Automated Generation of Management Workflows for Running Applications by Deriving and Enriching Instance Models

Lukas Harzenetter[1], Tobias Binz[2], Uwe Breitenbücher[1], Frank Leymann[1] and Michael Wurster[1]
[1]*University of Stuttgart, Institute of Architecture of Application Systems, Universitätsstraße 38, 70569 Stuttgart, Germany*
[2]*Robert Bosch GmbH, IoT & Digitalization Architecture, 70442 Stuttgart, Germany*

Keywords: Application Management, Management Automation, Cloud Computing, Workflows, TOSCA.

Abstract: As automation is a key driver to achieve efficiency in the ever growing IT landscape, many different deployment automation technologies arose. These technologies to deploy and manage applications have been widely adopted in industry and research. In larger organizations, usually even multiple deployment technologies are used in parallel. However, as most of these technologies offer limited or no management capabilities, managing application systems deployed using different deployment technologies is cumbersome. Thus, holistic management functionalities affecting multiple components of an application, e. g., update or back up all components, is impossible. In this paper, we present an approach that enables the automated execution of holistic management functionalities for running applications. To achieve this, we first retrieve instance information of a running application and derive a standardized instance model of the application. Afterwards, the instance model is enriched with additional management functionality. We hereby extend the existing Management Feature Enrichment and Workflow Generation approach to support running applications. To execute the enriched management functionalities on the running application, standard-based workflows are generated.

1 INTRODUCTION

The automation of application deployments became very important, especially in the widely adopted area of cloud computing as manually deploying applications is error-prone, cumbersome, and time-consuming (Oppenheimer, 2003). Today, many deployment automation technologies are available, e. g., Kubernetes, Puppet, and AWS CloudFormation. Most deployment technologies use deployment models to deploy applications automatically (Bergmayr et al., 2018). Hereby, deployment models usually describe applications declaratively in the form of their structure, i. e., their components and their relations. For example, a simple web-shop may connect to a database and is provided by a web-server installed on a virtual machine (VM). However, large organizations usually employ multiple deployment automation technologies to manage their wide variety of applications. Hence, each application must be managed using different technologies.

Although some technologies enable the management of individual application components, e. g., scaling the amount of web-servers, they rarely support performing component overarching and more complex management functionalities. Thus, *holistic management* of an application, i. e., management of all its components that may be distributed over multiple environments, is a major challenge and mostly not supported (Harzenetter et al., 2019b). An example for a holistic management functionality is the creation of backups of all stateful components that are distributed over multiple cloud providers or performing rolling updates on all components. Thus, to enable holistic management functionalities, operators must develop custom automations for each application and employed runtime technologies. This requires deep technical knowledge, expertise in different kinds of technologies, and is very time-consuming. To tackle this, we introduced an approach to automatically enrich deployment models with holistic and custom management functionalities which can be executed by generated workflows (Harzenetter et al., 2019b). For example, if a deployment model is enriched with a backup functionality, a workflow is generated that executes the backup operations of all stateful components.

The *Management Feature Enrichment and Workflow Generation* approach (Harzenetter et al., 2019b) is able to enrich applications before their deployment with additional management functionality by process-

Harzenetter, L., Binz, T., Breitenbücher, U., Leymann, F. and Wurster, M.
Automated Generation of Management Workflows for Running Applications by Deriving and Enriching Instance Models.
DOI: 10.5220/0010477900990110
In *Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021)*, pages 99-110
ISBN: 978-989-758-510-4

ing its declarative deployment model and generating executable management workflows. However, it is not possible to enrich *running* applications with additional management functionalities. Moreover, by performing management functionalities on a running application, the application's state may change, e. g., by installing security updates. Since most deployment technologies also monitor the applications they have deployed, they may detect performed changes and revert them. Hence, the main research questions (RQ) we are resolving are:

RQ 1. *How can running applications be enriched with additional, holistic management functionalities that are not supported by the used deployment technology?*

We are addressing RQ 1 by splitting it into the following questions tackling the two main challenges:

RQ 1.1. *How can a normalized, technology-independent, and typed instance model of a running application be retrieved automatically from a specific used deployment technology?*

RQ 1.2. *How can the interference of the underlying deployment technology be avoided?*

To resolve RQ 1.1, we introduce new concepts to (i) retrieve instance information about the components of a running application from its underlying deployment technology, and to (ii) derive a normalized instance model of the application by interpreting the retrieved component information and mapping them to normalized component types. Hence, by analyzing the generated instance model, we are able to tackle RQ 1.2 by (iii) extending the existing Management Feature Enrichment and Workflow Generation approach to support the enrichment of management functionality such that it also considers the underlying deployment technology the application was originally deployed with. As a result, management workflows can be generated for each management functionality that can be executed on the running application, enabling the management of all deployed applications in a company from a single dashboard.

However, the instance information provided by the deployment technologies is often insufficient to derive and perform arbitrary management functionalities for an application. For example, component-specific properties, such as the web application context, cannot always be identified based on the deployment technology-specific instance information. Therefore, we are additionally tackling RQ 2:

RQ 2. *How can a technology-independent instance model of a running application be refined to represent details about the application?*

To address RQ 2, we are adapting and extending our previous work (Binz et al., 2013) to automatically refine identified component types and retrieve component-specific runtime properties to create a detailed instance model of a running application.

Hereafter, Sect. 2 provides background followed by the presentation of our approach in Sect. 3. Sect. 4 outlines the prototype and presents a case study, while Sect. 5 discusses the approach. Finally, Sect. 6 discusses related work and Sect. 7 concludes the paper.

2 FUNDAMENTALS, PREVIOUS WORK, AND PROBLEM STATEMENT

This section introduces fundamental terms, describes existing work, and discusses its current limitations.

