Institute of Architecture of Application Systems

An Approach to Automatically Check the Compliance of Declarative Deployment Models

Christoph Krieger, Uwe Breitenbücher, Kálmán Képes, Frank Leymann

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

BibTeX :

```
@InProceedings{Krieger2018_DeploymentComplianceRules,
  author    = {Christoph Krieger and Uwe Breitenb{\"u}cher and
               K{\'a}lm{\'a}n K{\'e}pes and Frank Leymann},
  title     = {{An Approach to Automatically Check the Compliance of
                Declarative Deployment Models}},
  booktitle = {Papers from the 12\textsuperscript{th} Advanced Summer
               School on Service-Oriented Computing (SummerSoC 2018)},
  year      = {2018},
  pages     = {76--89},
  publisher = {IBM Research Division}
}
```

The full version of this publication has been presented as a poster at the Advanced Summer School on Service Oriented Computing (SummerSoC 2018).
http://www.summersoc.eu

University of Stuttgart
Germany

An Approach to Automatically Check the Compliance of Declarative Deployment Models

Christoph Krieger, Uwe Breitenbücher, Kálmán Képes, and Frank Leymann

Institute of Architecture of Application Systems, University of Stuttgart
Universitätsstr. 38, 70569 Stuttgart, Germany
{firstname.lastname}@iaas.uni-stuttgart.de

Abstract. The automation of application deployment has evolved into one of the most important issues in modern enterprise IT. Therefore, many deployment systems have been developed that process deployment models for automating the installation of systems. Creating such deployment models becomes more and more complex as compliance plays an increasingly important role. Not only external laws and regulations must be considered, but also a company's internal requirements must be fulfilled. However, this is a very complex challenge for the modelers as they require a firm knowledge of all the compliance rules that must be observed. As a result, this often leads to deployment models that violate compliance rules due to manual modeling mistakes or because of unawareness. In this paper, we introduce an approach that enables modeling of reusable Deployment Compliance Rules that can be executed automatically to check such regulations in declarative deployment models at design time. We validate our approach with a prototype based on the TOSCA standard and the OpenTOSCA ecosystem.

Keywords: Cloud Computing, Compliance, Deployment Modeling.

1 Introduction

There are many laws, regulations, and guidelines that influence an enterprise's information technology (IT), such as the General Data Protection Regulation [1], the ISO 2018 standard [2], or company internal regulations. Threats to security through the use of outdated software versions or financial risks due to the use of unlicensed software also have an impact on IT landscapes. Moreover, modern application landscapes often consist of complex composite applications and different heterogeneous systems [3]. Such large landscapes and systems present a challenging and expensive task to be maintained and managed manually. To decrease the costs and to minimize human errors [4], the automation of deployment and provisioning of applications has become an important subject in academia and industry. Approaches for deployment automation systems, such as Chef [5], Juju [6], Ansible [7], and Kubernetes [8], consume deployment models that describe the desired deployment and automatically execute all required tasks. However, enterprise's applications and IT landscapes are subject to a magnitude of requirements. If a company has a vast amount of requirements, it

would require great effort and expertise of modelers of deployment models to know all rules and regulations that concern a particular application deployment. Moreover, rules are subject to change as new requirements and needs arise, and others are suspended. Not being compliant to all necessary rules can quickly result in serious consequences for companies, such as lawsuits due to unlicensed software or even the loss of customers due to leaks of personal data and subsequent trust issues by the customers. Thus, checking deployment models for compliance with a company`s requirements is of vital importance, but practically not possible if performed manually by modelers and operators.

In this paper, we present an approach that automates compliance checking of declarative deployment models to ensure that new or updated deployment models are always compliant with a company's set of constraints. To achieve this, we present an approach to specify reusable Deployment Compliance Rules that enables automated compliance checking of declarative deployment models at design time, which decreases the chance of application deployments that are not compliant to a company's regulations. Furthermore, the approach separates the modeling of compliance rules from the modeling of deployment models. This ensures that modelers do not need to know all constraints and requirements to specify compliant deployment models.

The remainder of this paper is structured as follows: Section 2 motivates our approach, followed by Section 3 which presents the main concept. Section 4 provides a formal model for the approach. Section 5 describes the prototypical implementation. In Section 6, we discuss related work. Finally, Section 7 concludes the paper and describes future work.

