Institute of Architecture of Application Systems

Pattern-based Deployment Models and Their Automatic Execution

Lukas Harzenetter, Uwe Breitenbücher, Michael Falkenthal, Jasmin Guth, Christoph Krieger, and Frank Leymann

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{harzenetter, breitenbuecher, falkenthal, guth, krieger, leymann}@iaas.uni-stuttgart.de

BibTeX:

```
@inproceedings{Harzenetter2018_PatternbasedDeploymentModels,
  author    = {Lukas Harzenetter and Uwe Breitenb{\"u}cher and
               Michael Falkenthal and Jasmin Guth and Christoph Krieger and
               Frank Leymann},
  title     = {Pattern-based Deployment Models and Their Automatic Execution},
  booktitle = {11\textsuperscript{th} IEEE/ACM International Conference on
               Utility and Cloud Computing UCC 2018, 17–20 December 2018,
               Zurich, Switzerland},
  year      = 2018,
  pages     = {41--52},
  doi       = {10.1109/UCC.2018.00013},
  publisher = {IEEE Computer Society}
}
```


Universität Stuttgart
Germany

Pattern-based Deployment Models and Their Automatic Execution

Lukas Harzenetter, Uwe Breitenbücher, Michael Falkenthal,
Jasmin Guth, Christoph Krieger, and Frank Leymann

Institute of Architecture of Application Systems (IAAS), University of Stuttgart, Stuttgart, Germany
{harzenetter, breitenbuecher, falkenthal, guth, krieger, leymann}@iaas.uni-stuttgart.de

***Abstract*—The automated deployment of cloud applications is of vital importance. Therefore, several deployment automation technologies have been developed that enable automatically deploying applications by processing so-called deployment models, which describe the components and relationships an application consists of. However, the creation of such deployment models requires considerable expertise about the technologies and cloud providers used—especially for the technical realization of conceptual architectural decisions. Moreover, deployment models have to be adapted manually if architectural decisions change or technologies need to be replaced, which is time-consuming, error-prone, and requires even more expertise. In this paper, we tackle this issue. We introduce a meta-model for Pattern-based Deployment Models, which enables using cloud patterns as generic, vendor-, and technology-agnostic modeling elements directly in deployment models. Thus, instead of specifying concrete technologies, providers, and their configurations, our approach enables modeling only the abstract concepts represented by patterns that must be adhered to during the deployment. Moreover, we present how these models can be automatically refined to executable deployment models. To validate the practical feasibility of our approach, we present a prototype based on the TOSCA standard and a case study.**

***Keywords*-Deployment Automation; Deployment Modeling; Patterns; Model-driven Architecture; TOSCA;**

I. Introduction

Cloud computing has evolved to an important paradigm as it offers cost-efficient and on-demand IT infrastructure [1]. However, depending on the complexity of an application, manually deploying a cloud application quickly gets error-prone and requires immense expertise—especially when requirements on security, elasticity, and data consistency must be considered by the operator during the deployment [2], [3]. Therefore, several *deployment automation systems* have been developed that enable the automated deployment of applications, for example, CloudFormation. Most of these deployment systems provision applications by processing so-called *deployment models* [4], [5], which describe all components of the applications to be deployed as well as their relationships. For example, a typical deployment model may specify that a Java Web-application has to be deployed on Amazon's Beanstalk to enable automatic scaling, while a single Ubuntu virtual machine running on Amazon EC2 hosts a MySQL database to which the application connects.

However, already the creation of such simple deployment models quickly becomes a challenging task. For example, to achieve an optimal scaling behavior of the Web-application for handling varying workload, significant experience and technical expertise are required to configure the PaaS: Metrics have to be selected, thresholds need to be specified, etc.

Moreover, there is a lack when it comes to describing such deployments in a generic, vendor- and technology-independent way: Using a concrete technology in the model results in a *lock-in effect* as changing this technology afterwards possibly requires changes on multiple layers. However, the most influential issues are subsequent changes of architectural decisions that must be respected by the deployment. For example, if the previously non-scaling database should be scaled, too, this requires complex configurations on several components or replacements, e. g., by a database service—which, ironically, again requires specific knowledge.