2.1 Deployment Automation

The automation of application deployments is of vital importance, as manual deployments are cumbersome and error-prone (Oppenheimer, 2003). To describe an application deployment, *deployment models* which can be classified into *imperative* and *declarative* models (Endres et al., 2017) are typically used (Bergmayr et al., 2018). Imperative deployment models define the exact order of operations that must be executed to deploy an application, e. g., in the form of a script or workflow. In contrast, declarative deployment models describe the structure of an application by modeling its components, their relations, and their configuration—in general as directed and weighted graphs (Wurster et al., 2019). Thus, a deployment technology must interpret the declarative model and derive the necessary steps to deploy the application including all modelled components and relations.

A standardized language that combines declarative and imperative models is the *Topology and Orchestration Specification for Cloud Applications* (TOSCA) (OASIS, 2013, 2020). TOSCA is an actively maintained standard by OASIS and provides a vendor- and technology agnostic metamodel to deploy and manage applications. In TOSCA, the deployment of an application can be modeled (i) declaratively in the form of *Topology Templates*, and (ii) as imperative *Plans*, i. e., executable workflow models, that are defined by languages such as BPEL (OASIS,

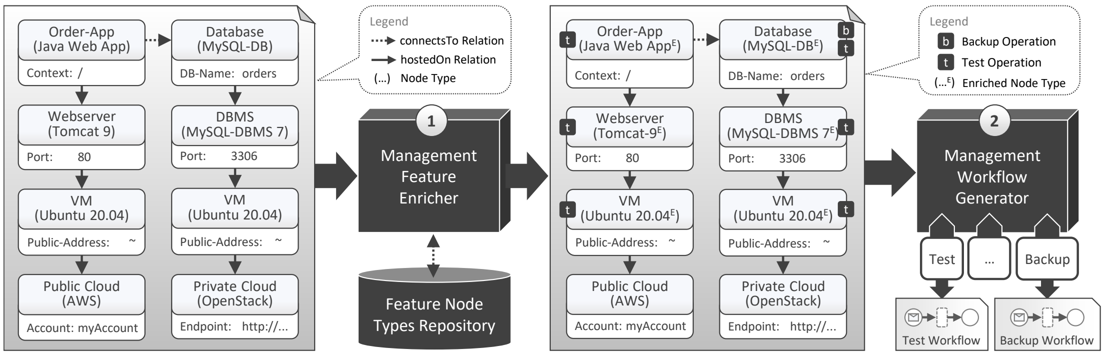

Figure 1: The Management Feature Enrichment and Workflow Generation approach (Harzenetter et al., 2019b).

2007). Since imperative models can be derived from declarative ones automatically (Breitenbücher et al., 2014), we focus on declarative models. To model applications in TOSCA, they are declaratively defined in *Topology Templates* which are directed graphs in which nodes are represented by *Node Templates* and edges are *Relationship Templates*. TOSCA defines a typing system which makes it ontologically extensible (Bergmayr et al., 2018) in the form of *Node Types* and *Relationship Types*. These types define a certain semantic by using *Properties*, *Interfaces*, and their corresponding *Operations*. For example, consider the application shown on the left of Fig. 1: Here, an "Order App" of type *Java Web App* is running on a *Tomcat* web server that is installed on a *Ubuntu* VM running on AWS. Additionally, to store its data, the Order App connects to a database of type *MySQL-DB* which is managed by a database management system (DBMS) of type *MySQL DBMS*. The DBMS is also installed on a Ubuntu VM which is, however, running in an *OpenStack* infrastructure. To express that, e. g., the web server is installed on the VM, the two node templates are connected by a relationship template which is an instance of the relation type *Hosted-On*. Further, the Tomcat node type, for example, defines a *Port* property to configure the port the web server instance should be listening to. Lastly, the whole application is packaged in a *Cloud Service Archive* (CSAR) that contains all necessary elements like node and relationship types, as well as executables, such as install scripts, to deploy the modeled application.

2.2 Management Feature Enrichment and Workflow Generation Method

To enrich applications with additional management functionality, we previously introduced the *Management Feature Enrichment and Workflow Generation* (MFEW) method (Harzenetter et al., 2019b).

2.2.1 Overview of the Management Feature Enrichment and Workflow Generation

The Management Feature Enrichment and Workflow Generation method is illustrated in Fig. 1. In the first step, a declaratively modeled application is passed to the *Management Feature Enricher* component which analyzes the application's components based on their types. It hereby searches through a repository containing additional management functionalities that are realized in the form of *Feature Node Types* that define specialized management operations. For example, one feature node type may provide an operation to backup a MySQL database, while another may offer operations to test the availability and accessibility of Ubuntu VMs. Thus, if a user wants to have additional backup and test functionalities, all node types in the given application are enriched with the corresponding operations, if they are available in the repository, by generating so-called *Enriched Node Types* that replace the current types in the topology template.

In the second step, management workflows, i. e., TOSCA management plans, are automatically derived by a *Management Workflow Generator*: For each enriched management functionality, there is a dedicated workflow generation plugin that generates a workflow that executes the corresponding operations on an application. Thus, a Test workflow and a Backup workflow are generated for the application shown in Fig. 1.

2.2.2 Limitations of the Current Method

The current MFEW method is based on declarative deployment models (Harzenetter et al., 2019b). Thus, it only works for not-deployed applications. In order to apply the idea of subsequently adding management functionality also to running applications, we transfer the general idea of our previous work to instance models in this paper. Since instance models of running applications can also be represented as directed and weighted graphs (Binz et al., 2012; Wurster et al., 2019), the MFEW approach provides the basic foundation. However, two major extensions and adaptations are required: (i) Instance models must be retrieved and normalized to correctly identify the node types of the components to enable the enrichment of management functionalities (cf. RQ 1.1). Therefore, a concept is needed that is capable of transforming deployment technology-specific instance models to normalized, i. e., technology-agnostic, instance models. (ii) To enrich instance models with additional management functionalities, the identification of available management functionalities requires a new concept: If an operation is changing the application's state, the underlying deployment technology may revert the performed changes and restore the previous state. Hence, the available functionalities must be selected carefully and the implementations must notify the deployment technology to avoid its interference. Therefore, we are distinguishing between two different kinds of management functionalities, *state-changing* and *state-preserving* management functionalities. State-changing functionalities change the applications state, e. g., changing the configuration of components or adding and removing components; state-preserving functionalities only interact with the components. For example, installing updates is a state-changing management functionality, while retrieving a backups is a state-preserving functionality.