2 Motivation Scenario

Declarative deployment models describe the structure of an application to be deployed by specifying the required components, their relationships, as well as all artifacts required for the deployment, thus, the *topology* of the application [9]. A topology is a directed, weighted, and possibly disconnected graph consisting of nodes representing the components of the application and edges representing the relationships between them. Types associated with the components and relations specify the desired semantics. In addition, attributes associated with components or relations represent properties, such as ports or network addresses for servers. Figure 1 illustrates a topology describing the deployment of an application. The left-hand side shows a stack consisting of a component of type *Web-application* which is hosted on a Tomcat server of type *Tomcat8.5.23* as indicated by the *hostedOn* relation between the two components. The Tomcat server is hosted on a component of type *Ubuntu16.04VM*, which is hosted on a component of type *AmazonEC2*. The web application is connected to a component of type *MySQLDB5.0*, which provides persistent data storage. The attribute with key *DataType* and value *PersonalData* associated to the component indicates the type of data stored in this database. The database is managed through a Database Management System (DBMS) of type *MySQLDBMS5.0* that is also hosted on a component of type *Ubuntu16.04VM*. However, this Ubuntu VM is hosted on an OpenStack cloud with a specific IP address, as indicated by the IP attribute associated

with the component. Thus, this deployment scenario describes a hybrid cloud application deployment which is partly hosted in a private cloud (OpenStack) and partly in a public cloud (Amazon EC2) [10]. In our scenario, the web application is a stateless component hosted on Amazon EC2 to be scaled out automatically based on the workload. The web application retrieves data from the MySQL database, performs processing tasks, and stores the results back to the database.

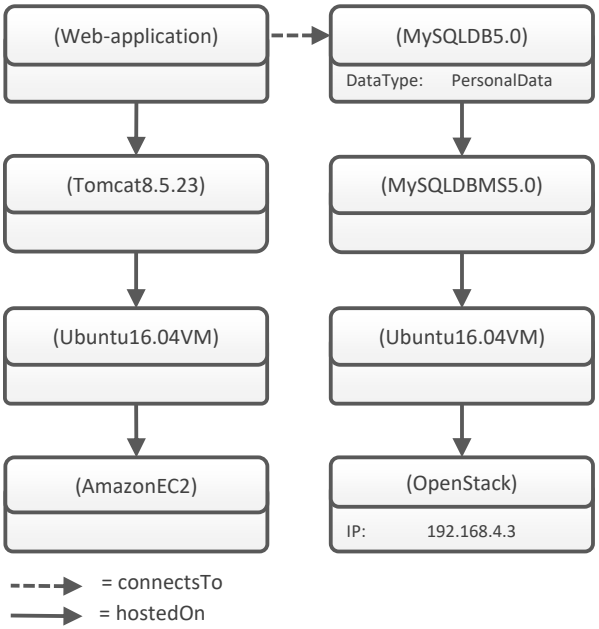

Fig. 1. A declarative deployment model describing the deployment of an application

Deployment models can be subject to a variety of constraints and rules which must be adhered to. However, it is difficult for modelers to be aware of all regulations, which quickly results in non-compliant deployment models that violate a company's rules. For the presented deployment model, there may be several rules that must be adhered to. In the scenario, the database contains personal data, i.e., customer names, which is sensitive data. A company could decide that all personal data must only be stored in a specific private cloud. The application uses components that should receive regular updates since old versions of software artifacts often present security risks. In our scenario, this is relevant for the MySQL database and management system, the Tomcat server, and the Ubuntu VMs. For example, an outdated Tomcat could expose vulnerabilities that have been fixed by later versions, such as remote code execution. In this paper, we present an approach to express such regulations as reusable compliance rules that can be automatically checked at design time.

3 Concept to Automatically Check the Compliance of Declarative Deployment Models

An overview of the included roles, components, and tasks of the approach to automatically check the compliance of declarative deployment models is shown in Figure 2. There, the *Compliant Deployment Modeling System* is divided into the two separate areas *Compliance Modeling* and *Deployment Modeling*. The system separates concerns, as the expertise of compliance experts and deployment experts is integrated in an automated fashion without the need to exchange knowledge between the involved roles.

The left-hand side of Figure 2 shows the involved roles, components, and tasks of the Compliance Modeling area. Compliance experts use *the Compliance Modeling Tool* for the definition and maintenance of *Deployment Compliance Rules*, which formally describe and capture all regulations that must be fulfilled by deployment models. These rules are stored persistently in a *Compliance Rule Repository* within the Compliance Modeling Tool. The stored rules provide a means to detect potentially relevant areas of deployment models and check their compliance by comparing them to a compliant fragment. A respective method will be elaborated in the next sections.

