Institute of Architecture of Application Systems

Modeling and Automated Execution of Application Deployment Tests

Michael Wurster*, Uwe Breitenbücher*, Oliver Kopp†, and Frank Leymann*

*Institute of Architecture of Application Systems,
†Institute for Parallel and Distributed Systems,
University of Stuttgart, Stuttgart, Germany
{wurster, breitenbuecher, kopp, leymann}@informatik.uni-stuttgart.de

BibTeX :

```
@inproceedings{Wurster2018_DeploymentTesting,
  author    = {Michael Wurster and Uwe Breitenb{\"u}cher and Oliver Kopp and
               Frank Leymann},
  title     = {{Modeling and Automated Execution of Application Deployment Tests}},
  booktitle = {Proceedings of the IEEE 22nd International Enterprise Distributed
               Object Computing Conference (EDOC)},
  year      = {2018},
  pages     = {171--180},
  doi       = {10.1109/EDOC.2018.00030},
  publisher = {IEEE Computer Society}
}
```


Universität Stuttgart
Germany

Modeling and Automated Execution of Application Deployment Tests

Michael Wurster*, Uwe Breitenbücher*, Oliver Kopp†, and Frank Leymann*

*Institute of Architecture of Application Systems, †Institute for Parallel and Distributed Systems
University of Stuttgart, Stuttgart, Germany
{wurster, breitenbuecher, kopp, leymann}@informatik.uni-stuttgart.de

***Abstract*—In recent years, many deployment systems have been developed that process deployment models to automatically provision applications. The main objective of these systems is to shorten delivery times and to ensure a proper execution of the deployment process. However, these systems mainly focus on the correct technical execution of the deployment, but do not check whether the deployed application is working properly. Especially in DevOps scenarios where applications are modified frequently, this can quickly lead to broken deployments, for example, if a wrong component version was specified in the deployment model that has not been adapted to a new database schema. Ironically, even hardly noticeable errors in deployment models quickly result in technically successful deployments, which do not work at all. In this paper, we tackle these issues. We present a modeling concept that enables developers to define deployment tests directly along with the deployment model. These tests are then automatically run by a runtime after deployment to verify that the application is working properly. To validate the technical feasibility of the approach, we applied the concept to TOSCA and extended an existing open source TOSCA runtime.**

***Keywords*-Testing, Declarative Application Deployment, Test Automation, Model-based Testing, TOSCA**

I. Introduction

Cloud Computing has emerged as a commonly accepted paradigm to develop, deploy, and operate applications. Moreover, due to market pressure and effects of competition, more and more companies adopt modern agile software development methods in conjunction with Continuous Delivery in order to shorten their development cycles and deliver their Cloud applications faster and more frequently [1]. An essential prerequisite for the frequent and fast delivery of applications is automating their deployment—especially as manual deployment is error-prone and quickly leads to severe application failures [2]. This also results from the high complexity of modern application which often requires combining several deployment technologies [3]. For example, often APIs of cloud providers have to be used to provision virtual machines while deployment automation systems such as Chef [4] or Ansible [5] install software. Most of these deployment systems employ *deployment models* that describe the desired deployment, which can be executed automatically [6].

However, deployment systems typically focus on the correct technical execution of the deployment, i. e., they focus on installing, configuring, and orchestrating the application's components as described by the deployment model, but do not check if the final deployment works as intended by the developers. For example, if a distributed application gets deployed across multiple clouds, components deployed in different data centers may need to communicate with each other. However, cloud providers typically apply different default security settings, for instance, some open common ports of created virtual machines while others close all ports by default. Thus, if this is not considered correctly in the deployment model, the technical deployment may be successful while the communication between the components does not work. Ironically, there are many reasons for a deployment to complete without technical errors while the application's functionality is broken—often this is only detected by a monitoring system or, even worse, users.

In this paper, we tackle these issues. We present a modeling concept for specifying *Application Deployment Tests* directly in deployment models and show how these tests are executed automatically after the deployment finishes. Thus, while the technical success of a deployment is already handled by deployment systems, our approach enables developers to specify deployment tests that also check the deployment success from a business perspective. We show that the approach is completely provider-agnostic and independent of individual technologies. Thus, we provide an extended meta-model for declarative application modeling in order to describe deployment tests as annotations so that they can be attached to arbitrary application topologies. As TOSCA [7] enables integrating various deployment systems [8], we show how our approach can be used to specify deployment tests for TOSCA topologies. Therefore, we validate the practical feasibility of the presented approach by an prototype based on the TOSCA standard and the OpenTOSCA ecosystem.

The remainder of this paper is structured as follows: Section II motivates the resulting work and highlights the challenges. Section III describes our approach in detail, while Section IV and Section V validates our approach. Finally, Section VI discusses related work whereas Section VII concludes and discusses future work.

II. Motivation and Fundamentals

In this section, we motivate the need for automated application deployment testing and introduced fundamentals terms and concept required for understanding this paper.

A. Deployment Models & Deployment Testing

For automating cloud application deployments several technologies and standards are available. To automate a deployment the desired result is typically described in the form of a *deployment model*, which are developed, tested, and maintained by development teams. In general, there are two classes of deployment models: (i) imperative and (ii) declarative models [9], [10]. Imperative models, such as Shell scripts or Ansible Playbooks [5], describe the deployment steps in a procedural manner and are executed exactly as described. In contrast, declarative deployment models, such as AWS CloudFormation [11], describe the desired result and a runtime drives the necessary deployment logic.

However, today's deployment systems typically verify only whether the technical execution of the deployment has been successful, but not if the deployed application works correctly. Thus, deployment errors that are not of a technical nature often have to be detected by monitoring systems, additional automated smoke or functional test systems, or when customers face problems by using the application. As a result, to avoid that errors are visible to users, *application deployment testing* is just as important as unit and system testing during development, but interestingly not well-supported by existing deployment technologies.