2.3 Crawling Instance Models

In previous work, we presented an iterative approach to automatically derive instance models of enterprise applications (Binz et al., 2013). To achieve this, we introduced a plugin-based crawler that is able to identify the components and their corresponding types of an application. These technology-specific plugins are able to perform all kinds of operations in order to derive and refine an instance model of a given application (Binz et al., 2013). For example, a plugin may send HTTP requests to determine a specific type of web server based on the response's headers, while another plugin may be able to log in to a VM via SSH to identify files or processes that indicate a specific running component. However, as we did not consider the underlying deployment technology, the generated models cannot be used to manage the applications as management functionalities may change the state of the application, e. g., by installing security updates. Hence, the deployment technology may interfere and revert the performed changes. Thus, we cannot use the crawling approach as we have to consider the dependencies to the deployment technology when performing state-changing management functionalities.

3 MANAGING RUNNING APPLICATIONS BY GENERATING WORKFLOWS

To enable the enrichment of management functionalities for running applications, a model of the application is required. This model must be retrieved from the running application and normalized to describe the components of the application in a technology-independent way. An overview of the proposed approach is illustrated in Fig. 2: The first, new step retrieves the deployment technology-specific instance information about the running application that should be manageable. Thus, plugins of the *Instance Information Retriever* component (1) access the APIs of the deployment technologies used to deploy the application. Second, the new *Instance Model Normalizer* component (2) interprets the gathered technology-specific information and derives a normalized instance model of the application. In this step, the information about the deployed components are mapped to known and normalized node types extending the normative TOSCA node types (OASIS, 2020) by concrete technologies such as Ubuntu, MySQL, or Tomcat. However, since the deployment technologies do not always hold all necessary information, the normalized instance model must be completed by a dedicated *Instance Model Completer* component (3) in the third step. For example, Kubernetes only knows containers and their configuration that can be passed from the outside but does not have information about the components inside, while Puppet usually knows all installed and configured components but may not hold all required properties about them. Fourth, in the extended *Management Feature Enricher* component (4), available management functionality is added to the normalized instance model. As described in Sect. 2.2.2, the identification of available management functionalities must consider the underlying deployment technology. Finally, the enriched instance model is interpreted by the *Management Workflow Generator* component (5) to derive workflows for each enriched management functionality that can be executed on a suitable *Workflow Engine* (6).

3.1 Instance Information Retriever

In the first step, the new *Instance Information Retriever* component retrieves the technology-specific instance models from deployment technologies using specialized plugins as depicted in Fig. 2. When using existing approaches, e. g., crawling (Binz et al., 2013) and network scanning (Holm et al., 2014), to retrieve instance information about running applica-

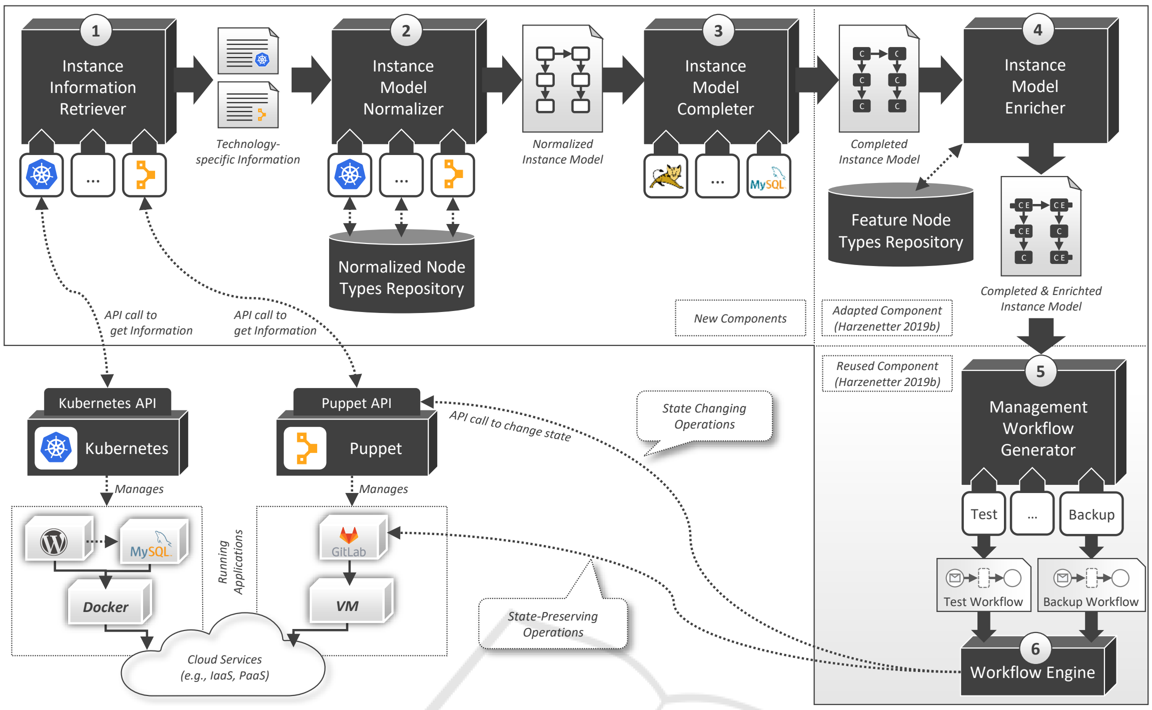
1
Instance Information Retriever
Technology-specific Information
2
Instance Model Normalizer
Normalized Instance Model
3
Instance Model Completer
Completed Instance Model
4
Instance Model Enricher
Normalized Node Types Repository
Feature Node Types Repository
Completed & Enriched Instance Model
New Components
Adapted Component (Harzenetter 2019b)
Reused Component (Harzenetter 2019b)
API call to get Information
API call to get Information
Kubernetes API
Kubernetes
Puppet API
Puppet
API call to change state
State Changing Operations
Manages
Manages
Running Applications
Docker
VM
GitLab
MySQL
Cloud Services (e.g., IaaS, PaaS)
State-Preserving Operations
5
Management Workflow Generator
Test
...
Backup
Test Workflow
Backup Workflow
6
Workflow Engine

Figure 2: Overview of the approach to generate management workflows for running applications.

tions, a major drawback arises if the generated models are used to manage the applications: The models do not contain any information about the deployment technologies used to deploy the applications. Thus, if a management functionality is changing the state of the application, e. g., installing security updates, most deployment technology recognize the changes and try to revert them to the previous state. To avoid this, we retrieve the information about running applications from the API of the respective deployment technology using plugins and add the information as annotations to the derived model. Hence, state-changing management functionalities are able to notify the deployment technology about state changes using the annotated information and, thus, can avoid their interference.