Deployment Modeling is concerned with the definition and maintenance of deployment models. The right-hand side of Figure 2 shows the involved roles, components, and tasks of the Deployment Modeling area. Deployment experts use the *Deployment Modeling Tool* to define and maintain deployment models that are stored persistently in a *Deployment Model Repository*. The deployment experts do not have to be aware of all regulations concerning deployment models since the Deployment Modeling Tool uses the *Compliance Checker* to verify the compliance of declarative deployment models based on the stored compliance rules. This ensures that only compliant deployment models are stored in the repository.

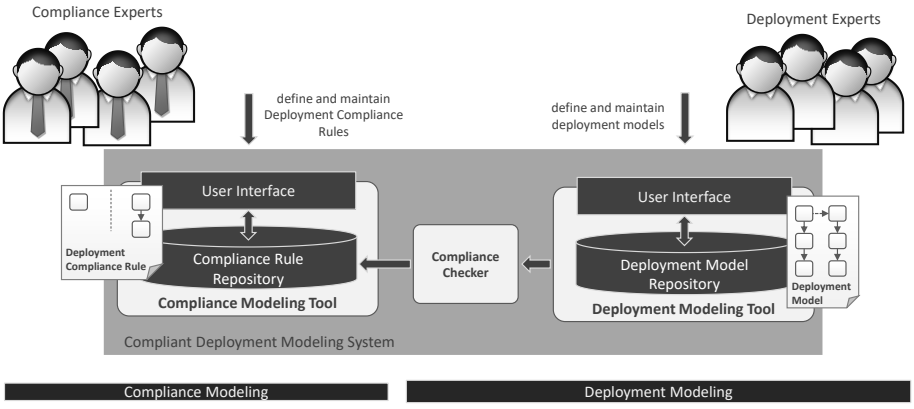

Fig. 2. Overview of roles, components, and tasks of the approach

4 Metamodel and Formalization

In our previous work [11], we have conceptually introduced Deployment Compliance Rules. In the context of this paper, we will present a formal metamodel for the definition of such Deployment Compliance Rules. For this, in Section 4.1, we first provide a formal metamodel for declarative deployment models, which is based on topologies. Section 4.2 provides the metamodel of Deplyoment Compliance Rules, while Section 4.3 describes the algorithm used for automated compliance checking.

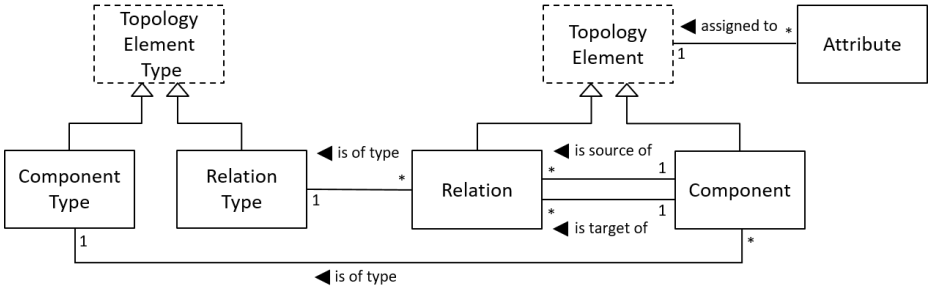

Fig. 3. Metamodel of topologies

4.1 Basic Metamodel of Topologies

The metamodel of topologies, is based on the Declarative Application Management Modeling and Notation (DMMN) introduced by Breitenbücher [12] and further abstracted by Saatkamp et al. [13]. Figure 3 gives an overview of the metamodel. There, class names contain a starting capital letter.

Let T be the set of all Topologies, then $t \in T$ is defined as an eight-tuple:

$$t = (C_t, R_t, CT_t, RT_t, A_t, type_t, attributes_t, supertype_t) \quad (1)$$

The elements of t are defined as follows:

- C_t: The set of Components in t, whereby each $c_i \in C_t$ represents a component of the application to be deployed.
- $R_t \subseteq C_t \times C_t$: The set of Relations in t, whereby each $r_i = (c_s, c_t) \in R_t$ represents the relationship between two components, where c_s is the source and c_t is the target of the relationship.
- CT_t: The set of Component Types in t, whereby each $ct_i \in CT_t$ describes the semantics for the Components that have this Component Type assigned.
- RT_t: The set of Relation Types in t, whereby each $rt_t \in RT_t$ describes the semantics for the Relations that have this Relation Type assigned.
- $A_t \subseteq \Sigma^+ \times \Sigma^+ \times \Sigma^+$: The set of Attributes in t, whereby each $a_i = (Id, Key, Value) \in A_t$ describes a property of a Component or Relation with a key and a value. Each Id must be unique within a Topology. Id, Key and $Value \in \Sigma^+$ are typically strings.

