, many repositories exist which contain executable and heavily used artifacts that can be used with these technologies, e.g., to deploy a WordPress application. However, to automate the deployment of more complex applications, typically, multiple of these technologies have to be used in combination. Thus, often, diverse artifacts stored in different repositories need to be integrated. This requires expertise about each technology and leads to a manual, complex, and error-prone integration step. In this paper, we tackle these issues: We present a method and system architecture that enables crawling repositories in order to transform the contained artifacts into technology-agnostic topology models, each describing the components that get installed as well as their dependencies. We show how these topologies can be combined to model the deployment of complex applications and how the resulting topology can be deployed automatically by one runtime. To prove the feasibility, we developed and evaluated a prototype based on the TOSCA standard and conducted a case study for Chef artifacts.

1 INTRODUCTION

In recent years, Cloud Computing gained a lot of attention as it helps to achieve flexible IT operation (Leymann, 2009). To automate the deployment of Cloud applications, besides the proprietary APIs offered by providers, many additional technologies have been developed that focus on different kinds of functionality. Among these technologies there are, e.g., several configuration management tools, e.g., Ansible, Chef, Juju, and Puppet; infrastructure and platform technologies, e.g., OpenStack and Cloud Foundry; as well as container-based approaches, e.g., Docker. Due to the heavy usage of these technologies, many open-source repositories have emerged that contain executable and heavily used artifacts that can be used by these technologies to deploy the desired application. For example, the Chef Supermarket[1] contains a plethora of cookbooks that can be used by the Chef runtime *chef-client* (Taylor and Vargo, 2014) to automatically deploy a certain application. Thus, installing, for instance, a WordPress application can be automated efficiently by reusing the cookbook.

[1] https://supermarket.chef.io/

However, reusing such *artifacts* comes with two challenges to be tackled: (i) Selecting appropriate artifacts often requires deep technology-specific expertise to understand the effect of executing an artifact and to configure the runtime correctly. For example, if a Chef cookbook shall be used to deploy WordPress, the cookbook needs to be analyzed to ensure that exactly the desired configuration gets deployed. In addition, the Chef runtime needs to be configured to deploy the application to virtual machine(s). Unfortunately, efficiently getting a quick overview of the components that get installed by an artifact and their dependencies is often not possible without highly specific domain expertise – especially as intuitive graphical tooling is missing in many technologies.

(ii) While understanding artifacts is a serious challenge, combining them to deploy non-trivial applications is another challenge that needs to be tackled in real-world scenarios. For complex applications, typically, multiple management technologies have to be integrated (Breitenbücher et al., 2013): the APIs of Cloud providers must be invoked to deploy virtual machines whereas configuration management technologies, e.g., may be used to deploy the desired components on the provisioned virtual machines.

Endres, C., Breitenbücher, U., Leymann, F. and Wettinger, J.
Anything to Topology - A Method and System Architecture to Topologize Technology-specific Application Deployment Artifacts.
DOI: 10.5220/0006305302080218
In *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*, pages 180-190
ISBN: 978-989-758-243-1

However, such combinations often require enormous expertise when multiple heterogeneous services need to be orchestrated, low-level technologies have to be wrapped, and diverse data formats must be integrated – to name a few challenges (Eilam et al., 2011). Thus, manually executing these steps is error-prone, time-consuming, and, therefore, not efficient (Breitenbücher et al., 2014). In this paper, we tackle these issues by introducing the *Topologize* method.

We present the *Topologize Method and System Architecture* that enables automated crawling of different kinds of repositories, e.g., Chef Supermarket, in order to transform the contained technology-specific artifacts into technology-agnostic topology models. Each generated topology model is a directed, labeled graph describing the components that get installed by a certain artifact as well as the relations between the components. Thus, the generated topology models ease understanding the functionality of artifacts since graphs can be interpreted without requiring any expertise about the employed technology, the artifact, and its serialization format. Moreover, we show that these generated topology models can be combined in a technology-agnostic manner to model the deployment and provisioning of complex applications using a single runtime. Thus, no manual integration of different technologies is required if diverse artifacts needs to be combined. To achieve this, we combine our method and system architecture with the TOSCA standard (OASIS, 2013b) that provides a sophisticated means to integrate arbitrary kinds of management technologies. To validate the practical feasibility of our approach, we developed a prototype that is integrated with the OpenTOSCA Ecosystem (Binz et al., 2013a), a standards-based implementation of the TOSCA standard. Moreover, we conducted a case study based on the configuration management technology Chef to show how the presented architecture and concepts can be applied to a technology.

The remainder of this paper is structured as follows. In Section 2, we motivate our work. In Section 3, we introduce our *Topologize* method enabling to crawl repositories and transform the contained artifacts into technology-agnostic topology models. In Section 4, we present the *Topologize System Architecture* that describes a system capable of automatically executing this method. Section 5 introduces TOSCA. To validate the feasibility of our approach, in Section 6, we describe a prototypical implementation of this system architecture. In Section 7, we describe a case study in which we apply our method to the configuration management technology Chef and evaluate the prototype. Section 8 describes related work, Section 9 concludes the paper and outlines future work.

2 MOTIVATION

Many deployment automation technologies, e.g., configuration management technologies such as Ansible (Mohaan and Raithatha, 2014), Chef (Taylor and Vargo, 2014), or Puppet (Uphill, 2014) come with huge open-source repositories that contain a plethora of artifacts usable for deployment. Typically, these artifacts, e.g., scripts, have to be adapted, deployed, and executed in correct order to install the desired application. The Chef Supermarket is one example that provides cookbooks for installing different kinds of applications, e.g., middleware components or database systems. Another example are GitHub repositories containing source code of applications and scripts for building and deploying the application. Furthermore, the documentation about artifacts and how to execute them is typically available in natural language. However, to ensure achieving the desired deployment and installation, artifacts and their implications must be analyzed and understood in detail to avoid undesired configurations or – in general – undesired results. Unfortunately, correctly interpreting all effects of executing such artifacts typically requires deep technical expertise about the used technology because the mentioned technologies employ different approaches, meta models, and serialization formats.