Depending on the deployment technology, the information about the components of an application that can be accessed varies in its expressiveness. For example, while it is possible to retrieve information about concrete software components that are installed on a specific machine from Chef or Puppet, Terraform and Kubernetes do not hold such information as they focus on managing cloud resources, computing infrastructure, and containers. Therefore, the granularity level of the instance data retrieved from the deployment technologies differs from infrastructure components, i. e., compute instances, such as VMs, to actual software components, such as applications and middleware, hosted on these compute instances.

3.2 Instance Model Normalizer

Based on the data retrieved in step one, a normalized model of the application is derived in the second step using the new *Instance Model Normalizer*. As this data is highly specific to the deployment technology it has been retrieved from, custom logic that is able to interpret the technology-specific data is required for further processing. Hence, plugins for the respective deployment technologies must be able to understand the retrieved instance data and generate normalized instance models. Moreover, the generated instance models must also contain all necessary information about the underlying deployment technology. For example, how it can be accessed, as well as technology-specific IDs that identify the components inside the deployment technology-specific instance model.

Since TOSCA defines a vendor and technology agnostic, as well as ontologically extensible meta model (Bergmayr et al., 2018), we use it to describe application instances. Thus, the technology-specific components are represented as node templates which must be mapped to node types defining their semantics. For example, if an instance of a Ubuntu operating system (OS) is found to be running as a VM, it can be mapped to a node template that is an instance of the normalized *Ubuntu* node type. These normalized node types can be defined in a repository to specify the semantics of components in a standardized and normalized manner. Additionally, the normalized node types may define abstract definitions to describe the overall semantics and explicit versions. For example, a *Ubuntu* node type defines that the VM is running a Ubuntu OS, while a *Ubuntu 20.04* node type refines this information with a specific version. However, there are cases in which the deployment technology-specific information is not sufficient enough to identify a component's type. In this case, the generic *TOSCA Normative Node Types* (OASIS, 2020) are used to represent the identified components. For example, TOSCA defines a generic *Software Component* node type which can be used to represent any kind of software component. Similarly, the generic *Compute* node type is available to represent computing resources such as VMs or containers if a more concrete node type cannot be identified.

Additionally, the instance model normalizer must map component specific properties from deployment technology-specific information to the normalized representation in TOSCA. Thus, as node types define the available properties their node templates have, four kinds of mappings between instance properties and properties of the node type, can be differentiated: (i) A property is named equally. In most cases, the mapping is straightforward. However, the contents may be semantically different and a more complex mapping may be required (cf. case iii). (ii) The name of a property is different but they are describing the same element. For example, a property of a VM in the retrieved instance information is named "IP-address", while the *Compute* node type defines it as "public-address". Hence, the mapping is usually straightforward. (iii) A property consists of two or more properties in one model, while it is combined into one in the other model. This may be the case, if a web component has only a property called "url" while the corresponding node type defines separate properties for the "hostname", "port", and "context-path". Thus, a more complex property mapping is required. (iv) Lastly, there may be properties which cannot be mapped. For example, if a property cannot be mapped to a property of a node type, it can be saved in the model as additional metadata to enable manual refinement of the component, e. g., to a custom node type that could not be identified automatically. In contrast, if a property is not available in the retrieved instance data, it cannot be filled and, thus, will remain empty.

3.3 Instance Model Completer

In the third, new *Instance Model Completer* component, the normalized instance model is refined with detailed information about component types and property values. Since the instance information that are accessible from the deployment technologies may not produce a complete model of an application, we introduce the *Instance Model Completer* as a third step. The instance model completer is based on our previously introduced plugin-based crawler (Binz et al., 2013). Hereby, plugins identify components of an application, retrieve their configuration, or refine their types. Then, these component-specific information are used to improve the derived instance model. For example, a Tomcat plugin is able to identify whether a Tomcat web server is running on a VM or container, refine an already identified Tomcat component to a specific version, and identify the port it is listening to. Additionally, plugins may also be able to refine the properties or types of multiple components. For example, as a Tomcat web server provides access to applications via the internet, it may be able to identify the context in which a particular web application is accessible. Thus, we extended our concepts by a sub-graph mechanism to identify plugins that are able to refine the instance model and enrich it with more details. Thus, to refine node types and fill their properties with additional runtime information, the plugins may specify multiple detectors that define graph elements they can refine. For example, the Tomcat plugin may be able to refine the normative *WebServer* (OASIS, 2020) node type to a concrete *Tomcat 9* node type if it can identify specific files or processes running an the corresponding VM. To tell the instance model normalizer, that the Tomcat plugin may be able refine a node template of type *WebServer* to a Tomcat web server, the plugin's detector contains a node template of type WebServer. Additionally, to also define that the plugin is able to find context paths of applications running on a Tomcat web server, it also defines a detector with two node templates: A Tomcat web server that hosts a node template of type *WebApplication*. Thus, to complete a whole application, the instance model completer applies all plugins that have matching detectors, i. e., their detectors are sub-graphs of the instance model, until there are no more plugins that can refine the instance model.