- $type_t$: The mapping that assigns all Relations and Components in t to their Relation Type or Component Type. Let the set of Topology Elements $E_t \coloneqq C_t \cup R_t$ be the union of the set of Components and the set of Relations of t, and the set of Topology Element Types $ET_t \coloneqq CT_t \cup RT_t$ be the union of the set of Component Types and the set of Relation Types of t. Then, the mapping $type_t$ associates each $e_i \in E_t$ with an $et_j \in ET_t$ to provide the semantics for each Topology Element.

$$type_t: E_t \to ET_t \tag{2}$$

- $attributes_t$: The mapping that assigns each Topology Element to a set of Attributes.

$$attributes_t: E_t \to \wp(A_t) \tag{3}$$

- $supertype_t$: The mapping that assigns Relation Types and Component Types to their respective supertype. Let ET_t be the set of Topology Element Types of t. Then, the mapping $supertype_t$ associates an $et_i \in ET_t$ with an $et_j \in ET_t$ with $i \neq j$. This means that et_j is the supertype of et_i.

$$supertype_t: ET_t \to ET_t \tag{4}$$

Additionally, we define the mapping $supertypes_t$ that maps a Topology Element of t to its respective Topology Element Type specified by $type_t$ combined with all transitively resolvable supertypes of $type_t$. Let E_t be the set of Topology Elements and ET_t be the set of Topology Element Types of t.

$$supertypes_t: E_t \to \wp(ET_t) \tag{5}$$

4.2 Metamodel of Deployment Compliance Rules

In the following, we will provide a formal metamodel for Deployment Compliance Rules. Additionally, we define the conditions under which such a rule is *valid*, *detected*, or *satisfied*. The concept of automated compliance checking based on Deployment Compliance Rules is illustrated in Figure 4. There, the left-hand side of the figure illustrates a declarative deployment model, the right-hand side illustrates a Deployment Compliance Rule. A Deployment Compliance Rule consists of two Topologies called Detector and Required Structure. The Detector is used to detect areas in the deployment model that are subject to the rule while the Required Structure is used to verify if the rule is satisfied, i.e., if the rule is fulfilled. The rule shown in Figure 4 is concerned with the storage of personal data in a specific private cloud as discussed in the motivation scenario presented in Section 2. It specifies that any database that stores personal data has to be managed by a database management system (DBMS).

The DBMS must in turn be hosted in a virtual machine provided by a specific OpenStack cloud.

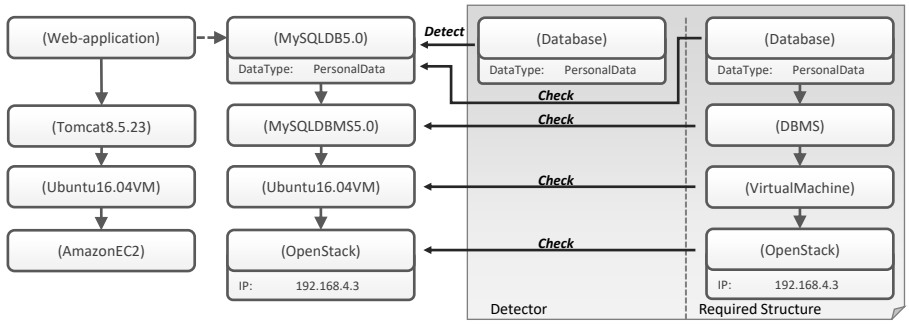

Fig. 4. Concept of detection and checking of Deployment Compliance Rules in declarative deployment models

Let $CR \subseteq T \times T$ be the set of all Deployment Compliance Rules, then $cr \in CR$ is defined as:

$$cr \in CR := (Detector,\ RequiredStructure) \tag{6}$$

The $Detector$ and $RequiredStructure$ are provided as a Topology, i.e., $Detector \in T$ and $RequiredStructure \in T$.

The detection and checking of Deployment Compliance Rules is based on the detection of subgraph isomorphisms that also considers the types and attributes of the Topology Elements that form the deployment model. Subgraph isomorphisms can be detected by using the VF2 algorithm described by Cordella et al. [14]. We omit the VF2 algorithm here and refer interested readers to the provided reference. The detection of relevant areas in a deployment model is formally the detection of subgraph isomorphisms. To detect semantically compatible subgraph isomorphisms, i.e. to find areas with the same structure, types, and attributes, we define a matching relation to decide if a Topology Element can be matched to another Topology Element. Let $e_i \in E_t$ and $e_j \in E_t$ with $i \neq j$ be two Topology Elements in t. Then the matching relation $\equiv_e$ is defined as follows:

$$e_1 \equiv_e e_2 :\Leftrightarrow \left(type_t(e_1) \in supertypes_t(e_2)\right) \wedge \Big(\forall a_i \in attributes_t(e_1) \exists a_j \in attributes_t(e_2) :\ \pi_2(a_i) \Leftrightarrow \pi_2(a_j) \wedge \pi_3(a_i) \Leftrightarrow \pi_3(a_j)\Big) \tag{7}$$