In the following, we discuss three main issues we identified that possibly result in technically successful deployments which do not function correctly from a business perspective. Afterwards, we present our approach based on declarative deployment models in Section III to tackle these issues.

B. Issue 1: Differences in Environments

Deployment models are often used for deploying a certain kind of application in multiple environments which slightly differ from each other, for example, different clouds. Depending on the evolution of the employed technologies and the infrastructure on which the application needs to be deployed, failures may lead to an unsuccessful deployment. For example, a deployment model that creates a virtual machine using the API of OpenStack and that provisions a web server may be successful in an environment in which strict network configurations are not enforced, but not in another environment where strict network settings are enforced. This results in the problem that another virtual machine or service cannot communicate with the web server. Therefore, a deployment model that expects a certain combination of resources, settings, and configurations may run successfully in one environment, but not in another if not all mandatory requirements are specified by the model.

C. Issue 2: Deployment Model Aging

Deployment models are often built to deploy a certain kind of application repetitively. For example, a company typically specifies deployment models for all services they offer to their customers, e. g., the deployment of a LAMP-based application, which can then be instantiated automatically whenever a customer requests it. However, from time to time new versions of software and hardware components need to be used due to security fixes or feature roll outs. A prominent example is the release and the migration to a new hypervisor version, for example, due to a security issue. However, exchanging component versions often result in successfully executed deployment models but on runtime the component may not work due to the fact that, for example, the API has not changed syntactically, but semantically [12]. For example, in a previous version all virtual machines—provisioned within one security group—have been able to communicate with each other by default, whereas in the new version it is required to enable this configuration setting explicitly. Then, a deployment model which worked correctly in a previous version may be executed successfully as well on the new version, but the application itself does not work correctly anymore. Unfortunately, such problems are typically hard to detect on the model layer as only the execution of the deployment model shows what works and what fails. On top of that, cloud deployments often consist of multiple, even hundreds, deployment models. Therefore, maintaining them on every single change of a cloud platform version is not feasible. Thus, we require an deployment testing approach, which detects outdated deployment models immediately instead of waiting until problems get reported by users.

D. Issue 3: Complexity of Cloud Applications

Cloud applications typically consist of various different types of components which have complex relationships to each other. Nowadays, cloud applications often apply a microservice architecture, where components are loosely-coupled and independently deployable, resulting in a more complex, large-scale, and distributed application system [13]. This makes deployment testing very hard to realize for operation teams that have not been directly involved in the development. In addition, modern cloud applications can be distributed to multiple cloud providers each supporting different deployment technologies. Thus, deployment testing often requires understanding each of these technologies to detect possible problems. As a result, a lot of expertise is required to (i) implement, (ii) execute, and (iii) orchestrate automated testing routines. Even if developers provide automated test routines for their components, these test routines must be integrated into one overall deployment test that can be executed automatically after the deployment. To achieve this, scripts, workflows, or programs must be implemented that orchestrate the individual tests, which is a highly non-trivial technical integration challenge.

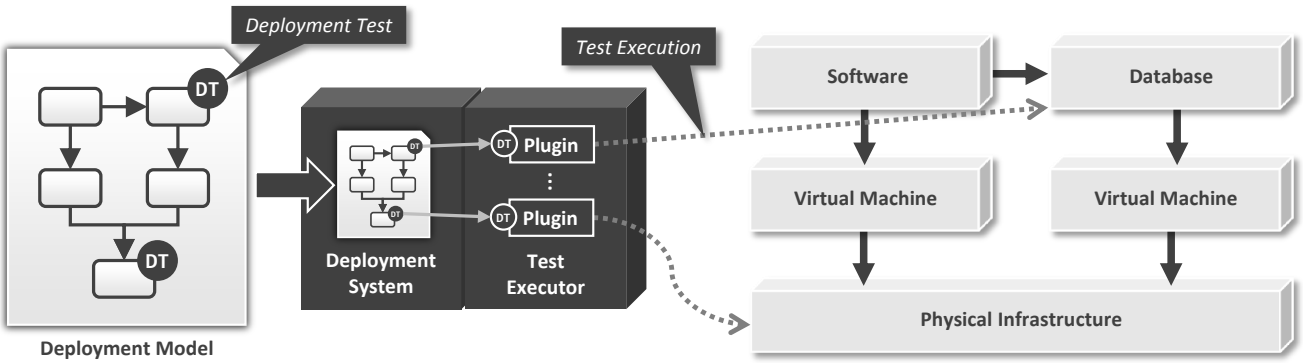

Figure 1. Overview of the Automated Deployment Test concept: A deployment model annotated with Application Deployment Tests is automatically tested once it has been deployed by a deployment system.

III. Automated Deployment Test Concept

To deal with the aforementioned challenges, we introduce an automated deployment test concept. We first present an overview of our approach in Section III-A. Afterwards, we present in Section III-B a general meta-model for application topologies that uses an *Annotation* class to express deployment tests. To conclude this section, we discuss possible limitations of our approach in Section III-C.

A. Conceptual Overview

Figure 1 shows an overview of the proposed concept. The idea is that a developer annotates components in the deployment model that should be tested with *Application Deployment Tests*. Application Deployment Tests are declarative by nature. This means that they do not specify a control flow of steps to be executed to test a component but only *what* shall be tested and the expected *result*. For example, a developer could annotate a Deployment Test to check if a certain port of a component is exposed publicly. Another example is a test case to check if a database schema has been correctly installed or if a RESTful HTTP API returns an expected result. Similarly, arbitrary kinds of deployment tests can be annotated to arbitrary types of components.