Especially, the *heterogeneity and diversity of deployment automation technologies* lead to serious integration challenges if multiple technologies have to be combined (Breitenbücher et al., 2013). To achieve this, often the workflow technology is used for orchestration purposes (Arshad et al., 2007; Bellavista et al., 2013; Breitenbücher et al., 2014; Keller and Badonnel, 2004; Mietzner et al., 2009). However, even if an orchestration approach is used for integrating different technologies, nevertheless, (i) the individual artifacts and their effects must be understood to achieve the desired goals, (ii) the orchestration flow must be specified, and (iii) wrappers need to be implemented and configured. In addition, the used runtimes must be installed, maintained, and updated, which typically takes a serious amount of time (Brown and Hellerstein, 2005). Thus, a normalized model is desirable that only describes the desired application and its deployment without the technical details of the technologies used to deploy distinct parts of the model.

For many technologies, these artifacts are files that reference other files, thus, the *files are linked*. However, inspecting all the possible dependencies manually to determine the components that get installed is significantly more error-prone, knowledge-intensive, and time-consuming than having a short look on a structured graphical diagram such as a topology.

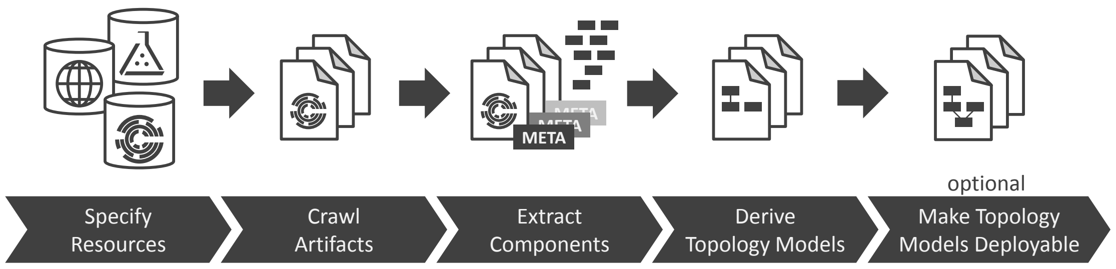


Figure 1: Overview of the Topologize method and its steps.

3 THE TOPOLOGIZE METHOD

In this section, we introduce the *Topologize* method and its five steps. Figure 1 depicts the Topologize method, its steps, and the transformation of crawled, technology-specific artifacts to topologies.

3.1 Goal of the Topologize Method

The goal of the method is to crawl different kinds of repositories, e.g., the Chef Supermarket, in order to transform the contained *technology-specific artifacts* into *technology-agnostic topology models*. The generated topology models (i) support the understanding of artifacts, which is typically not trivial as correctly interpreting the flat and linked files of a certain technology and its serialization format often requires deep technical expertise, cf. Section 2. Moreover, (ii) the generated topology models can be easily combined in a technology-agnostic manner to model the deployment of more complex applications that consist of different building blocks. If, e.g., an artifact describes the automated installation of a database setup while another artifact of a different technology describes the deployment of a Web-frontend, combining the technologies is error-prone and knowledge-intensive contrary to combining only technology-agnostic topology models. We will introduce a system architecture in Section 4 that enables automating the deployment of the resulting topology models by one runtime.

3.2 Step 1: Specify Resources

There are various kinds of repositories available, e.g., many configuration management technology communities, such as of Ansible, Chef, and Puppet, provide open-source repositories that contain various artifacts for the technologies. One example is the Chef Supermarket providing cookbooks for different kinds of applications, middleware components, database systems, and more. In the first step of the method, the repositories are specified that shall be crawled.

3.3 Step 2: Crawl Artifacts

In step two, the specified repositories are crawled for artifacts. Queries may be specified that describe characteristics of the artifacts to be captured, e.g., that only database installation artifacts shall be crawled.

3.4 Step 3: Extract Components

In step three, all artifacts are analyzed in a technology-specific manner to determine the components, their characteristics, and meta information. These components are stored separately in the form of *definitions documents* and meta information, e.g., the license information, are attached to these documents.

3.5 Step 4: Derive Topology Models

In step four, the analyzed artifacts and their meta information are interpreted in a technology-specific manner in order to derive the structure of the application that gets installed by an artifact, i.e., the components that are installed and their relationships. This structure is captured in the form of a topology model, which is a technology-agnostic, directed, labeled graph wherein nodes describe the components and edges their dependencies. The generated topologies serve as normalized models that describe the functionality of artifacts independently of the actual technologies. Thus, they can be understood by non-experts as no expertise about technologies is required. For serializing topology models, the OASIS TOSCA standard (OASIS, 2013b) can be used, for which a visual notation is available (Breitenbücher et al., 2012).

3.6 Step 5: Make Topologies Deployable

In this (optional) step, the topology models are refined to enable their automated deployment if the original artifacts did not include all required information. For example, if a Chef cookbook only describes a database installation, an operating system and virtual machine node gets added to the topology model.