These issues mainly result from the very detailed technical modeling of components, relationships, and configurations required to fully automate deployments. Therefore, in this paper, we present an approach that eliminates the necessity for manually modeling such technical details beforehand. We introduce *Pattern-based Deployment Models*, which enable using conceptual patterns directly in deployment models instead of modeling concrete components, technologies, and configurations. Further, we present algorithms to refine these pattern-based models automatically to executable deployment models. The approach goes beyond the mere replacement of abstractly modeled components by more concrete components as it also includes the semantics of the patterns. By combining both contributions, the approach follows the concepts of Model-driven Architecture (MDA) [6] and avoids vendor- and technology-lock-ins as technologies can be selected for each deployment individually. We validate the practical feasibility of our approach by a prototype based on the TOSCA standard.

The remainder is structured as follows: Section II motivates our approach and introduces the concept of patterns and pattern languages. Section III introduces a method for *Pattern-based Deployment Modeling*, while Sect. IV introduces the meta-model, which can be refined by the algorithms presented in Sect. V. Section VI presents our prototype. Finally, related work and the conclusion are discussed in Sect. VII and VIII.

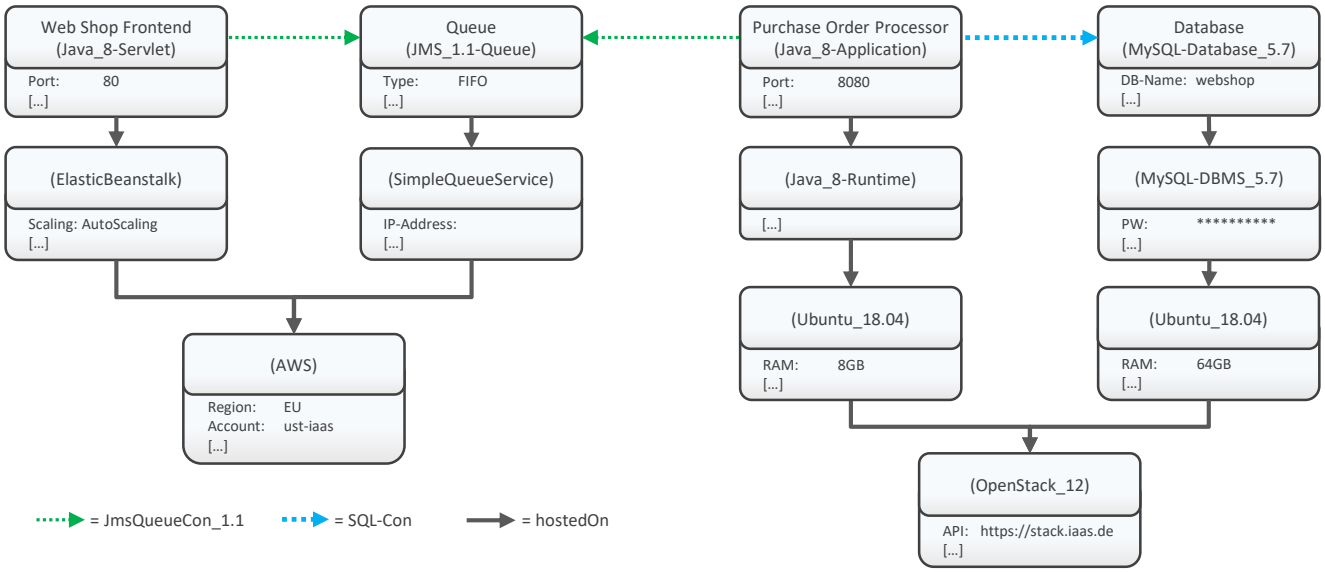

Figure 1. Exemplary declarative deployment model of a simple cloud-based Web shop

II. Motivation Example and Fundamentals

In this section, we present our motivation and describe fundamentals about deployment automation and patterns.