A Topology Element e_1 can be matched to a Topology Element e_2 if the type of e_1 is of the same type or supertype of the type of e_2. Furthermore, all attributes of e_1 must also be present in e_2 with equivalent Key and $Value$. For example, the Component with Component Type "Database" shown in Figure 4 can be matched to the Component with Component Type "MySQLDB5.0". The "Database" Component has an attribute with Key = "$DataType$" and $Value$ = "PersonalData". The equivalent attribute, i.e., same Key and $Value$, is also assigned to the "MySQLDB5.0" Compo-

nent. In addition, the Component Type “Database” is a supertype of the Component Type “MySQLDB5.0”.

Let E be the set of all Topology Elements. Then $m \in M, m \subseteq E \times E$ is defined as a bijective mapping that maps a Topology Element $e_i \in E$ to another Topology Element $e_j \in E$ with $i \neq j$:

$$M := \wp(E \times E) \setminus \emptyset \tag{8}$$

A matching mapping is a mapping between two Topologies that preserves the structure and semantics of the Topologies and the individual Topology Elements by also considering the matching relation $\equiv_e$. A mapping $m \in M$ is a matching mapping from $t_1 \in T$ to $t_2 \in T$ if t_1, t_2, and m fulfill the relation $matching \subseteq T \times T \times M$:

$$(t_1, t_2, m_1) \in matching :\Leftrightarrow \left(m_1 \subseteq E_{t_1} \times E_{t_2}\right) \wedge \left(\forall c_i \in C_{t_1} \exists! \, c_j \in C_{t_2} : \left(c_i, c_j\right) \in m_1 \wedge c_i \equiv_e c_j\right) \wedge \left(\forall r_k \in R_{t_1} \exists! \, r_l \in R_{t_2} : (r_k, r_l) \in m_1 \wedge r_k \equiv_e r_l \wedge \left(\pi_1(r_k), \pi_1(r_l)\right) \in m_1 \wedge \left(\pi_2(r_k), \pi_2(r_l)\right) \in m_1\right) \tag{9}$$

The mapping m_1 represents a subgraph isomorphism from t_1 to t_2 that also considers the matching relation $\equiv_e$. Each Component in t_1 is mapped to exactly one matching Component in t_2, to which no other Component in t_1 has been mapped to. Analogously each Relation in t_1 is mapped to exactly one matching Relation in t_2 with the addition that the relations sources and targets have also been mapped to each other and therefore the mapping preserves the structure of the topologies. For example, in Figure 4, the Required Structure can be matched completely to the right-hand stack of the Topology since all indicated component pairs fulfill the matching relation $\equiv_e$ and both stacks are structurally identical.

A Deployment Compliance Rule cr_1 is *valid* if there is exactly one matching mapping from the Detector to the Required Structure and the following holds:

$$\exists! \, (\pi_1(cr_1), \pi_2(cr_1), m) \in matching \tag{10}$$

This ensures that each matching mapping found for the Required Structure corresponds to a matching mapping found for the Detector.

A Deployment Compliance Rule cr_1 is *detected* in a Topology t_1 if there is at least one matching mapping from the rule’s Detector $\pi_1(cr_1)$ to the Topology:

$$\exists (\pi_1(cr_1), t_1, m) \in matching \tag{11}$$

For a Deployment Compliance Rule cr_1 to be *satisfied* for a Topology t_1, there has to be exactly one corresponding matching mapping from the Required Structure $\pi_2(cr_1)\ to\ t_1$ for each matching mapping form the Detector $\pi_1(cr_1)\ to\ t_1$:

$$\forall (\pi_1(cr_1), t_1, m_i) \in matching \; \exists! \left(\pi_2(cr_1), t_1, m_j\right) \in matching : \Big(\forall e_k \in \pi_2(m_i) : e_k \in \pi_2\big(m_j\big)\Big) \tag{12}$$

Each area that has been found in t_1 by mapping the Detector to t_1 also satisfies the Required Structure and therefore, the Deployment Compliance Rule cr_1 is *satisfied*, i.e., the rule is fulfilled. For example, the rule in Figure 4 is *detected* in the declarative deployment model as there is a matching mapping from the rule's Detector to the Topology of the deployment model. Since there exists exactly one corresponding matching mapping from the Required Structure to the Topology for each matching mapping from the Detector to the Topology, the rule is also *satisfied*.