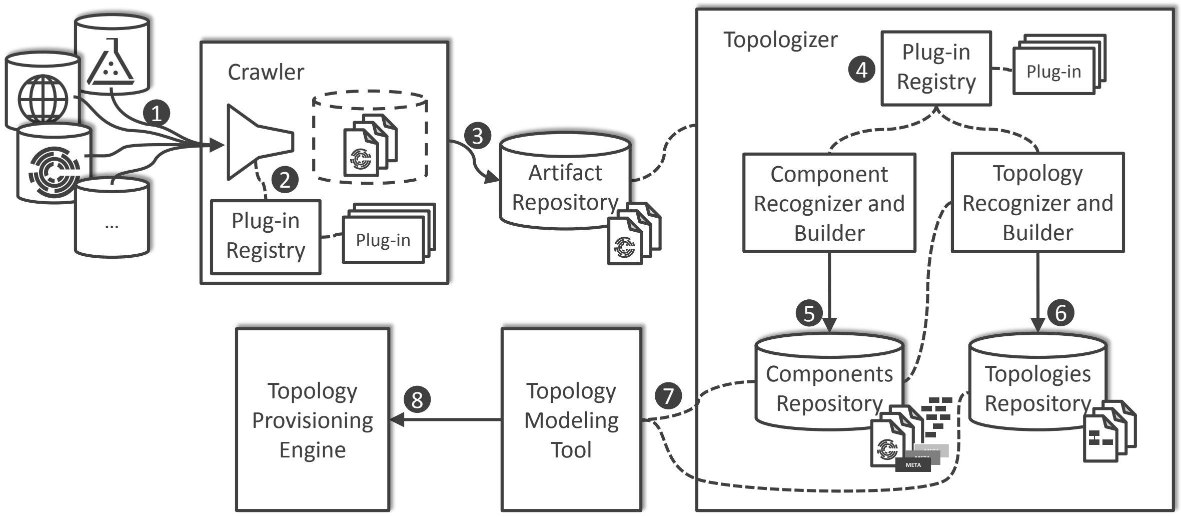


Figure 2: The general system architecture implementing the Topologize method.

4 SYSTEM ARCHITECTURE

In this section, we describe a system architecture for implementing the Topologize method proposed in Section 3. The system architecture is sketched in Figure 2. At the top left, databases are depicted that represent repositories containing technology-specific artifacts. These repositories are (1) specified by stating the resource location and type and registered at the *Crawler*. This corresponds to the first step *Specify Resources* of the Topologize method.

Then, (2) the Crawler discovers and identifies artifacts in the defined repository by executing an appropriate plug-in. Each plug-in comprises the specific functionality to process a location specification, identify contained artifact representations, and obtain the artifacts. Also, meta information about artifacts are gathered, e.g., name and version. Then, (3) each crawled artifact and its information are stored within the *Artifact Repository*. This corresponds to the second step *Crawl Artifacts* of the Topologize method.

The *Topologizer* processes the artifacts, interprets the contained component and structure information, and derives topology models. These models may contain components whose requirements are only partly satisfied. Thus, additional information or additional components may be required for the model being deployable. For being able to analyze technology-specific artifacts, (4) a plug-in mechanism is implemented within the Topologizer. The *Component Recognizer and Builder* (5) analyzes technology-specific artifacts and extracts technology-agnostic component definitions and meta information about them, for example, requirement statements. All components and their information are stored in the *Components Repository*. This corresponds to the third step *Extract Components* of the Topologize method.

The *Topology Recognizer and Builder* (6) interprets component information and infers topology models. The artifacts in the components within the Components Repository expose their requirements, thus, using these information the Topology Recognizer and Builder searches for components within the Components Repository that expose appropriate capabilities. To build topology models, the Topology Recognizer and Builder creates nodes for each component and connects them with edges according to the requirement statements, cf. Section 6. The resulting topology models are stored within the Topology Model Repository. This corresponds to the fourth step *Derive Topology Models* of the Topologize method.

The derived models may not be provisionable because the original artifacts may not state all required information, e.g., credentials, or do not define the target infrastructure, e.g., a specific application server. Thus, (7) a modeling environment enables the user to manually customize the topology model or it completes the topology model in an automated manner. The topology model is packaged and (8) provided to a suitable *Topology Provisioning Engine* for provisioning instances of the modeled application. This corresponds to the fifth step *Make Topology Models Deployable* of the Topologize method. Therefore, automatically maintaining technology-agnostic topology models of technology-specific artifacts is enabled as well as combining them to arbitrary complex topologies using the technology-agnostic modeling tool.

5 THE TOSCA STANDARD

On this page, we provide the fundamentals of the *Topology and Orchestration Specification for Cloud Applications* (TOSCA) (OASIS, 2013a; OASIS, 2013b) that is used in the prototype to serialize topology models, cf. Section 6. For giving a brief and compact background, details are omitted. Details can be found in the specification (OASIS, 2013b), whilst hints for the interpretation of the standard are documented in the *TOSCA Primer* (OASIS, 2013a).

A topology model – called *Topology Template* in TOSCA – is a directed graph in which components (vertices) are connected regarding their relations (edges). The components, e.g., a virtual machine or a web application, are called *Node Templates* and define various characteristics of an instance. The structure of a topology model can be specified by *Relationship Templates* that connect Node Templates pairwise regarding their interplay. For instance, in Figure 3[2] at the left, the Node Template *WebShop* is connected with the Node Template *Apache-v2.4* via a Relationship Template *hostedOn*. Furthermore, semantics are specified by typing the Node Templates and Relationship Templates: *Node* and *Relationship Types*, respectively, define operations, properties, capabilities, and requirements of that type of component or relationship. For instance, the *Ubuntu-v16.04* Node Template references an *UbuntuVM-v16.04* Node Type that defines an *IP-Address* property and that this component is a virtual machine. Thus, for each instance of the *Ubuntu-v16.04* Node Template, its IP-address is set in the modeled property.

For instantiating Node Templates, various actions have to be executed. For installing a *Web Server* Node Type, for example, a script can be executed that invokes *apt-get install*. Such a script is executed on an operating system, thus, the respective Node Template is connected to the operating system Node Template that exposes an operation *execute Script*. Operations enable to create and interact with instances of Node Templates or Relationship Templates, for example, realizing the provisioning of instances. For example, invoking a script execution on a distinct operating system Node Template instance, the operating system instance has to be accessed. By modeling the IP address property as input for the *execute Script* operation, the script invocation execution targets the correct operating system instance. To foster reusability, such operations and properties are modeled in Node Types and Relationship Types. Additionally, Node Types and Relationship Types support inheritance.

[2]We use the visual notation VINO4TOSCA (Breitenbücher et al., 2012) to render topology models.