A. Deployment Automation & Deployment Models

The automation of application deployment is essential as manually deploying applications is too error-prone, time-consuming, and requires an immense level of technical expertise [2], [3]. Therefore, various deployment automation systems have been developed that process deployment models in order to automatically deploy applications [4]. Deployment models are either of *declarative*, or *imperative* nature. *Imperative deployment models* explicitly describe processes including all technical activities to be executed to deploy an application, their sequence, and the data flow in between [5]. In contrast, *declarative deployment models* describe only the desired outcome of a deployment in the form of the application's structure including all components and their relations in a directed, weighted, and possibly disconnected graph, which is commonly refereed to as the *topology* of an application [5]. Thereby, they omit technical details on how the deployment should be performed [5]. In this paper, we focus on declarative deployment models as this kind is supported by a variety of deployment systems [4].

B. Running Case Study

In this section, we introduce a case study that is used throughout the paper to motivate and explain our approach. Figure 1 shows a typical declarative deployment model of a company's Web shop application. The shown model is a conceptual representative and not specific to a particular technology, but its topology-based structure allows implementing this model in various declarative technologies [5], e. g., TOSCA [7]. The left side shows the application stack building the shop frontend, which consists of a Java servlet hosted on an instance of Amazon's PaaS offering Elastic Beanstalk in order to enable automatic scaling for handling unpredictable workload. Due to laws concerning private data protection, the data tier needs to be hosted in the private cloud of the company. Therefore, the data tier shown on the right side consists of a relational database (MySQL Database 5.7) running in a virtual machine, which is hosted by the company's own OpenStack environment. As the frontend is running in a public cloud, it should not directly access the database running in the private environment for security reasons. Therefore, a Java processor running in the private part picks all purchase order requests from a queue, which is hosted on Amazon SQS, and processes them. Deployment models specify desired configurations by attributes, e. g., scaling thresholds or application ports. In addition, attributes can also contain instance information such as IP-addresses of the provisioned instances as shown for the Queuing Service.

However, already the creation of this simple deployment model requires significant technical expertise about several concrete technologies, e. g., for configuring Beanstalk appropriately for handling unpredictable workload. These problems arise primarily from the widespread focus on *technology-oriented deployment modeling* forced by available deployment systems, which requires a detailed and concrete specification of the components, relationships, and configurations to be deployed in order to achieve the desired effects—e. g., to achieve an optimal scaling of a component for a certain workload prediction. Therefore, we propose to shift this focus towards *concept-oriented deployment modeling*, i. e., reducing dependencies to concrete technologies by using *patterns* as first-class deployment modeling elements.

C. Architecture Patterns & Pattern Languages

Patterns are an established means to document abstract solutions for frequently recurring problems in a specific domain [8]. They typically follow a well-defined structure, which mainly consists of a *problem* description, a description of the *context* in which it can be applied, a proven *solution*, and an *icon* for graphical modeling.

Typically, patterns are not isolated pieces of advices, instead each pattern is linked to other patterns which might be relevant in the same context. These links between patterns, establish a network of patterns—a so called pattern language [8]. Moreover, those links can have different semantic meanings to clearly indicate different navigation possibilities [9]. Among others, links with AND semantics describe that two related patterns are often used in combination, while OR-links indicate alternative choices, and XOR-links imply exclusive options of the described pattern [10]. Hence, these links can share insights to equivalent problems, context, or solutions of other patterns [11]. Semantic links can also be used to describe that linked patterns deal with equivalent problems but different levels of granularity regarding implementation-specific or technology-specific details [12].

In the domain of software architecture there are several different pattern languages. For example, the Cloud Computing Patterns by Fehling et al. [13], the Enterprise Integration Patterns by Hohpe et al. [14], or the Internet of Things (IoT) Patterns by Reinfurt et al. [15], [16], which describe proven solutions for reoccurring problems for developing software and hardware components in an abstract and reusable manner.