Once the used *Deployment System* has successfully executed the deployment, a *Test Executor* component is triggered. The Test Executor follows a plugin architecture where the logic is implemented how to execute certain types of tests onto the deployed infrastructure. Based on a set of available plugins and the supplied deployment model, the Test Executor runs the specified Application Deployment Tests and reports the results accordingly. In case of failed tests, a deployment system is able to automatically revert the deployment completely or to roll-back to a previous version.

B. Meta-Model

To present our concept for declarative deployment models, we introduce a minimalistic meta-model. We describe the formal elements of application topologies abstractly in order to be independent of a concrete definition language and to enable the adaptation of this approach to different languages.

Figure 2 gives an overview on the meta-model, which is presented briefly in the following. A Topology is a directed, weighted, and possibly disconnected graph and describes the structure of an application. The structure consists of Components and Relations, whereas Relations are used to represent the relationship between two components. Relations and Components are either of type Component Type or Relation Type. These types describe the semantics for a Component or Relation having this type. For the management of Components, various management technologies can be used, such as executable scripts or declarative tools for configuration management. To describe that a particular Component can be managed with such technologies, our meta-model defines the class Management Operation and can be defined for Component Types. Moreover, each element class of the meta-model can define multiple Properties, whereby these Properties can be in the form of simple key-value pairs (KVP) or rather complex structures, such as structures defined with XML or JSON syntax. Besides that, our meta-model allows to define and attach Annotations to Components. In our concept, we use the Annotation class to express Deployment Tests which are automatically executed once the deployment has been done. Therefore, we define a Deployment Test class that acts the base type for possible sub classes. Based on this, arbitrary kinds of specialized Deployment Tests can be derived and used to be executed.

In the following, we categorize and describe two kinds of predefined classes: (i) *Domain-Specific Test* and (ii) *Management Operation Test*.

1) Domain-Specific Test: A Domain-Specific Test is a test for a certain functionality in a specific field. For example, one can create a test for a Component's RESTful HTTP API. A concrete test would check whether a `GET` on a resource returns an expected result. Such tests require a specific plugin in the deployment system for execution and may execute arbitrary test logic implemented by the plugin. Here, the functionality and the execution of the test is completely

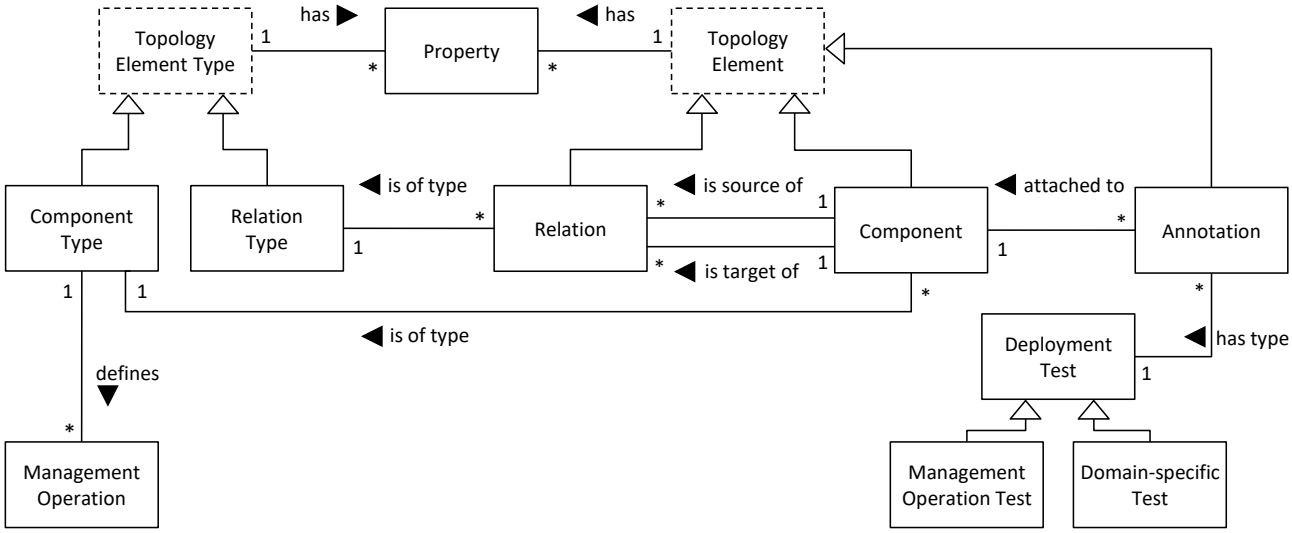

Figure 2. Meta-Model: Automated Deployment Tests for Application Topologies

provided by the plugin in the deployment system while the Deployment Test only specifies the path, the HTTP method to be executed, as well as the expected result. This configuration is specified in the form of Properties.

2) Management Operation Test: Besides Domain-Specific Tests that require a matching plugin in the deployment system, the second class we predefine is Management Operation Test. A Management Operation Test executes a specified Management Operation of the associated Component with specified input parameters and compares the result with specified output parameters. Thus, this class of Deployment Tests enables utilizing the management functionality already provided by the model. For example, a Component having the possibility to run executable scripts for management, a certain test can specify the operation to be executed as well as the expected result, which provides a powerful and generic test interface. Hence, only one generic plugin is require being able to invoke arbitrary Management Operations.

C. Limitations of the Approach

To check if a certain application port is publicly available or whether a HTTP endpoint returns a certain payload, specific plugins have to be developed and registered in the deployment system. The presented approach bases on the assumption, that a deployment system provides a set of plugins that understand how to interpret and execute attached deployment test annotations; and is therefore declarative by default. Thus, a deployment system has to provide a basic set of practically relevant plugins as well as has to be designed to be extensible with custom plugins from arbitrary sources. It is left for future work to incorporate an imperative approach in order to mitigate this limitation, for example, to specify an execution flow for test cases or to define that the output of one test case serves as the input for another one.