4.3 Compliance Checking Algorithm

Based on the presented metamodel, an algorithm is proposed to automatically check a Topology for a given Deployment Compliance Rule. This algorithm is described in Algorithm 1 in pseudocode. It is executed once for every Deployment Compliance Rule that is defined for a Topology. The algorithm uses a Deployment Compliance Rule and a Topology as inputs. The result is a set containing matching mappings of the Detector to the Topology to indicate rule violations. If the set is empty, the given Deployment Compliance Rule is satisfied for that Topology. In the case, that a Deployment Compliance Rule is not detected for a Topology, the algorithm also returns an empty set indicating that the rule is satisfied. The given Topology is analyzed for matches to both the Detector (line 3) and the Required Structure. Subsequent, all mappings found for the Detector are checked for a corresponding mapping for the Required Structure (line 5). If none is found, the Detector's mapping is added to the result set *R* (line 6). At the end of the algorithm, *Result* contains mappings to areas where the given Deployment Compliance Rule is violated. If *Result* is empty, the rule is satisfied for the given Topology.

procedure: $findViolations(cr \in \mathcal{CR},\ t \in \mathcal{T})$

1: $Result := \emptyset$
2: *//find all matching mappings for the Detector of cr*
3: **for all** $((\pi_1(cr_1), t, m_i) \in matching)$ **do**
4: *//find a matching mapping for the Required Structure of cr*
5: **if** $\nexists(\pi_2(cr_1), t, m_j) \in matching : (\forall e_k \in \pi_2(m_i) : e_k \in \pi_2(m_j))$ **then**
6: $Result := Result \cup m_j$
7: **end if**
8: **end for**
9: **return** $Result$

Algorithm 1. Pseudocode to identify Deployment Compliance Rule violations in a Topology

5 Prototypical Implementation

The prototype described in this section implements the metamodel for Topologies and Deployment Compliance Rules as well as the algorithm introduced in Section 4. Furthermore, the prototype enables compliance experts and deployment experts to model Deployment Compliance Rules, Topologies, Component Types, and Relation Types with a graphical user interface while also providing the functionality to check the Topologies for compliance.

The prototype consists of a graphical user interface, a compliance checker, a model repository, and an application programmer interface (API). In our prototype, we combine the Compliance Checker, the Compliance Modeling Tool, and the Deployment Modeling Tool introduced in Section 3 to one component, the Modeling Tool. The graphical user interface enables modelers to create and update deployment models as well as Deployment Compliance Rules. The user interface also enables the definition of reusable component and relation types and their hierarchical structure. The compliance checker provides the functionality to validate Deployment Compliance Rules and to check the compliance of deployment models. For that purpose, the compliance checker has access to the model repository. The model repository is a combination of the Compliance Rule Repository and the Deployment Rule Repository introduced in Section 3. It is used to persistently store deployment models, Deployment Compliance Rules, component types, and relation types. The API of the modeling tool provides external access to the functionality of the modeling tool, such as the storing and checking of deployment models.

Since our prototype is based on TOSCA and the OpenTOSCA ecosystem [15], we briefly introduce the TOSCA standard and provide a mapping from the formal metamodel to TOSCA. TOSCA is an OASIS standard that enables the specification of cloud applications by defining their structure and their orchestration. We only introduce constructs in TOSCA that are necessary for our method and refer interested readers to the specification [16], the simple profile [17], and the primer [18]. A TOSCA *Topology Template* represents the structure of an application. It specifies all components needed for the deployment of an application as well as the relationships between them. The *Topology Template* is a directed graph with typed nodes and weighted edges and corresponds to a Topology defined in the metamodel. *Node Templates* and *Relationship Templates* represent the Components and Relations of the metamodel. Component Types and Relation Types can be mapped to *Node Types* and *Relationship Types* of TOSCA. The attributes defined by the metamodel can be mapped to *Properties*. In TOSCA, Node Types can be specified with a *PropertiesDefinition* that defines the structure of possible properties for the Node Templates. The actual attribute values are specified by Node Templates. Since TOSCA allows extensions to the specification, we introduce the new element *ComplianceRule* that has exactly two Topology Templates as elements: *Detector* and *RequiredStructure*. Therefore, we have a complete mapping from the metamodel to TOSCA.