The Cloud Computing Patterns describe problems and best practices in the domain of cloud computing as defined by the NIST definition [17]. Hereby, they describe different solutions for (i) cloud computing fundamentals, (ii) cloud offerings, (iii) cloud application architectures, (iv) cloud application management, and (v) composite cloud applications [13]. The latter ones introduce, e.g., the *Elastic Platform* pattern [13] solving the problem of how execution environments offered by cloud providers should behave and how customers can employ them for their applications. The context of the Elastic Platform pattern is considering a fundamental cloud property: The ability to share resources among different customers which are maintained by the provider. By increasing the amount of shared resources the benefit for both, provider and customer, can be increased as the management effort and consequently costs can be reduced. Therefore, the Elastic Platform pattern as introduced by Fehling et al. [13] describes sharing of middleware components as a solution. The vendor hereby extends the provided offerings and presents managed middleware which is shared by multiple customers and provides simple scaling functionality to its users. Hence, the pattern conceptually describes how to achieve scalability for a component, independently of concrete technologies or providers. Thereby, the Elastic Platform pattern is a vendor and technology agnostic representative of middleware for the execution of applications provided and maintained by a cloud provider which reduces the management and operation costs for its customers.

However, to enable asynchronous and reliable communication between distributed components in a cloud application, the Cloud Computing Patterns have to be combined with the Enterprise Integration Patters by Hohpe et al. [14]. For example, they introduced the Point-to-Point Channel pattern as a specific form of the *Messaging* pattern [14]. While the Messaging pattern generically solves the problem of how multiple applications can communicate with each other responsively in an asynchronous manner, the Point-to-Point Channel pattern describes how an application can perform message-based commands, as an implementation of RPC, or send data to another application or component. It is thereby ensured that a message is received and processed by another application which is able to handle the request. To transfer data in the sense of a Point-to-Point Channel, a typical implementation is a message queue such as Amazon's Simple Queue Service (SQS) or Google's Task Queue.

Moreover, nowadays software components more and more interact with physical hardware in the form of sensors and actuators forming the Internet of Things (IoT). As involved hardware components also influence the overall application architecture, IoT Patterns are used to document best practices in this domain. An example for a hardware device pattern of the IoT Patterns by Reinfurt et al. [15], [16] is the *Energy Harvesting Device* [16]. It solves the problem of powering a hardware device that requires little power by adding a power generation element such as a solar cell to the device. However, IoT devices may still not be available at all times, e.g., due to network outages. To enable other components to communicate with them either way, the *Device Shadow* pattern [15] introduces a server component managing all communication to each device by persistently saving an image of each device.

Thus, patterns provide a suitable means to be used as nodes in deployment models, while behavioral patterns could annotate nodes and relations. Furthermore, by combining multiple pattern languages, for example, the presented Cloud Computing Patterns, Enterprise Integration Patterns, and IoT Patterns, a large set of structural and behavioral patterns can be used for modeling the deployment of an application in a generic and vendor independent way. While structural patterns can be used as nodes to abstractly represent a set of components, behavioral patterns can be used to annotate nodes and relations to detail their characteristics. Thereby, patterns are predestined to be used in deployment models to reduce dependencies to concrete technologies which provides the basis for our approach. In this paper, we focus on structural patterns to provide a first basis for *Pattern-based Deployment Modeling*. In future work, we plan to extend our metamodel to also support suitable behavioral patterns.