Integrating test execution into a deployment system adds additional overhead in time of execution. However, even very small differences in environments may hinder the successful deployment of applications and services (cf. Section II-B). Furthermore, deployment models should work with different kinds of cloud platform versions and therefore have to be forward and backward compatible (cf. Section II-C). Therefore, we argue that it is worth having an integrated system that detects errors in deployment models quickly and immediately after deployments.

IV. Validation Based on TOSCA

In this section, we explain how this approach can be applied to TOSCA, a standard to describe cloud application deployments [7]. However, the conceptual idea could be similarly applied to other deployment technologies that are based on declarative deployment modeling languages which are similar to TOSCA [6]. We chose TOSCA since it can not only be used to model and deploy cloud native applications [14], but also be used to model arbitrary kinds of IoT deployments including different kinds of IoT messaging middlewares [15], [16]. On top of that, it was shown that TOSCA can be used for DevOps automation [17], [18].

In the following, we show a TOSCA-based modeling approach for specifying Application Deployment Tests in Section IV-A while we explain thoroughly how to map our meta-model to TOSCA. Usage examples are presented in Section IV-B and we present the serialization in TOSCA in Section IV-C. Finally, we explain in Section IV-D how the modeled Deployment Tests can be executed automatically.

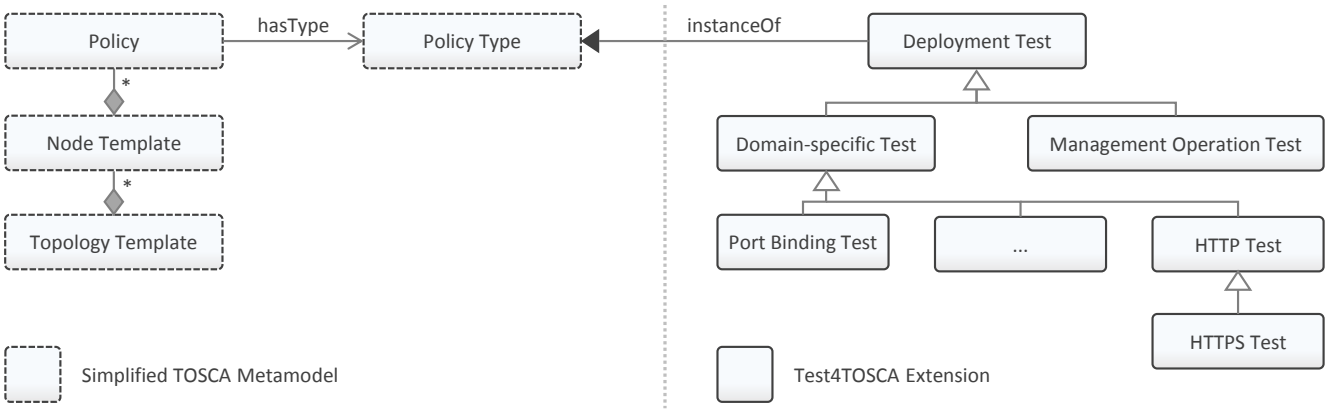

Figure 3. Simplified TOSCA meta-model (left) and TEST4TOSCA extension (right)

A. Application Deployment Test Modeling

In this section, we briefly introduce TOSCA and show how we can map our meta-model (cf. Section III-B) to TOSCA's meta-model. We present the mapping along with the description of the specific TOSCA element, whereby we use the following notation: *Meta-Model Element* ⇒ *TOSCA Element*. We simplify and skip all TOSCA details that are not important in our context.

Topology ⇒ *Topology Template*: TOSCA specifies a meta-model that enables describing the deployment of an application in a declarative manner by specifying the structure of the application to be deployed, i. e., their components as well as their relationships. This structure can be modeled as a directed graph consisting of nodes and edges and is defined in a Topology Template.

Component ⇒ *Node Template*: The nodes are called Node Templates and represent the components of an application, for example, a web server or a virtual machine.

Relation ⇒ *Relationship Template*: The edges are called *Relationship Templates* and represent the relations between nodes, e.g., that a certain software component is "hosted on" another one.

Component Type ⇒ *Node Type* and *Relation Type* ⇒ *Relationship Type*: For reusability purposes, the semantics of Node Templates and Relationship Templates are specified by defining Node Types and Relationship Types. For example, a Node Template can be of Node Type "WebServer" whereas a Relationship Template can be of Relationship Type "hostedOn".

Property ⇒ *Properties Definition*: Node Types as well as Relationship Types define Properties, enabling the configuration of instances of these types.

Management Operation ⇒ *Interface, Operation*: Further, Node Types define Management Interfaces and Operations for managing the instances of these types. For example, a Node Type representing an OpenStack computing environment may define both "startVM" and "stopVM" operations to start and stop a virtual machine.

Annotation ⇒ *Policy (Template)*: To express non-functional requirements on deployment and runtime, TOSCA employs Policies. A TOSCA Policy can be attached to a Node Template and is used to specify a non-functional requirement for the associated component. Fig. 3 shows the simplified TOSCA meta-model including these relations on the left side. For example, security aspects such as that a virtual machine must be created in a data center in a certain country can be specified using such a Policy. For configuration, Policies also provide properties that can be specified by the modeler, e. g., the aforementioned region. Policies are then enforced during the deployment of the application, which depends on the kind of Policy [19], [20].

Deployment Test ⇒ *Policy Type*: To provide the semantics and schema of a Policy, a Policy Type can be created and referenced similarly to Node Templates that reference a certain Node Type. TOSCA also enables to create sub types of Policy Types to refine their semantics.