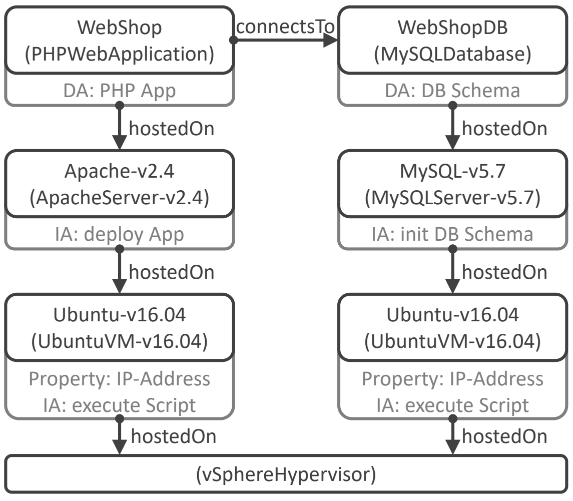


Figure 3: An exemplary TOSCA Topology Template.

For instantiating the modeled functional components, e.g., a customized web application, the implementation of the components has to be provided by the respective Node Template. TOSCA defines such implementations as *Deployment Artifacts* provided by Node Templates or Node Types. These Deployment Artifacts have to be shipped and installed in the target environment of the application to be instantiated.

Contrary, in the former example of the application server, the Node Template can be instantiated by executing a script that invokes *apt-get install*. In this example, the actual implementation of the Node Template is not provided with the model, but is downloaded by apt-get. Such a script is called *Implementation Artifact* because it is not solely deployed, but also executed. An Implementation Artifact can be processed in three flavors: (i) It is executed by the TOSCA Runtime Environment in its own environment, e.g., for accessing an operating system via the *ssh-client*. (ii) The script that executes *apt-get install* is executed in the application's target environment. (iii) Implementation Artifacts that refer, e.g., the interface of a cloud service provider run in their own environment and are called directly. To ship such models and implementations, TOSCA defines the *Cloud Service Archive (CSAR)*. A CSAR is self-contained, contains or refers to all required artifacts, and is portable.

It is important that the Topology Template shown in Figure 3 clearly shows the structure of the application that gets provisioned without showing any details about how this provisioning is technically executed. Consequently, such models can be understood more easily than technology-specific artifacts such as Chef cookbooks. Thus, TOSCA-based topology models provide a suitable abstraction layer for our intend.

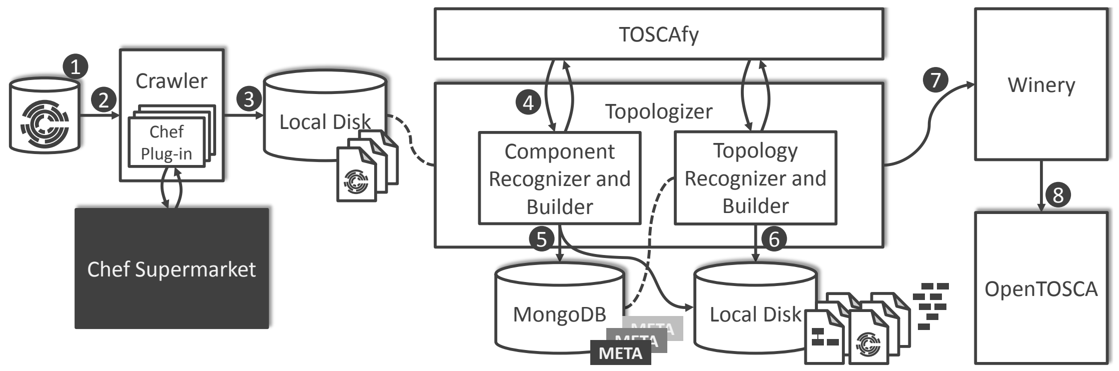


Figure 4: The prototype we implemented to validate the Topologize method.

6 PROTOTYPE

In Section 4, we introduced an architecture for the Topologize method, cf. Section 3. In this section, we describe our prototypical implementation for the Topologize method using TOSCA, cf. Section 5. The prototype is implemented using Java 8[3]. Details about the Chef-specific implementation follow in Section 7.

For the step *Specify Resources*, the location definition (1) comprises the artifact type, location, and access protocol to a repository containing technology-specific artifacts and information. This definition (2) is provided to the Crawler, a Java program that accesses the repository using an appropriate plug-in. The identified artifacts (3) are downloaded to disk and some meta information, for example, name and version are stored. This corresponds to the step *Crawl Artifacts*. We implemented a corresponding plug-in for Chef, details are provided in the next section.

Within the Topologizer, the Component Recognizer and Builder accesses the disk and processes the technology-specific artifacts. Each artifact is unpacked and its contents are analyzed, cf. Section 7. Then, (4) the artifact is transfered to a TOSCA Node Type using *TOSCAfy*[4]. TOSCAfy is an open-source framework for analyzing artifacts and generating TOSCA Node Types. The resulting CSAR (5) contains the Node Type and is downloaded to disk. The artifact is stored as Implementation Artifact at the Node Type. Furthermore, meta data resulting of the analysis (5) are stored in a MongoDB[5] database. MongoDB was chosen because of its abilities to handle documents that are not mapped to a relational scheme, e.g., TOSCA XML and meta data JSON documents. This corresponds to the step *Extract Components* of the Topologize method.

[3]https://www.java.com/

[4]https://github.com/toscafy/

[5]https://www.mongodb.com/

Within the Topologizer, available structure information in the found artifacts are processed by the Topology Recognizer and Builder. For deriving topology models, the exposed requirements are tried to be satisfied with the capabilities of components within Node Types that are found by the Component Recognizer and Builder and located in the Components Repository. Thus, due to this exploration, topology models can be built. This is implemented with an adaption of the depth-first search (Tarjan, 1972).