3.4 Management Feature Enricher

After a normalized and completed instance model of the application is generated, it is enriched with additional management functionalities using the adapted *Management Feature Enricher* in the fourth step. The enrichment of instance models with additional management functionality is hereby based on the MFEW method as described in Sect. 2.2. In general, we can distinguish two kinds of management functionalities: State-changing and state-preserving management functionality. While state-changing functionality alters the application's components or their configuration, e. g., renewing licenses or updating components, state-preserving functionality only interacts with the components, e. g., to retrieve their data. Therefore, the derived instance model of an application can always be enriched with state-preserving functionalities. In contrast, to execute state-changing functionalities on a running application, the underlying deployment technology must be taken into account, as state changes may be detected by the deployment technology which may restore the previous state. To avoid this, the instance model must be annotated with the deployment technology it was retrieved from. Using this information, state-changing functionality can be filtered by implementations that support the propagation of state changes to the respective deployment technology's API to avoid its interference. Therefore, feature node types that provide state-changing operations, must also be annotated with the deployment technologies they support. We extended the Management Feature Enricher to support filtering of functionalities based on the annotation of the used deployment technology. As a result, the enriched instance model contains only management functionalities that either support the deployment technology, or do not change the application's state.

3.5 Management Workflow Generator

In the fifth step, the reused *Management Workflow Generator* component automatically derives executable management workflows. After the instance model has been enriched with management functionalities, i. e., its node types have been replaced with generated enriched node types that provide the selected management functionalities, workflows are generated for each management functionality. For example, if a user selected the backup and test functionalities, two workflows are generated: One executing all tests of all components that have test operations, the second creates backups of the stateful components that provide a backup operation. The generation of these workflows is handled by plugins that implement logic to execute the respective operations of each component in the correct order.

Finally, the generated workflows can be executed on a compatible *Workflow Engine*. To invoke the workflows' execution, the application is accessible in a dashboard enabling users to manage the applications. Depending on the management operation, the workflow may require additional input. For example, to perform operations on a VM, a username and password or private key are required in order to connect to the VM and execute script-based operations.

4 PROTOTYPICAL VALIDATION

To prove the feasibility of the presented approach, a prototypical open-source implementation based on the OpenTOSCA ecosystem (Breitenbücher et al., 2016; OpenTOSCA, 2020) and the new Instance Model Retrieval Framework (Mathony, 2020) is provided. OpenTOSCA consists of three components: (i) Winery, a web-based modeling environment for TOSCA applications, (ii) the OpenTOSCA Runtime Container to deploy TOSCA applications, and (iii) Vinothek, a UI to manage running applications.

4.1 Ecosystem Overview & Extensions

The extensions to the OpenTOSCA Ecosystem and the new Instance Model Retrieval Framework are analogous to the extensions illustrated in Fig. 2. The new Instance Retrieval Framework implements both, the information retrieval and the transformation of the information to a normalized, TOSCA-based instance model. Thus, we designed a plugin-based component inside the Instance Model Retrieval Framework to enable its extensibility for new deployment technologies. To prove the concepts, we implemented plugins for AWS CloudFormation, OpenStack Heat, Puppet, and Kubernetes. These plugins implement technology-specific logic to retrieve information about running applications, i. e., their components, properties, and configurations, by sending requests to the respective deployment technology's API.

To derive a normalized instance model, the plugins identify the semantics of technology-specific components by mapping them to normalized node types defined in a node types repository. Thereby, provider-specific properties are mapped to the properties defined by the node types. As a result, a technology-agnostic model of the running application is generated which is then imported into the TOSCA modeling tool Winery. Winery not only provides modeling features but also implements the new Instance Model Completer as well as the Management Feature Enricher components. Thus, after importing the generated instance model, first the new Instance Model Completer component is executed to refine the node types and fill missing property values. Then the updated Management Feature Enricher can be invoked to identify available management functionalities that can be selected by the user depending on the desired functionalities for the current application. Moreover, to support the enrichment of state-changing management functionalities, the Management Feature Enricher component has been extended to filter available management operations based on the supported

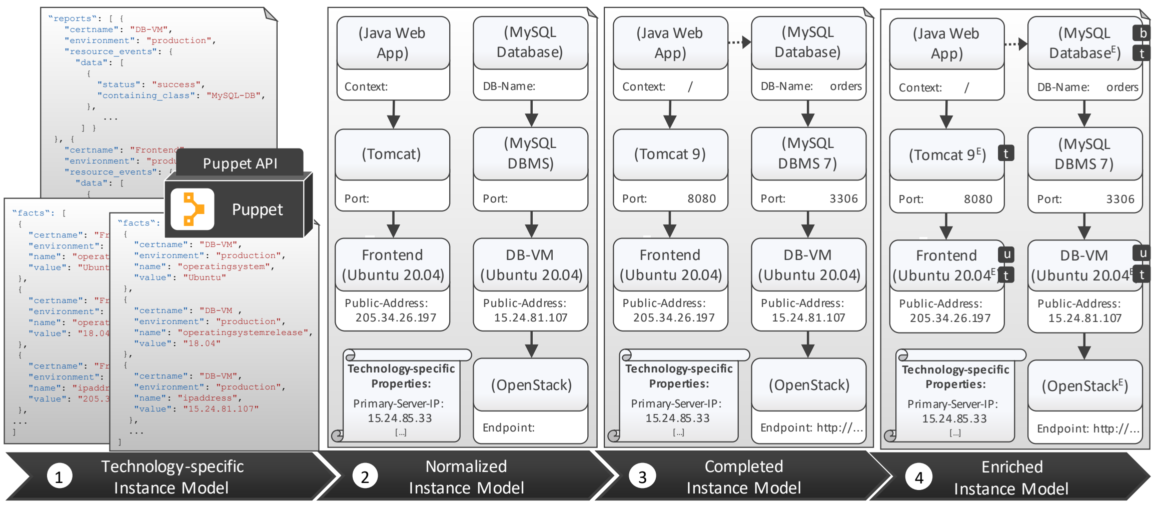
Puppet API
Puppet
(Java Web App)
(MySQL Database)
(Tomcat)
(MySQL DBMS)
Frontend (Ubuntu 20.04)
DB-VM (Ubuntu 20.04)
Public-Address: 205.34.26.197
Public-Address: 15.24.81.107
Technology-specific Properties:
Primary-Server-IP: 15.24.85.33
(OpenStack)
Endpoint:
(Tomcat 9)
(MySQL DBMS 7)
Port: 8080
Port: 3306
DB-Name: orders
Context: /
1 Technology-specific Instance Model
2 Normalized Instance Model
3 Completed Instance Model
4 Enriched Instance Model

Figure 3: Evolution of the application instance models.

deployment technologies. Thus, feature node types must define the set of deployment technologies their management operation implementations support. For example, if a feature node type supports updating VMs managed by Puppet or Chef, the feature node type must be annotated with Puppet and Chef to indicate that these technologies are supported.