For specifying Application Deployment Tests accordingly to the previous section, we use this Policy modeling construct. A specific test is modeled as a TOSCA Policy attached to a Node Template. To indicate that the Policy specifies a Deployment Test, a special Deployment Test Policy Type is defined. We call this TOSCA extension TEST4TOSCA which is shown on the right of Fig. 3. However, the Deployment Test Policy Type itself is specified as *abstract*, which means that it cannot be used directly as type of a Policy. Thus, each non-abstract sub type of this abstract Policy Type can be used to specify a Deployment Test to be executed. Based on this Deployment Test Policy Type, arbitrary kinds of specialized Deployment Tests can be derived as sub types, for example, Deployment Tests that check if a certain HTTP request is responded with the correct status as indicated in Fig. 3 on the right bottom. Furthermore, a Management Operation Test is defined being able to execute a specified Management Operation of the associated Node Template with specified input parameters and to compare the result with specified output parameters. This class of Deployment

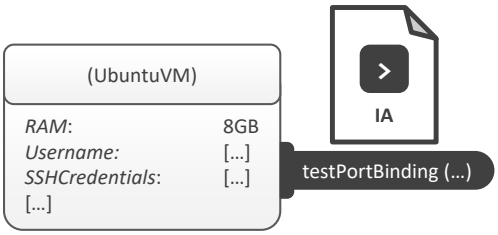

Figure 4. Node Type exposing a test interface.

Tests enables utilizing the management functionality already provided by TOSCA for testing. Moreover, also application-specific tests can be implemented individually and stored along with the application topology: If a certain complex test needs to be executed after deployment, this test can be implemented in a custom management operation and an appropriate Deployment Test Policy refers to this operation, as show in Fig. 4. All elements, artifacts, and files can then be bundled in a Cloud Service Archive (CSAR) which enables shipping self-contained archives that also contain their own test logic.

B. Deployment Test Examples

Figure 5 shows a simplified application topology that is used as use case scenario. For our scenario, we choose a Java web application representing an arbitrary web shop application. The shop application is packaged as web application archive (WAR)—indicated as a Deployment Artifact on the *Shop Application* Node Template—and is hosted on Apache Tomcat. An instance of Apache Tomcat is installed and configured on an Ubuntu virtual machine. For the sake of brevity, we omitted the infrastructure layer. Anyhow, the Ubuntu virtual machine can be hosted on a elastic infrastructure layer, such as OpenStack, on a cloud provider, such as Amazon Web Services, or on a bare metal server. We annotated this example with four test cases. First of all, we want to test the Shop Application if the `connectTo` operation of the Node Type *JavaWebApplication* works based on the given input parameters and returns the expected result. Furthermore, we want to test if the path `/shop` of the Shop Application can be successfully reached using a `HTTP GET`. For the Apache Tomcat instance, we specified a test to check if port `8080` is publicly available. On the operating system level, we specified a test to check if port `22` is bound. The differences between those two tests is that the *PortBindingTest* checks if a port is bound but not necessarily available from the public, e.g., through a firewall. Whereas, the *TcpPingTest* checks if the port is reachable from the public, also through firewalls. However, in case of the *PortBindingTest*, the Test Executor can utilize a test operation exposed by the Ubuntu Node Type, as indicated by Fig. 4.

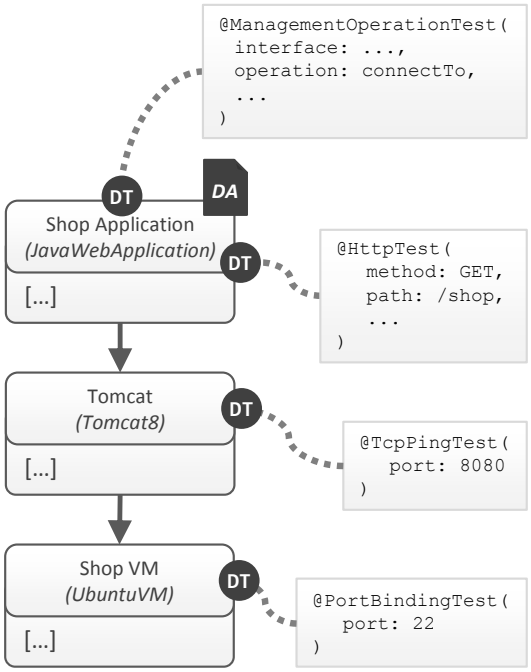

Figure 5. Excerpt of a simplified Topology Template decorated with Deployment Test Annotations.

C. Serialization of Application Deployment Tests in TOSCA

Based on the introduced example from the previous subsection, we show how deployment tests can be serialized using TOSCA. We base this work on TOSCA's XML version [7], but it could also be used with the later version *TOSCA Simple Profile*, which uses YAML [21].

In Listing 1 we show an excerpt of the resulting XML[1]. The listing shows the serialization result of the four test cases from Fig. 5. In each Policy, properties can be specified as a list of key-value pairs (KVP). The possible set of properties are defined by the respective Policy Types. As also explained in previous sections, the *ManagementOperationTest* can be used to invoke arbitrary Management Operations of respective Node Types in order to check the result against an expectation. Therefore, after specifying an interface and operation name, a developer can name input parameters that should be used in the generated test case by using the property `TestInputParameters`. Furthermore, to specify the expectation, the property `ExpectedOutputParameters` can be used. In TEST4TOSCA we define that such properties are either primitive types, such as integers or strings, or complex ones containing JSON syntax. A respective TOSCA-runtime, enabled to run the specified deployment tests, must be capable of interpreting the properties accordingly. Moreover, for the *HttpTest* certain HTTP related properties,