Duplicate requirements stated by different artifacts within the same topology model are eliminated according (Binz et al., 2013b). This may lead to a topology model that does not reflect exactly the structure of the original artifact, but is necessary for not representing duplicate requirements, e.g., Java, multiple times as a Node Template. Each derived topology model is serialized using TOSCA, cf. Section 5, for example, by using TOSCAfy to define and generate a TOSCA Definitions containing the topology model, all components and relations, and the artifact resources. Finally, the CSAR (6) is stored to the local disk. This corresponds to the step *Derive Topology Models* of the Topologize method.

Often, derived topology models are not provisionable directly because of missing information. Therefore, the CSARs are not only stored locally on disk, but also (8) sent to the TOSCA Modeling Tool Winery[6] enabling the graphical and user-friendly customization of the derived topology models. Thus, a user is enabled to inject credentials and even customize the whole TOSCA Topology Template. Moreover, TOSCA completion algorithms can automatically complete the topology model if components are missing (Hirmer et al., 2014), for example, to inject a Cloud provider Node Template which is typically not described in Chef cookbooks. This corresponds to the step *Make Topology Models Deployable*.

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

Based on the standardized TOSCA metamodel, topology models generated out of technology-specific artifacts can be combined easily to compose more complex applications: the crawled topology models can be used as building blocks in a technology-agnostic manner for the development of new applications on the TOSCA-layer. The graphical modeling tool Winery (Kopp et al., 2013) can be used, e.g., to merge such topology models. Thus, the resulting and merged topology models may contain Implementation Artifacts of different technologies, which is inherently supported by the TOSCA standard. To deploy such merged topology models that contain Node Templates and Types having Implementation Artifacts implemented in different technologies, we developed a plan generator (Breitenbücher et al., 2014) within the OpenTOSCA Runtime Environment (Binz et al., 2013a) that is capable of executing different kinds of artifacts. The generator supports technologies, for example, Chef, Ansible, and Docker. Thus, with our approach, artifacts of different technologies can be combined in a technology-agnostic manner on the TOSCA-layer, while the resulting topology model can be deployed automatically by a single runtime – in this prototype using the OpenTOSCA ecosystem[7].

7 CASE STUDY: CHEF

In Section 6, we introduced a prototype implementing generically the Topologize method. In this section, we provide a case study showing how to apply and implement the method for Chef *cookbooks*.

7.1 Specify the Chef Resources

In the first step, we specify the artifact location and type. The Chef Software Inc. itself provides a publicly available repository for Chef cookbooks: the Chef Supermarket[8]. Therefore, we specified the HTTP API of the Chef Supermarket as location and the artifact type cookbook. For crawling the Chef Supermarket, we implemented a respective plug-in that processes the location specification.

7.2 Crawl for Chef Cookbooks

In the second step, the Crawler searches for artifacts at the specified location. The plug-in for the Chef Supermarket employs the JAX-RS Client API of Jersey[9] to download the cookbooks and retrieve meta information. Most of the information and artifacts were accessible whilst some could not be processed because these cookbooks could not be downloaded.

[7]http://www.opentosca.org
[8]https://supermarket.chef.io/
[9]https://jersey.java.net/

7.3 Extract the Components

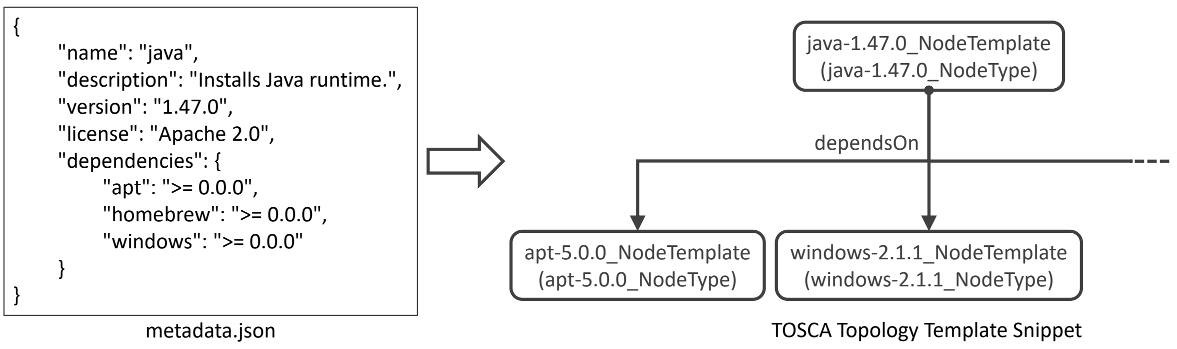
In the third step, the found cookbooks are analyzed to identify components, extracting characteristics, and build TOSCA Node Types. A cookbook states its attributes and templates, includes recipes and files, and provides necessary extensions to the chef-client for enabling it to instantiate the cookbook. Therefore, such a cookbook correlates to a Node Type. The detailed mapping of cookbooks to Node Types can be found in (Wettinger et al., 2014b). For building TOSCA Node Types, we used TOSCAfy.

Besides the transformation to a Node Type, the requirement information of the cookbook have to be extracted. The *chef_version*, *ohai_version*, and *depends* are distinct requirements, but the supported platforms have to be treated differently: a cookbook instance cannot be installed on, e.g., Windows and Linux at once, thus, these requirements have to be treated mutually exclusive. Chef has no cookbooks for installing, e.g., Linux, thus, the platform information cannot be satisfied. These information are needed later for deriving the topology models regarding the capabilities and requirements of the cookbooks.

7.4 Derive Topology Models

In the third step, an extensive repository of TOSCA Node Types is built up containing many cookbooks with requirements. Thus, by traversing transitively all requirements, cf. Section 6, topology models are constructed that represent the requirements graph of the initial cookbook. Using TOSCAfy, the TOSCA Definitions containing the constructed topology model and the cookbooks are packaged to a CSAR.