The prototypical implementation extends Winery [19][1], which is a graphical modeling tool for modeling and managing applications using TOSCA. Winery already provides a mechanism for persistent storage of TOSCA elements which was extended

[1] http://eclipse.github.io/winery

to also store Deployment Compliance Rules. Additionally, we added the new component *Compliance Checker* to Winery which realizes the concepts presented in this paper. To associate TOSCA deployment models with Deployment Compliance Rules, we use the concept of namespaces, i.e., all Topology Templates must be checked for all Deployment Compliance Rules that are in the same namespace. Therefore, each rule defined for a certain namespace automatically applies to all Topology Templates defined in that namespace.

6 Related Work

In this section, we discuss and compare works that are related to our method. Software architecture is often described with 5 views first described by Kruchten [20]. However, there can still be other views on the architecture of a software system, such as the view on the deployment model of a system. Software architecture erosion [21] describes the deviation of actual software artifacts from their architecture that manifests over time. It has been addressed with various methods such as model-driven approaches, through dependency analysis, or through checking mechanisms, such as the reflexion models approach by Murphy et al. [22]. They describe high-level models, i.e. boxes and arrows that are used by software architects to describe and reason about the architecture of a software artifact. To test the compliance with the architecture, they generate a low-level model from existing artifacts through the use of code-analysis and compare the two models to find convergences, divergences, and absences, e.g., correct, additional, and missing connections between components. Koschke and Simon extended this approach by introducing hierarchical reflexion models [23]. The method uses an architecture model to express implicit requirements and constraints for the resulting software artifact. With our method, we provide reusable rules, that can be applied to deployment models in general. Other approaches to control software architectures include Architecture Description Languages [24] to describe architectures or Architecture Constraint Languages [25] to express constraints. Deiters et al. [26] introduce Rule-Based Architectural Compliance Checks for Enterprise Architecture Management that are expressed as queries. However, these approaches for describing constraints to check the compliance of software artifacts to their intended architecture are application specific and on the level of artifacts. With our method, we enable to specify compliance rules on the level of deployment models in a very generic manner. Hence, they can be used to check the deployment models of different applications.

There are many works in the area of business process verification to ensure that business process models adhere to regulations. Due to the great number of works we refer interested readers to a survey on business process compliance by Fellmann and Zasada [27]. Schleicher et al. [28] introduce an approach to express control-flow and data-related compliance rules for business processes while Koetter et al. [29] introduce a generic Compliance Descriptor that links compliance rules to their source. Tran et al. [30] introduce an approach that enables compliance modeling for service-oriented systems through the use of domain-specific languages to model compliance for different areas. Liu et al. [31] propose to separate the modeling of business pro-

cesses from the modeling of compliance rules. In their method, they use the Business Process Execution Language (BPEL) [32] as a model for business processes and the Business Property Specification Language (BPSL) [33] to specify their compliance rules. They use established model checkers based on linear temporal logic to verify process models with their rules. These compliance rules deal mostly with temporal aspects in the execution of business processes while our method provides a compliance checking mechanism for deployment models that can be used to address issues, such as outdated software versions or structural properties, i.e., how components may be hosted under certain circumstances.

7 Conclusion & Future Work

In this paper, we presented Deployment Compliance Rules that enable to describe restrictions, constraints, and requirements for declarative deployment models in a reusable manner. We provided a formalized model for Deployment Compliance Rules and an algorithm to automatically check if a deployment model violates a given rule. Based on this, we presented an approach to ensure that created or updated deployment models are compliant to the current set of Deployment Compliance Rules. Further, the approach allows separating the definition of Deployment Compliance Rules from the creation and maintenance of deployment models. A validation of the approach is provided via the implementation of a TOSCA-based prototype although the provided formalization enables to apply the approach to any graph-based and declarative deployment model language. In the future, we plan to extend the approach to also verify the consistency of the rule repository through detection of conflicting rules within the repository.

Acknowledgments. This work was partially funded by the German Research Foundation (DFG) project ADDCompliance (636503) and the BMWi project SmartOrchestra (01MD16001F).

References

1. General Data Protection Regulation, https://eur-lex.europa.eu/legal-content/EN/TXT/?qid=1532348683434&uri=CELEX:02016R0679-20160504.
2. ISO/IEC 27018:2014 Code of practice for protection of personally identifiable information (PII) in public clouds acting as PII processors. International Organization for Standardization (2014).
3. Breitenbücher, U., Binz, T., Kopp, O., Leymann, F., Wettinger, J.: Integrated Cloud Application Provisioning: Interconnecting Service-Centric and Script-Centric Management Technologies. In: On the Move to Meaningful Internet Systems: OTM 2013 Conferences (CoopIS 2013). pp. 130–148. Springer (2013).
4. Oppenheimer, D., Ganapathi, A., Patterson, D.A.: Why do internet services fail, and what can be done about it? In: Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems (USITS 2003). USENIX (2003).