[1]For reasons of clarity, we do not present the TOSCA indirection of Policy Templates in this paper.

```
<tosca:Policy
  type="tests:ManagementOperationTest"
  xmlns:tests="annotations/tests">
  <Properties>
    <InterfaceName>
      interfaces/database
    </InterfaceName>
    <OperationName>
      connectTo
    </OperationName>
    <TestInputParameters>
      {
        "DBName": "shop",
        "DBUser": "app",
        ...
      }
    </TestInputParameters>
    <ExpectedOutputParameters>
      { "Result": "SUCCESS" }
    </ExpectedOutputParameters>
  </Properties>
</tosca:Policy>

<tosca:Policy
  type="tests:HttpTest"
  xmlns:tests="annotations/tests">
  <Properties>
    <Method>GET</Method>
    <Path>/shop</Path>
    <ExpectedStatus>200</ExpectedStatus>
    <!-- ... -->
  </Properties>
</tosca:Policy>

<tosca:Policy
  type="tests:TcpPingTest"
  xmlns:tests="annotations/tests">
  <Properties>
    <Port/>
    <PortPropertyName>
      Port
    </PortPropertyName>
  </Properties>
</tosca:Policy>

<tosca:Policy
  type="tests:PortBindingTest"
  xmlns:tests="annotations/tests">
  <Properties>
    <Port>22</Port>
    <PortPropertyName/>
  </Properties>
</tosca:Policy>
```

Listing 1. Application Deployment Tests modeled as TOSCA Policies.

such as the method and the expected status code, can be specified. In fact, there are more properties available but for the sake of simplification we omitted them in the example. For the *TcpPingTest* as well as the *PortBindingTest*, a specific port number can be specified. In turn, by using the `PortPropertyName` property, the name of a TOSCA property can be specified. The generated test case will then use the respective value during execution.

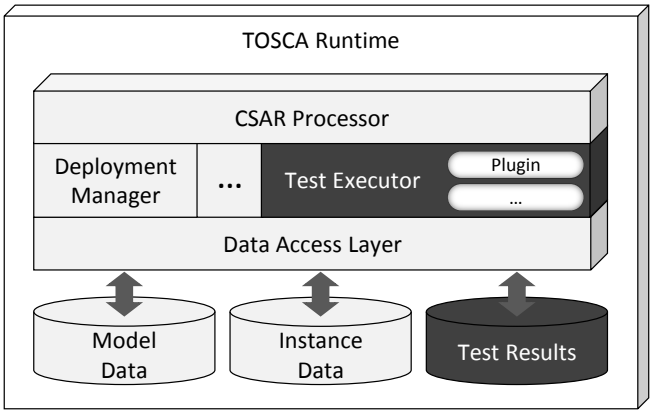

Figure 6. Architecture of an extended TOSCA runtime that supports executing TEST4TOSCA Application Deployment Tests.

D. Application Deployment Test Execution

In this section, we present how a TOSCA runtime needs to be extended to automatically execute modeled tests.

Figure 6 shows a simplified conceptual deployment system architecture for TOSCA. To execute the specified deployment tests after an application deployment job finished, we introduce a *Test Executor* component as part of a TOSCA runtime, similarly as described in Section III. The Test Executor component is triggered by the *Deployment Manager* directly after a deployment finished. This Deployment Manager is responsible for retrieving the TOSCA models, deriving the deployment tasks to be executed, and to execute these tasks—for example, the approach presented by Breitenbücher et al. [10] describes how such a component can be implemented. Once invoked, the Test Executer determines for each modeled Application Deployment Test Policy a *plugin* that is registered in a local plugin registry. Each plugin specifies which types of Deployment Test it can execute. When a matching plugin has been found, a method of the plugin gets invoked which determines if the plugin can execute the test for the associated Node Template. If the plugin can execute the test, it gets invoked with the corresponding Application Deployment Test to actually perform the specified test on the running component represented by the Node Template. For this execution, instance data of the deployed application is maybe required, e. g., the IP address of a virtual machine to be tested. To access this information, a *Data Access Layer* provides access to this instance data. The result of the overall test execution and each test case is stored in a *Test Results* database and, for example, exposed to a user interface. In terms of a successful deployment, the deployment is considered to be successful only if all modeled Deployment Tests were executed without failures. In the next section, we describe a prototypical implementation of this abstract deployment system architecture to prove the practical feasibility of the presented modeling and execution approach.

V. Prototype and Application

To show the feasibility of the concept, we implemented our approach by extending the existing TOSCA-based ecosystem OpenTOSCA. Thereby, we extended the modeling tool Winery[2] and the TOSCA runtime OpenTOSCA Container[3]. We set up a public GitHub repository (https://github.com/miwurster/opentosca-deployment-tests) where we published detailed screenshots as well as a screencast to showcase the scenario using our prototype based on OpenTOSCA. The TOSCA modeling tool Winery can be used to model topologies and to export them as CSARs [22]. The OpenTOSCA Container is a TOSCA-compliant runtime that can process CSARs and deploy the applications accordingly [23]. All extension in the course of this work has been merged into the MASTER of the individual source code repositories. We used Winery to create five Deployment Test Annotations and to decorate Node Templates with them. Our prototype supports Management Operation Tests and four Domain-specific Tests: (i) *HttpTest*, (ii) *TcpPingTest*, (iii) *PortBindingTest*, and (iv) *SqlConnectionTest*. We created these types and made them available in our TOSCA definitions repository. Furthermore, we extended Winery's graphical Topology Modeler component to drag and drop the annotations onto Node Templates. Therefore, we filter for our introduced annotations namespace "http://opentosca.org/policytypes/annotations/tests," because only these should be available to decorate Node Templates. With that, an application developer is now able to decorate a Node Template. The Topology Modeler creates a new Policy, attaches it to the Node Template and links to an UI, where annotation-specific properties can be entered (cf. Fig. 3). For the execution part, we extended the OpenTOSCA Container with the pictured Test Executor component. The component is working based on plugins to generate and execute the actual test cases. The Test Executor is triggered after a build plan has been executed and creates 1) a `Test Context`, 2) determines the information about the current CSAR to test, 3) as well as the created instance data. In a threaded way, the Test Executor component schedules and launches the test cases as separate jobs. Before it runs the tests, the component determines if there is a registered plugin to handle the used annotations. Therefore, all Node Templates and Annotations are checked by each plugin. A plugin calls the `canExecute()` method to check whether it can execute. If `true` is returned, a job is scheduled and run as soon as the Test Executor has the capacities. We created five plugins for our prototype. One plugin that can handle Management Operation Tests and four specific plugins to handle the Domain-specific Tests outlined above. The test execution as well as the result of each test case is stored in a database aside to the respective instance data. We implemented a RESTful HTTP API to query the results for the related service instances. Finally, we enriched OpenTOSCA's user interface in order to show the results and to visually indicate whether the test execution was successful.