In Figure 5, a Chef *metadata.json* file is depicted on the left side. This file defines the cookbook's *name*, *version*, some meta information, e.g., its *description* and *license*, and the cookbook's *dependencies* that refer to other cookbooks. In this example, the *java* cookbook defines the cookbooks *apt*, *homebrew*, and *windows* as requirements. Derived from these information, the java component and its resolved requirements are represented as Node Templates in a Topology Template, of which a snippet is depicted on the right. Further resolved requirements of cookbooks are omitted for the sake of brevity.



```
{
    "name": "java",
    "description": "Installs Java runtime.",
    "version": "1.47.0",
    "license": "Apache 2.0",
    "dependencies": {
        "apt": ">= 0.0.0",
        "homebrew": ">= 0.0.0",
        "windows": ">= 0.0.0"
    }
}
```

Figure 5: A Chef *metadata.json* file mapped to a TOSCA Topology Template representation.

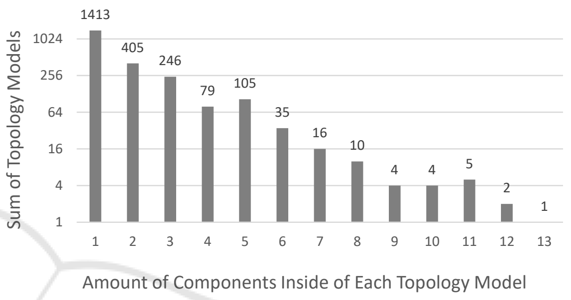


Figure 6: Relation of the amount of Node Templates contained in a topology model to the amount of topologies.

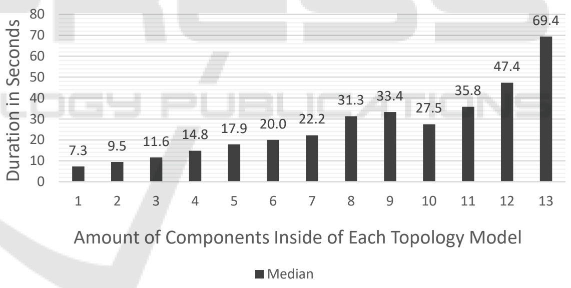


Figure 7: Relation of the amount of Node Templates contained in a topology model to the duration in seconds for analyzing the artifact and constructing the respective topology. The stated durations are result of 10 measurements.

7.5 Make the Topology Models Deployable

The constructed topology models are not provisionable, because Chef presumes a bootstrapped environment that is not installed by Chef itself. Thus, detailed information about the infrastructure layer are missing. Within the OpenTOSCA ecosystem, the fifth step *Make Topology Models Deployable* can be applied by using Winery, cf. Section 6. Using the topology modeler of Winery, the generated topology model can be completed by adding customized infrastructure information. Thus, using the Topologize method, cookbooks can be used to provision applications to not bootstrapped environments. Also, topology models can be composed to arbitrary complex topology models without needing to have expertise of the composed cookbooks. Additionally, Winery serves as TOSCA Repository for the provisioning engine OpenTOSCA that enables automated provisioning.

7.6 Evaluation Results

In this section, we showed how the Topologize method can be applied to Chef. Using our prototype and Topologize method, we crawled 3,191 Chef cookbook files from the Chef Marketplace on the 17th February 2017 and derived 2,325 topology models from the cookbook artifacts. In Figure 6, the size of each topology model, i.e., the amount of the contained components, is related to the amount of topology models derived by the prototype. With a big gap, the most found topology models are singletons whose component either is not stating requirements or the stated requirements are older, not crawled versions or not processable cookbooks that could not be resolved.

In Figure 7, we address the time that it takes to analyze a cookbook for which a topology model shall be constructed and – basing on that knowledge – derive and construct the topology model from the initial cookbook and all requirements. These values result from 10 measurements on Ubuntu 16.04 virtual machines with 8 cores having 3.0GHz and 16 GB RAM. At first glance, a duration of up to 70 seconds for topology models seems to be vast. But, as our goal is having an efficient approach for transforming artifacts into technology-agnostic topologies, the Topologize method is suitable, because the time-costly construction of topology models has to be done only once.

8 RELATED WORK

Cloud Computing enables many benefits, e.g., on-demand provisioning and resource sharing (Leymann, 2009). Whilst some reduce Cloud Computing to merely packaging functionality into a virtual machine or container, in fact, the application usually forms a complex structure that has to provide distinct properties (Leymann et al., 2016). Creating a more detailed view of such architectures can be visualized as a graph of components and their relations, detailed with information about their functional capabilities and requirements, policies of non-functional requirements, interfaces, properties for customization, and definitions about included or referenced resources. Such a graph is called a topology and is defined in the industry-driven standard *Topology and Orchestration Specification for Cloud Applications* (OASIS, 2013a; OASIS, 2013b) and refined in (OASIS, 2015).

TOSCA defines a topology model that describes in detail the structure of an application. A topology consists of components that are related to each other, e.g., a web application is hosted on an application server. The components, relations, and other elements are typed to foster the reusability. Thus, TOSCA enables manifold benefits: the *TOSCA Definitions* may contain a model of components and structures with clear semantics that enables modeling complex applications in a visual encoded way (Breitenbücher et al., 2012) using specific tooling. With Winery, the whole complexity of TOSCA is accessible to human modelers in a visual and guided manner (Kopp et al., 2013). By matching not satisfied requirements of components within the topology with capabilities of other components, topology models can be completed (Hirmer et al., 2014). Thus, TOSCA enables humans to understand and model more easily such complex applications. Additionally, TOSCA enables the automation of management of arbitrary complex applications. For application topologies consisting of components with well-defined provisioning and management functionality, the orchestrated provisioning logic can be generated (Breitenbücher et al., 2014; Eilam et al., 2011). For arbitrary complex and customized applications, e.g., prepared workflows can be provided directly with the Service Template.

Contrary, *Enterprise Topology Graphs* (ETGs) define a formal model describing the structure of running enterprise IT to support tasks, e.g., consolidation, migration, or outsourcing by enabling proven graph algorithms on the model (Binz et al., 2012a; Binz et al., 2012b). Such tasks are time-consuming and error-prone if the underlying ETG has to be created manually. Therefore, such ETGs can be generated by an automated discovery that crawls the running enterprise IT (Binz et al., 2013b). But, ETGs do not enable modeling and inferring automated provisioning and management of application instances.