Finally, the manageable instance model is passed to the OpenTOSCA runtime which generates executable BPEL workflows (OASIS, 2007) for each management operation. Then, the application is registered as a running instance which enables users to manage the imported application instance using the Vinothek. Thus, the Vinothek is a single dashboard that can be used to manage all applications registered as running applications in the OpenTOSCA runtime.

4.2 Case Study

To proof the feasibility of our approach, we deployed the application shown in Fig. 1 using Puppet and employed our prototype to retrieve its instance information. In general, Puppet is an agent-based deployment and configuration management technology (Puppet Labs, 2020). The agents are managed by a so-called *Primary Server* that specifies which software must be installed in which configuration on which *node*. A node is hereby a VM instance on which a *Puppet Agent* is installed performing installation and configuration tasks. The Puppet nodes are uniquely identified by *certnames*. For example, the certname of the VM hosting the Order App is called “Frontend”, while the VM hosting the database is called “DB-VM”.

To retrieve an application’s instance information, our Puppet instance retrieval plugin assumes that all nodes managed by one primary server are part of one application, a limitation of the current implementation we want to tackle in future work. Thus, the Instance Information Retriever component requires the IP-address and a password or private key of the VM running the primary server. Afterwards, by querying the list of all certnames registered at the Puppet primary server, all *facts*, e. g., IP-addresses and OS properties, about the nodes can be retrieved. Additionally, as the Puppet agents generate *reports* about the performed operations, e. g., which components have been installed and configured, they can be used to derive a normalized instance model of the application. Hence, facts and reports build the puppet-specific instance model of an application (cf. step 1 in Fig. 3) that can be accessed via Puppet’s API.

In the next step, the facts and reports are passed to the Instance Model Normalizer alongside plugin-specific information. In the Puppet case, these plugin-specific information are, among others, the IP-address and password of the primary server which are annotated to the instance model as shown in Fig. 3. Afterwards, all facts about Puppet nodes are investigated to identify the OS, including its release version the VM is running, as well as its properties, such as the IP address. In addition, the facts may also contain data about the hypervisor that is running the VM. However, as depicted in step 2 of Fig. 3, it is not always possible to correctly identify a hypervisor. In this case, the hypervisor information from the facts about the VM running on AWS are too generic to identify it as AWS, while OpenStack is directly named. Based on the reports generated by the Puppet nodes, the plugin is able to derive the node types describing the installed components. However, the reports do not always contain properties about which component version has been installed or how a component has been configured. Therefore, the normalized instance model in Fig. 3 does not contain values for all properties.

After the normalized instance model was derived, it must be completed to describe all required properties as well as to identify the components' versions: The normalized instance model is passed to the Instance Model Completer component that iteratively refines the instance model to complete it. Technology-specific plugins determine whether they are able to refine the instance model based on a subgraph detector (cf. Sect. 3.3). However, to execute a technology-specific plugin, it may require additional input, e. g., a private key to access a VM via SSH. These inputs can be given to the plugin by the user if it is not already available in the instance model. As a result, a completed instance model contains all properties and, in this case, even horizontal relations (cf. Fig. 3) can be identified and added to the model.

The completed instance model can be enriched with management functionalities using the adapted Instance Model Enricher component. Hereby, the available management functionalities are first divided into state-changing and state-preserving functionalities. In contrast to state-changing functionality, state-preserving functionality can always be enriched to an instance model. State-changing functionality can only be enriched, if an implementation is available that supports notifying the underlying deployment technology. Hence, a user may have different options to choose from while enriching different applications with management functionalities. In this case, the MySQL Database can be enriched with backup and testing functionality, the Tomcat web server can be tested, while the Ubuntu VMs can be tested and updated automatically (cf. step 4 of Fig. 3).

In the last step, the completed and enriched instance model is passed to the Management Workflow Generator that derives management workflows for each enriched management functionality. Here, a test workflow, a backup workflow, as well as a update workflow are generated that all can be executed on the running application and, thus, manage it.

5 DISCUSSION

As our main goal is to enrich running applications with holistic management functionality, we focused on the retrieval of instance information about the application in question from its underlying deployment technology. However, this poses three major challenges: (i) The access and expressiveness of an application's instance information provided by a deployment technology differs significantly. For example, while it is possible to retrieve software components and their configurations from Chef and Puppet, OpenStack Heat and AWS CloudFormation only allow to derive information about infrastructure components. Moreover, to enable a specific management of all components, the node types must be identified as exact as possible. In some cases, specific normalized types cannot be identified, and, thus, more generic ones, such as *Software Component* (OASIS, 2020), are used. In these cases, the available management operations that can be enriched to these components are limited as generically providing all management operations is not always possible. However, the models can be refined by our Instance Model Completer component, if corresponding plugins are available which are able to refine the derived instance model. (ii) The identification and normalization of node types is difficult and requires immense technical expertise. Therefore, deployment technology experts need to maintain the plugins to ensure that they are future-proof. (iii) Detecting horizontal relations is difficult or, depending on the deployment technology, even impossible to derive from the retrieved instance information. Deriving horizontal relations requires very detailed and component-specific knowledge about the respective components which is not maintained by all deployment technologies. However, by executing technology-specific plugins, we are able to identify horizontal relations with the limitation that a corresponding plugin must be available. Additionally, it is always possible to refine the instance model manually to enhance its expressiveness.