[2]Eclipse Winery: https://github.com/eclipse/winery

[3]OpenTOSA Container: https://github.com/OpenTOSCA/container

VI. Related Work

Extensive research has been conducted on automated software testing in terms of test design, test execution, test coverage, and test result analysis. One approach in software testing is *model-based testing* (MBT) where models of a system under test are employed to derive and to generate test cases for the system [24]. For example, once tests are derived, they can be directly run by a runtime or transformed into artifacts for execution. Model-based testing has a long history in software engineering to assure quality [25], [26]. Different test case generation techniques have been developed and researched in the past years. Based on MBT, there are approaches to automate security testing [27], to generate tests based on requirements [28], to define and perform system testing using UML [29], to test GUI-based web applications [30], [31], as well as the automated performance testing [32]. However, these approaches are usually not integrated with the software delivery process [33]. More recent work [34] tries to integrate functional test automation methods with the software delivery processes and CI/CD servers. Anyhow, this approach assumes an already running application inside a testing environment. In our work, we introduce a model-based deployment testing approach that is tightly coupled with the provisioning system itself. Testing is therefore integrated by design and executed continuously based on the final deployment result.

Test automation in DevOps plays vital role [35]. Usually, different test types target different deployment environments. There are prominent examples, such as unit tests, integration tests, and end-to-end tests, that typically target development and test environments. However, there are also approaches for production environments, such as smoke tests or synthetic tests. The aim is to run a set of functional tests under low load and with simulated user inputs to ensure that major features of the applications work as expected. Here, the test specification is decoupled from the actual deployment model and usually maintained in a different location. Furthermore, a survey of software testing in the cloud showed that acceptance testing as well as interoperability testing is not thoroughly studied [36]. Anyhow, there are research activities to test and validate the idempotency of infrastructure as code [37]. Their focus is rather on the infrastructure level than on the application level and, hence, do not test the final orchestrated application as we do.

To the best of our knowledge no published work uses MBT for deployment models. Therefore, we introduced a concept to model and maintain test specifications along with the actual deployment model. Our concept further increases the level of automation in testing deployment models in the context of DevOps. In this work, the meta-model is abstracted from

the DMMN (Declarative Application Management Modeling and Notation) meta-model [38]. We extend the DMMN meta-model with an *Annotation* class and show how this class is used to express Deployment Tests as well as how it is related to an application topology and its elements. Annotations in general are used to describe or add additional contextual information to the elementes of an application topology. For example, as also introduced by the authors of GENTL [39]—a description language for Cloud Application Topologies—, annotations convey additional information, such as metering, billing, management, or even test specification information. We used the DMMN meta-model and its graph-based nature since it is abstract and generic enough to be mapped to different kinds of graph-based languages, also shown and exploited by Saatkamp et al. [40]. Representing an application topology as a graph is a common approach in research. For example and among others[4], Andrikopoulos et al. [39] as well as TOSCA [7], [41] propose a graph-based description language for application topologies.

VII. Conclusion and Future Work

In this paper, we presented a concept to enable modeling and automating the execution of deployment tests. The presented approach enables eliminating manual and error-prone testing effort by a simple declarative approach, which requires no deep technical understanding of the component to be tested but only about the test itself. The approach is important to be integrated in continuous software delivery processes in order to not only check the technical success of an application deployment, but also if the business functionality works properly. We showed how our concept can be applied to TOSCA and demonstrated in detail—based on a concrete use case scenario—how to model and serialize such deployment tests. Furthermore, we proposed an extended system architecture of a TOSCA runtime in order to execute such tests. On top of that, we implemented our TEST4TOSCA extension and integrated it into the open source TOSCA ecosystem OpenTOSCA.

In future work, we plan to automatically generate test plans based on TOSCA topology models to eliminate the task of manually specifying deployment tests. Furthermore, we envision the concept of *Mimic Tests* where components are imitated in order to test relations between components.

Acknowledgment

This work is partially funded by the Federal Ministry for Economic Affairs and Energy (BMWi) project *Smart-Orchestra* (01MD16001F) and *Industrial Communication for Factories* (01MA17008G).

[4]An overview and comparison of different cloud modeling languages and their deployment modeling capabilities is presented by Bergmayr et al. [6].

References

[1] J. Humble and J. Molesky, "Why Enterprises Must Adopt Devops to Enable Continuous Delivery," *Cutter IT Journal*, 2011.

[2] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in *USITS*, 2003.

[3] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, and J. Wettinger, "Integrated Cloud Application Provisioning: Interconnecting Service-Centric and Script-Centric Management Technologies," in *CoopIS 2013*, 2013.