The OpenTOSCA ecosystem enables interpreting TOSCA models and automated provisioning of the modeled applications (Binz et al., 2013a). Such provisioning can be divided into two opposing approaches: with *imperative provisioning*, all steps necessary for provisioning the application are modeled in full detail, e.g., by using workflow models (Breitenbücher et al., 2013; Keller and Badonnel, 2004; Mietzner et al., 2009). The *declarative provisioning* enables modeling solely the application with its characteristics. Both approaches require a runtime that is able to interpret the models, infer management functionality, and execute it. There are various approaches for automating the provisioning of applications: Ansible, Chef, Planit, and Puppet, to just name some (Arshad et al., 2007; Mohaan and Raithatha, 2014; Taylor and Vargo, 2014; Uphill, 2014). But, all these approaches are domain- and technology-specific, thus, the user needs specific expertise and there is a lack of tooling integrating the heterogeneity of all these technologies.

Although, workflow model based orchestration is a working approach to integrate diverse provisioning technologies, the technologies have to be prepared for being orchestrated beforehand (Wettinger et al., 2014a; Wettinger et al., 2014b). All artifacts have to be provisionable and manageable, thus, have to expose their interfaces. Nevertheless the variety of interfaces, with the Any2API[10] approach the functionality of artifacts can be wrapped with high-level APIs, e.g., RESTful web services (Wettinger et al., 2015). Thus, generating distinct models of all components enables to populate topology models, as shown in this paper.

Subsequent to the Any2API approach, TOSCAfy[11] is a publicly available, open-source framework that provides two major capabilities: (i) retrieving and analyzing existing technology-specific artifacts, e.g., Chef cookbooks and Docker container images to extract and normalize their metadata; (ii) generating portable Cloud Service Archives (CSARs) comprising the artifacts. By using TOSCAfy, CSARs are no longer maintained manually as source artifacts, but they are generated in a repeatable manner. TOSCAfy is implemented using JavaScript based on Node.js. Moreover, it is integrated with Any2API.

But, before applying the aforementioned approaches, artifacts have to be obtained beforehand. The basic idea of conventional Web crawling follows a straightforward process: *“(1) select a URL to crawl,*

[10]http://www.any2api.org

[11]https://github.com/toscafy

(2) fetch and parse page, (3) save the important content, (4) extract URLs from page, (5) add URLs to queue, and (6) repeat" (Matsudaira, 2014). Following this, a broad variety of crawling approaches exist in research and industry. Consequently, implementing a small-scale crawler, e.g., to fetch a distinct set of documents from the Web and store them 'as is' is a mere programming challenge. However, it is not trivial to implement large-scale crawlers that repeatedly fetch large sets of documents to semantically inspect and normalize their content, detect updates, and classify them. Therefore, several research efforts focus on the design of highly scalable and distributed crawlers to improve performance in large-scale crawling scenarios (Boldi et al., 2004; Da Silva et al., 1999; Shkapenyuk and Suel, 2002; Heydon and Najork, 1999; Thelwall, 2001; Edwards et al., 2001). Thus, there are two major categories of crawlers: (i) general-purpose crawlers that fetch and inspect any kind of document, e.g., to populate a search engine or analyze data using mining techniques (Matsudaira, 2014; Thelwall, 2001); (ii) specialized and focused crawlers that only inspect distinct documents (Chakrabarti et al., 1999). Focused crawling is utilized, e.g., to establish a domain-specific knowledge base as it is the purpose of this paper. Therefore, a domain-specific and specialized crawling framework is presented. However, up to now, none of the existing works analyzes application structure information inside of crawled artifacts of configuration management technologies. Therefore, the proposed approach is a novel contribution.

9 CONCLUSION

We presented Topologize that enables (i) crawling technology-specific artifacts, (ii) extracting and abstracting contained component information, and (iii) inferring technology-agnostic topology models that (iv) are provisionable in an automated manner. These models are serialized in TOSCA that enables the modeling and provisioning of complex, technology-specific applications whilst keeping a modular, customizable, and technology-agnostic topology model. The topologies are generated by satisfying dependencies of contained components. Benefits of topologies are (a) a technology-agnostic representation of the technology-specific implications by showing a component's transitive requirements, (b) automated maintaining of such topologies, (c) supporting a user in understanding and selecting artifacts and topology models, and (d) enabling customization and combination of topologies in a technology-agnostic manner.

We validated our Topologize method and architecture by implementing a prototype and applying Topologize to Chef. On the 17th February 2017, we crawled 3,191 Chef cookbooks at the Chef Marketplace and transformed these technology-specific artifacts to technology-agnostic components serialized in TOSCA. We derived 2,325 topologies that comprise the transitive dependencies as far as they could be satisfied. These constructed topologies are of size up to 13 components within one topology model. We evaluated our prototype regarding the analyzing and constructing duration of topologies and packaging as CSAR that took between 7.3 seconds 69.4 seconds depending on the size of the topology.

In this paper, we focused our case study on Chef that is a well-known configuration management system. In the future, we plan to conduct case studies with other technologies, e.g., Docker. Also, for speeding up the analysis and the inferring of topology models, we plan to improve the prototype, e.g., by parallelizing the execution of the depth-first search.

ACKNOWLEDGMENTS

This work was partially funded by the BMWi project *SePiA.Pro* (01MD16013F).

REFERENCES

Arshad, N., Heimbigner, D., and Wolf, A. L. (2007). Deployment and Dynamic Reconfiguration Planning For Distributed Software Systems. *Software Quality Journal*, 15(3).

Bellavista, P., Corradi, A., Foschini, L., and Pernafini, A. (2013). Towards an Automated BPEL-based SaaS Provisioning Support for OpenStack IaaS. *Scalable Computing*, 14(4).