5. Opscode, Inc.: Chef Official Site. (2017).
6. Canonical Group Ltd(GB): Juju Official Site. (2017).
7. Mohaan, M., Raithatha, R.: Learning Ansible. Packt Publishing (2014).
8. Kubernetes, https://kubernetes.io/.
9. Breitenbücher, U., Képes, K., Frank, L., Wurster, M.: Declarative vs. Imperative: How to Model the Automated Deployment of IoT Applications? (2017).
10. Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications. Springer (2014).
11. Fischer, M.P., Breitenbücher, U., Képes, K., Leymann, F.: Towards an Approach for Automatically Checking Compliance Rules in Deployment Models. In: Proceedings of The Eleventh International Conference on Emerging Security Information, Systems and Technologies (SECURWARE). pp. 150–153. Xpert Publishing Services (XPS) (2017).
12. Breitenbücher, U.: Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements, (2016).
13. Saatkamp, K., Breitenbücher, U., Kopp, O., Leymann, F.: Topology Splitting and Matching for Multi-Cloud Deployments. In: Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017). pp. 247–258. SciTePress (2017).
14. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub) graph isomorphism algorithm for matching large graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence. 26, 1367–1372 (2004).
15. Binz, T., Breitenbücher, U., Haupt, F., Kopp, O., Leymann, F., Nowak, A., Wagner, S.: OpenTOSCA - A Runtime for TOSCA-based Cloud Applications. In: Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013). pp. 692–695. Springer (2013).
16. OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0. Organization for the Advancement of Structured Information Standards (OASIS) (2013).
17. OASIS: TOSCA Simple Profile in YAML Version 1.0. Organization for the Advancement of Structured Information Standards (OASIS) (2015).
18. OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0. Organization for the Advancement of Structured Information Standards (OASIS) (2013).
19. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F.: Winery – A Modeling Tool for TOSCA-based Cloud Applications. In: Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013). pp. 700–704. Springer (2013).
20. Kruchten, P.B.: The 4+1 view model of architecture. IEEE software. 12, 42–50 (1995).
21. De Silva, L., Balasubramaniam, D.: Controlling software architecture erosion: A survey. Journal of Systems and Software. 85, 132–151 (2012).
22. Murphy, G.C., Notkin, D., Sullivan, K.J.: Software reflexion models: Bridging the gap between design and implementation. IEEE Transactions on Software Engineering. (2001).
23. Koschke, R., Simon, D.: Hierarchical Reflexion Models. In: WCRE. pp. 186–208 (2003).
24. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on software engineering. 26, 70–93 (2000).

25. Tibermacine, C., Fleurquin, R., Sadou, S.: A family of languages for architecture constraint specification. Journal of Systems and Software. 83, 815–831 (2010).
26. Deiters, C., Dohrmann, P., Herold, S., Rausch, A.: Rule-based architectural compliance checks for enterprise architecture management. In: Enterprise Distributed Object Computing Conference, 2009. EDOC'09. IEEE International. pp. 183–192. IEEE (2009).
27. Fellmann, M., Zasada, A.: State-of-the-art of business process compliance approaches. (2014).
28. Schleicher, D., Grohe, S., Leymann, F., Schneider, P., Schumm, D., Wolf, T.: An approach to combine data-related and control-flow-related compliance rules. In: Service-Oriented Computing and Applications (SOCA), 2011 IEEE International Conference on. pp. 1–8. IEEE (2011).
29. Koetter, F., Kochanowski, M., Weisbecker, A., Fehling, C., Leymann, F.: Integrating compliance requirements across business and it. In: Enterprise Distributed Object Computing Conference (EDOC), 2014 IEEE 18th International. pp. 218–225. IEEE (2014).
30. Tran, H., Zdun, U., Oberortner, E., Mulo, E., Dustdar, S., others: Compliance in service-oriented architectures: A model-driven and view-based approach. Information and Software Technology. 54, 531–552 (2012).
31. Liu, Y., Muller, S., Xu, K.: A static compliance-checking framework for business process models. IBM Systems Journal. 46, 335–361 (2007).
32. OASIS: Web Services Business Process Execution Language (WS-BPEL) Version 2.0. Organization for the Advancement of Structured Information Standards (OASIS) (2007).
33. Xu, K., Liu, Y., Wu, C.: Bpsl modeler–visual notation language for intuitive business property reasoning. Electronic Notes in Theoretical Computer Science. 211, 211–220 (2008).