The enrichment and management of container-based deployment technologies, such as Kubernetes or Docker Compose, posses an additional challenge as containers are usually composed of multiple components that cannot be seen from outside. However, as containers can be accessed from the outside using an interactive shell, similar to SSH sessions on virtual machines, corresponding Instance Model Completer plugins can provide access to the containers. Thus, other plugins, such as the Tomcat plugin, can be used to identify components hidden inside the containers.

To enable the enrichment and execution of state-changing functionalities, the implementations must be done in a deployment technology-specific way. Otherwise, the deployment technology may interfere and restore the previous state. Therefore, for each state-changing management functionality, there must be a corresponding implementation for each supported deployment technology. Hereby, the implementations must consider the underlying deployment technologies when changing the applications' states. For example, to manage Kubernetes applications, Kubernetes' Operators concept can be used to realize an operation's implementation, while Puppet or Chef applications, for instance, can be implemented using the Puppet domain specific language (DSL) and the Chef DSL respectively. Such management operation implementations, however, can be arbitrarily complex as developers need to use the deployment technologies' APIs, which differ significantly in how to use them and their maturity level. However, state-preserving management operations can always be enriched as they only interact with the applications and do not change their state. Nevertheless, once implemented, all operations can be enriched to applications running instances of the corresponding node types. Thus, our approach facilitates the reuse of management functionalities in different running applications.

As state-changing functionalities require information about the underlying deployment technology, the instance model must include this information. However, there are multiple options to store these information. For example, the deployment technology specific information to access the APIs can be annotated to the application, as described in Sect. 4.2, or they can be added as separate model entities. For example, in the case of Puppet, the primary server could be represented as an additional node template of type Ubuntu 20.04. This node template would have relations to each identified node template it manages to indicate their relationship. However, we chose to use the annotation method to avoid mixing instance model information with management requirements.

Finally, the generated normalized model of the application only represents the retrieved instance data. In contrast to deployment models, the generated instance model may not be deployable, as artifacts, relationships, and even properties may be missing in the model. However, the gathered information is sufficient to enrich and execute management operations as the case study demonstrates.

6 RELATED WORK

To retrieve information about running services and whole applications, several approaches exist ranging from service discovery (Brogi et al., 2017) and network scanning (Holm et al., 2014) to identifying explicit software components as well as their properties and configurations (Binz et al., 2013; Farwick et al., 2011; Machiraju et al., 2000; Menzel et al., 2013). There are similar approaches available in the models@runtime community which mostly require an a priori model of the application (Bencomo et al., 2019). Thus, Bencomo et al. (2019) consider runtime model inference to be still an open research area.

To collect data for enterprise architecture (EA) management and automated maintenance, Farwick et al. (2011) introduce a semi-automated process to retrieve the components of an application and to enhance the actuality of EA models. Similarly, Binz et al. (2013) present a crawler to identify components of an application in an iterative process. Hereby, they employ a large set of plugins that are executed depending on the retrieved information in the previous iteration. The plugins are then able to identify new components or to refine the type of already discovered ones. Holm et al. (2014) use network scanners to identify infrastructure and software components. They use multiple authenticated and unauthenticated scanners to derive the components of an application and transform it into an ArchiMate model. ExplorViz (Fittkau et al., 2015) is a tool to monitor and visualize applications and their components. However, ExplorViz focuses on the visualization applications and their interactions by employing dynamic analysis techniques. In contrast to our approach, they all focus on the retrieval of component instances using custom software programs, such as crawlers, network scanners, and dynamic analysis techniques. We depend on instance information provided by the APIs of the used deployment technologies as our goal is to enrich applications with management functionalities that may change their state. This, however, requires the information about the deployment technology.

Machiraju et al. (2000) introduced a generic approach to discover application configurations. To generate models of running applications, they use predefined *application templates* specifying, e. g., the discovery technique that should be used, and the required attributes that should be identified for the running instance. Thus, an *application template model* must be created beforehand, i. e., the components must already be known and only their configurations can be retrieved automatically. Thus, the approach could also be integrated into our approach to refine the retrieved instance model and complete it. However, we chose to base our Instance Model Completer on the concepts introduced by Binz et al. (2013) as the plugin-based architecture facilitates its extension.

To enable the management of cloud applications, several works exist, such as basic application provisioning (Mietzner et al., 2009; Breitenbücher et al., 2014; Eilam et al., 2011; Herden et al., 2010), changing application configurations (Brown and Keller, 2006), and state-changing management functionalities such as the termination of applications while ensuring that their internal data is saved and the whole application can be restored, including its previous state (Harzenetter et al., 2019a), or the *Context-Aware Management Method* which enables the migration of applications from one cloud providers to another as presented by Breitenbücher et al. (2013).

Other work focuses on generating management workflows based on a desired state model that declaratively describes the state in which an application has to be transferred: Breitenbücher (Breitenbücher et al., 2013; Breitenbücher, 2016) introduced *Management Planlets*, which are workflow fragments that can be orchestrated automatically by a *Plan Generator* to perform management functionalities for applications. A Planlet consists (i) of a detector fragment that specifies which *Management Annotations* a Planlet realizes on a certain graph of components and relations and (ii) provides a workflow model that implements this functionality. Hereby, a Management Annotation specifies a management functionality to be realized, e.g., that a backup has to be done for a component. These Management Annotations are used in the desired state model to describe the desired functionalities, which is then executed by orchestrating planlets into a workflow model that can be executed automatically. Eilam et al. (2011) introduced automation signatures that define “patterns” to specify operations that can be performed on a given state model of an application. In contrast to these approaches, we enrich instance models with operations that are then executed by a generated workflow. Moreover, the mentioned approaches do neither cover automated retrieval of instance models from underlying deployment technologies nor their normalization as proposed by our approach.