Binz, T., Breitenbücher, U., Haupt, F., Kopp, O., Leymann, F., Nowak, A., and Wagner, S. (2013a). OpenTOSCA - A Runtime for TOSCA-based Cloud Applications. In *Proceedings of the 11th International Conference on Service-Oriented Computing*. Springer.

Binz, T., Breitenbücher, U., Kopp, O., and Leymann, F. (2013b). Automated Discovery and Maintenance of Enterprise Topology Graphs. In *Proceedings of the 6th IEEE International Conference on Service Oriented Computing & Applications*. IEEE.

Binz, T., Fehling, C., Leymann, F., Nowak, A., and Schumm, D. (2012a). Formalizing the Cloud through Enterprise Topology Graphs. In *Proceedings of 2012 IEEE International Conference on Cloud Computing*. IEEE.

Binz, T., Leymann, F., Nowak, A., and Schumm, D. (2012b). Improving the Manageability of Enterprise

Topologies Through Segmentation, Graph Transformation, and Analysis Strategies. In *Proceedings of 2012 Enterprise Distributed Object Computing Conference*. IEEE.

Boldi, P., Codenotti, B., Santini, M., and Vigna, S. (2004). Ubicrawler: A Scalable Fully Distributed Web Crawler. *Software: Practice and Experience*, 34(8).

Breitenbücher, U., Binz, T., Képes, K., Kopp, O., Leymann, F., and Wettinger, J. (2014). Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA. In *International Conference on Cloud Engineering*. IEEE.

Breitenbücher, U., Binz, T., Kopp, O., Leymann, F., and Schumm, D. (2012). Vino4TOSCA: A Visual Notation for Application Topologies based on TOSCA. In *On the Move to Meaningful Internet Systems: OTM 2012*. Springer.

Breitenbücher, U., Binz, T., Kopp, O., Leymann, F., and Wettinger, J. (2013). Integrated Cloud Application Provisioning: Interconnecting Service-Centric and Script-Centric Management Technologies. In *On the Move to Meaningful Internet Systems: OTM 2013 Conferences*. Springer.

Brown, A. B. and Hellerstein, J. L. (2005). Reducing the cost of IT operations: is automation always the answer? In *Proceedings of the 10th Conference on Hot Topics in Operating Systems*. USENIX.

Chakrabarti, S., Van den Berg, M., and Dom, B. (1999). Focused crawling: A new approach to topic-specific web resource discovery. *Computer Networks*, 31(11).

Da Silva, A. S., Veloso, E. A., Golgher, P. B., Ribeiro-Neto, B., Laender, A. H., and Ziviani, N. (1999). Cobweb – a crawler for the brazilian web. In *Proceedings of the String Processing and Information Retrieval Symposium and International Workshop on Groupware*. IEEE.

Edwards, J., McCurley, K., and Tomlin, J. (2001). An adaptive model for optimizing performance of an incremental web crawler. In *Proceedings of the 10th International Conference on World Wide Web*. ACM.

Eilam, T., Elder, M., Konstantinou, A., and Snible, E. (2011). Pattern-based Composite Application Deployment. In *Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management*. IEEE.

Heydon, A. and Najork, M. (1999). Mercator: A Scalable, Extensible Web Crawler. *World Wide Web*, 2(4).

Hirmer, P., Breitenbücher, U., Binz, T., and Leymann, F. (2014). Automatic Topology Completion of TOSCA-based Cloud Applications. In *Proceedings des CloudCycle14 Workshops auf der 44. Jahrestagung der Gesellschaft für Informatik e.V.*

Keller, A. and Badonnel, R. (2004). Automating the Provisioning of Application Services with the BPEL4WS Workflow Language. In *Proceedings of the 15th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*. Springer.

Kopp, O., Binz, T., Breitenbücher, U., and Leymann, F. (2013). Winery – A Modeling Tool for TOSCA-based Cloud Applications. In *Proceedings of the 11th International Conference on Service-Oriented Computing*. Springer.

Leymann, F. (2009). Cloud Computing: The Next Revolution in IT. In *Proceedings of the 52th Photogrammetric Week*. Wichmann Verlag.

Leymann, F., Fehling, C., Wagner, S., and Wettinger, J. (2016). Native Cloud Applications: Why Virtual Machines, Images and Containers Miss the Point! In *Proceedings of the 6th International Conference on Cloud Computing and Service Science*. SciTePress.

Matsudaira, K. (2014). Capturing and structuring data mined from the web. *Communications of the ACM*, 57(3).

Mietzner, R., Unger, T., and Leymann, F. (2009). Cafe: A Generic Configurable Customizable Composite Cloud Application Framework. In *On the Move to Meaningful Internet Systems: OTM 2009*. Springer.

Mohaan, M. and Raithatha, R. (2014). *Learning Ansible*. Packt Publishing.

OASIS (2013a). *Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0*. Organization for the Advancement of Structured Information Standards (OASIS).

OASIS (2013b). *Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0*. Organization for the Advancement of Structured Information Standards (OASIS).

OASIS (2015). *TOSCA Simple Profile in YAML Version 1.0*. Organization for the Advancement of Structured Information Standards (OASIS).

Shkapenyuk, V. and Suel, T. (2002). Design and Implementation of a High-Performance Distributed Web Crawler. In *Proceedings of the 18th International Conference on Data Engineering*. IEEE.

Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2).

Taylor, M. and Vargo, S. (2014). *Learning Chef: A Guide to Configuration Management and Automation*. O'Reilly.

Thelwall, M. (2001). A Web Crawler Design for Data Mining. *Journal of Information Science*, 27(5).

Uphill, T. (2014). *Mastering Puppet*. Packt Publishing.

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

Wettinger, J., Breitenbücher, U., and Leymann, F. (2014b). Standards–based DevOps Automation and Integration Using TOSCA. In *Proceedings of the 7th International Conference on Utility and Cloud Computing*. IEEE.

Wettinger, J., Breitenbücher, U., and Leymann, F. (2015). Any2API - Automated APIfication. In *Proceedings of the 5th International Conference on Cloud Computing and Service Science*. SciTePress.