[4] Opscode, Inc., "Chef Official Site," 2018. [Online]. Available: http://www.opscode.com/chef

[5] Red Hat, Inc., "Ansible Official Site," 2018. [Online]. Available: https://www.ansible.com

[6] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, and G. Kappel, "A Systematic Review of Cloud Modeling Languages," *ACM Computing Surveys (CSUR)*, 2018.

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

[8] J. Wettinger *et al.*, "Unified Invocation of Scripts and Services for Provisioning, Deployment, and Management of Cloud Applications Based on TOSCA," in *CLOSER*, 2014.

[9] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, and J. Wettinger, "Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications," in *PATTERNS*, 2017.

[10] U. Breitenbücher *et al.*, "Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA," in *IC2E*, 2014.

[11] Amazon Web Services, Inc., "AWS CloudFormation Official Site," 2018. [Online]. Available: https://aws.amazon.com/de/cloudformation

[12] S. Wang, I. Keivanloo, and Y. Zou, "How Do Developers React to RESTful API Evolution?" in *Service-Oriented Computing*, 2014.

[13] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Migrating to Cloud-Native Architectures Using Microservices: An Experience Report," in *Advances in Service-Oriented and Cloud Computing*, 2016.

[14] M. Wurster, U. Breitenbücher, M. Falkenthal, and F. Leymann, "Developing, Deploying, and Operating Twelve-Factor Applications with TOSCA," in *iiWAS*, 2017.

[15] A. C. F. da Silva, U. Breitenbücher, P. Hirmer, K. Képes, O. Kopp, F. Leymann, B. Mitschang, and R. Steinke, "Internet of Things Out of the Box: Using TOSCA for Automating the Deployment of IoT Environments," in *CLOSER*, 2017.

[16] K. Saatkamp, U. Breitenbücher, F. Leymann, and M. Wurster, "Generic Driver Injection for Automated IoT Application Deployments," in *iiWAS*, 2017.

[17] J. Wettinger, U. Breitenbücher, and F. Leymann, "Standards-Based DevOps Automation and Integration Using TOSCA," in *UCC*, 2014.

[18] J. Wettinger, M. Behrendt, T. Binz, U. Breitenbücher, G. Breiter, F. Leymann, S. Moser, I. Schwertle, and T. Spatzier, "Integrating Configuration Management with Model-Driven Cloud Management Based on TOSCA," in *CLOSER*, 2013.

[19] T. Waizenegger, M. Wieland, T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, B. Mitschang, A. Nowak, and S. Wagner, "Policy4TOSCA: A Policy-Aware Cloud Service Provisioning Approach to Enable Secure Cloud Computing," in *OTM*, 2013.

[20] K. Képes, U. Breitenbücher, M. P. Fischer, F. Leymann, and M. Zimmermann, "Policy-Aware Provisioning Plan Generation for TOSCA-based Applications," in *SECURWARE*, 2017.
[21] OASIS, *TOSCA Simple Profile in YAML Version 1.0*, 2015.
[22] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, "Winery – A Modeling Tool for TOSCA-based Cloud Applications," in *ICSOC*, 2013.
[23] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "OpenTOSCA – A Runtime for TOSCA-based Cloud Applications," in *ICSOC*, 2013.
[24] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification and Reliability*, 2012.
[25] I. K. El-Far and J. A. Whittaker, *Model-Based Software Testing*. John Wiley & Sons, Inc., 2002.
[26] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz, "Model-based Testing in Practice," in *ICSE*, 1999.
[27] M. Blackburn, R. Busser, A. Nauman, and R. Chandramouli, "Model-based Approach to Security Test Automation," in *Quality Week*, 2001.
[28] L. H. Tahat, B. Vaysburg, B. Korel, and A. J. Bader, "Requirement-based automated black-box test generation," in *COMPSAC*, 2001.
[29] L. Briand and Y. Labiche, "A UML-Based Approach to System Testing," *Software and Systems Modeling*, 2002.
[30] M. Katara, A. Kervinen, M. Maunumaa, T. Paakkonen, and M. Satama, "Towards Deploying Model-Based Testing with a Domain-Specific Modeling Approach," in *TAIC PART'06*, 2006.
[31] M. Nabuco and A. C. R. Paiva, "Model-Based Test Case Generation for Web Applications," *ICCSA*, 2014.
[32] J. Zhou, B. Zhou, and S. Li, "Automated Model-Based Performance Testing for PaaS Cloud Services," in *COMPSACW*, 2014.
[33] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A Survey on Model-based Testing Approaches: A Systematic Review," in *WEASELTech*, 2007.
[34] L. M. Hillah, A.-P. Maesano, F. De Rosa, L. Maesano, M. Lettere, and R. Fontanelli, "Service functional test automation," in *STV*, 2015.
[35] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.
[36] K. Inçki, I. Ari, and H. Sözer, "A Survey of Software Testing in the Cloud," in *SERE-C*, 2012.
[37] W. Hummer, F. Rosenberg, F. Oliveira, and T. Eilam, "Testing Idempotence for Infrastructure as Code," in *Middleware*, 2013.
[38] U. Breitenbücher, "Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements," Ph.D. dissertation, 2016.
[39] V. Andrikopoulos, A. Reuter, S. Gómez Sáez, and F. Leymann, "A GENTL Approach for Cloud Application Topologies," in *Service-Oriented and Cloud Computing*, 2014.
[40] K. Saatkamp, U. Breitenbücher, O. Kopp, and F. Leymann, "Topology Splitting and Matching for Multi-Cloud Deployments," in *CLOSER*, 2017.
[41] T. Binz, G. Breiter, F. Leymann, and T. Spatzier, "Portable Cloud Services Using TOSCA," *IEEE Internet Computing*, 2012.