7 CONCLUSION AND FUTURE WORK

In this paper, we showed how running applications can be enriched with additional management operations and how they can be executed by automatically generating management workflows. To achieve this, our approach enables the generation of normalized instance models of running applications in the form of TOSCA topology templates. The retrieval of instance information about an application from its deployment technology poses multiple challenges and may not produce a complete instance model. However, we showed that by reusing and integrating existing approaches into the proposed approach, a complete instance model of an application can be generated automatically. Based on the derived instance model, we are able to perform holistic management functionalities, such as testing, updating, and backing up the application’s components, by generating corresponding executable management workflows.

In future work, we plan to implement more plugins to (i) support more deployment technologies, as well as to (ii) support the refinement of more component-specific information inside the Instance Model Completer. Additionally, we want to support deriving instance models of applications that depend on multiple deployment technologies.

ACKNOWLEDGEMENT

This work was partially funded by the EU project *RADON* (825040) and the German Research Foundation (DFG) projects SustainLife (379522012) and DiStOPT (252975529). The authors thank Tobias Mathony for his help in implementing the prototype.

REFERENCES

Bencomo, N., Götz, S., and Song, H. (2019). Models@run.time: a guided tour of the state of the art and research challenges. *Software and Systems Modeling*, 18(5):3049–3082.

Bergmayr, A., Breitenbücher, U., Ferry, N., Rossini, A., Solberg, A., Wimmer, M., and Kappel, G. (2018). A Systematic Review of Cloud Modeling Languages. *ACM Computing Surveys (CSUR)*, 51(1):1–38.

Binz, T., Breitenbücher, U., Kopp, O., and Leymann, F. (2013). Automated Discovery and Maintenance of Enterprise Topology Graphs. In: *SOCA 2013*, pages 126–134. IEEE.

Binz, T., Fehling, C., Leymann, F., Nowak, A., and Schumm, D. (2012). Formalizing the Cloud through Enterprise Topology Graphs. In: *CLOUD 2012*, pages 742–749. IEEE.

Breitenbücher, U. (2016). *Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements*. Dissertation, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology.

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: *IC2E 2014*, pages 87–96. IEEE.

Breitenbücher, U., Binz, T., Kopp, O., and Leymann, F. (2013). Pattern-based Runtime Management of Composite Cloud Applications. In: *CLOSER 2013*, pages 475–482. SciTePress.

Breitenbücher, U. et al. (2016). The OpenTOSCA Ecosystem - Concepts & Tools. In: *European Space project on Smart Systems, Big Data, Future Internet - Towards Serving the Grand Societal Challenges - Volume 1: EPS Rome 2016,*, pages 112–130. SciTePress.

Brogi, A., Cifariello, P., and Soldani, J. (2017). DrACO: Discovering available cloud offerings. *Computer Science - Research and Development*, 32(3-4):269–279.

Brown, A. and Keller, A. (2006). A Best Practice Approach for Automating IT Management Processes. In: *NOMS 2006*, pages 33–44. IEEE.

Eilam, T., Elder, M., Konstantinou, A. V., and Snible, E. (2011). Pattern-based Composite Application Deployment. In: *IM 2011*, pages 217–224. IEEE.

Endres, C., Breitenbücher, U., Falkenthal, M., Kopp, O., Leymann, F., and Wettinger, J. (2017). Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications. In: *PATTERNS 2017*, pages 22–27. Xpert Publishing Services.

Farwick, M., Agreiter, B., Breu, R., Ryll, S., Voges, K., and Hanschke, I. (2011). Automation Processes for Enterprise Architecture Management. In: *EDOC 2011*, pages 340–349.

Fittkau, F., Roth, S., and Hasselbring, W. (2015). ExplorViz: Visual runtime behavior analysis of enterprise application landscapes. In: *ECIS 2015*. AIS.

Harzenetter, L., Breitenbücher, U., Képes, K., and Leymann, F. (2019a). Freezing and Defrosting Cloud Applications: Automated Saving and Restoring of Running Applications. *SICS*, 35:101–114.

Harzenetter, L., Breitenbücher, U., Leymann, F., Saatkamp, K., and Weder, B. (2019b). Automated Generation of Management Workflows for Applications Based on Deployment Models. In: *EDOC 2019*, pages 216–225. IEEE.

Herden, S., Zwanziger, A., and Robinson, P. (2010). Declarative Application Deployment and Change Management. In: *CNSM 2010*, pages 126–133. IEEE.

Holm, H., Buschle, M., Lagerström, R., and Ekstedt, M. (2014). Automatic data collection for enterprise architecture models. *Software and Systems Modeling*, 13(2):825–841.

Machiraju, V., Dekhil, M., Wurster, K., Garg, P. K., Griss, M. L., and Holland, J. (2000). Towards Generic Application Auto-Discovery. In: *NOMS 2000*, pages 75–87. IEEE.

Mathony, T. (2020). Instance Model Retrieval Framework. https://github.com/ust-edmm/edmm/tree/master/edmm-instance, 2020-10-30.

Menzel, M., Klems, M., Le, H. A., and Tai, S. (2013). A configuration crawler for virtual appliances in compute clouds. In: *2013 IEEE International Conference on Cloud Engineering (IC2E)*, pages 201–209. IEEE.

Mietzner, R., Unger, T., and Leymann, F. (2009). Cafe: A Generic Configurable Customizable Composite Cloud Application Framework. In: *CoopIS 2009*, pages 357–364. Springer.

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

OASIS (2013). *Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0*. OASIS.

OASIS (2020). *TOSCA Simple Profile in YAML Version 1.3*. OASIS.

OpenTOSCA (2020). OpenTOSCA Ecosystem. https://github.com/OpenTOCSA, 2020-10-30.

Oppenheimer, D. (2003). The importance of understanding distributed system configuration. In: *CHI 2003*. ACM.

Puppet Labs (2020). Puppet Official Site. https://puppet.com/solutions/cloud-hybrid-automation/, 2020-10-30.

Wurster, M., Breitenbücher, U., Falkenthal, M., Krieger, C., Leymann, F., Saatkamp, K., and Soldani, J. (2019). The Essential Deployment Metamodel: A Systematic Review of Deployment Automation Technologies. *SICS*, 35:63–75.