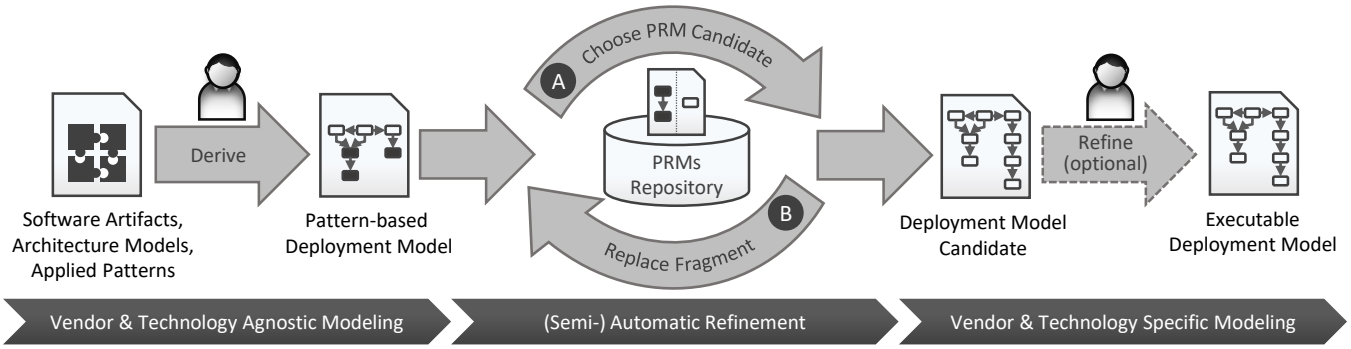

Figure 2. Pattern-based Deployment Modeling Method

III. A Method for Pattern-based Deployment Modeling

In order to model the deployment of an application without introducing dependencies to technologies or providers, we propose a method to describe the deployment in an abstract way. Thereby, patterns are employed as *first-class deployment modeling elements* that allow to relax the coupling of a deployment model to specific components and technologies. The method consists of three phases as illustrated in Fig. 2 from left to right: (i) the *Vendor & Technology Agnostic Modeling* phase, (ii) the *(Semi-) Automatic Refinement* phase, and (iii) the *Vendor & Technology Specific Modeling* phase. Thereby, the approach applies the concepts of Model-driven Architecture to the domain of deployment automation.

A. Vendor & Technology Agnostic Modeling

As depicted in Fig. 2, in the first phase, the modeler analyzes the software artifacts, architecture models, and other entities of the application to be deployed as basis for deriving a deployment model—similarly to the common procedure. However, instead of creating a deployment model that specifies all concrete components, providers, technologies, and configurations, our method also allows the use of *patterns* for describing requirements from a conceptual point of view. Therefore, the method introduces so-called *Pattern-based Deployment Models*, which specify concrete components and configurations similarly to traditional deployment models, but also enable the use of conceptual patterns for specifying *concepts* that must be realized during deployment.

For instance, to relax the coupling of the deployment model to specific technologies, such as a specific version of a MySQL Database Management System, the concrete MySQL Database Management System can be replaced by the *Relational Database* pattern from the Cloud Computing Pattern Language by Fehling et al. [13]. Thus, the semantics of the deployment model is relaxed to cover all technologies that are implementations of the Relational Database pattern. Following this example, the modeler can introduce variability points into the deployment model where coupling to specific technologies and providers has to be avoided.

B. (Semi-) Automatic Refinement

The previous phase results in a Pattern-based Deployment Model, which leverages patterns as first-class modeling elements of application structures to be deployed. However, these models cannot be instantiated directly as they contain patterns which only describe concepts rather than concrete components and technologies. Therefore, all patterns have to be replaced by concrete components and technologies in order to create a deployment model that can be consumed and executed by a deployment automation system. Of course, executing this refinement manually is a complex task, which is time-consuming, error-prone, and requires immense expertise about the used patterns and technologies.

Therefore, we introduce so-called *Pattern Refinement Models (PRM)*, which we introduce in detail in the next sections. PRMs specify a refinement of patterns to concrete components implementing these patterns. Thus, they are used to (i) automatically detect patterns and related composite structures in a Pattern-based Deployment Model in order to (ii) replace them with suitable deployment model fragments specifying concrete components and technologies implementing the detected patterns. The replacement is executed iteratively (see circular steps A and B in Fig. 2). In one iteration, several suitable Pattern Refinement Models could be found for refining the modeled patterns, e. g., several databases may be used as implementations for the Relational Database pattern. Thus, in this case the modeler can select the desired refinement. In the next sections, we also show how this can be automated completely. These iterations are executed until all patterns in the Pattern-based Deployment Model have been replaced by concrete components. The resulting model is called a *Deployment Model Candidate*.

C. Vendor & Technology Specific Modeling

Finally, in the last, optional phase, the deployment modeler may need to additionally refine the Deployment Model Candidate towards an *Executable Deployment Model*. Manual refinements could be necessary to further configure components to enable automatic deployment, e.g., provide user accounts and passwords of the cloud offerings used.

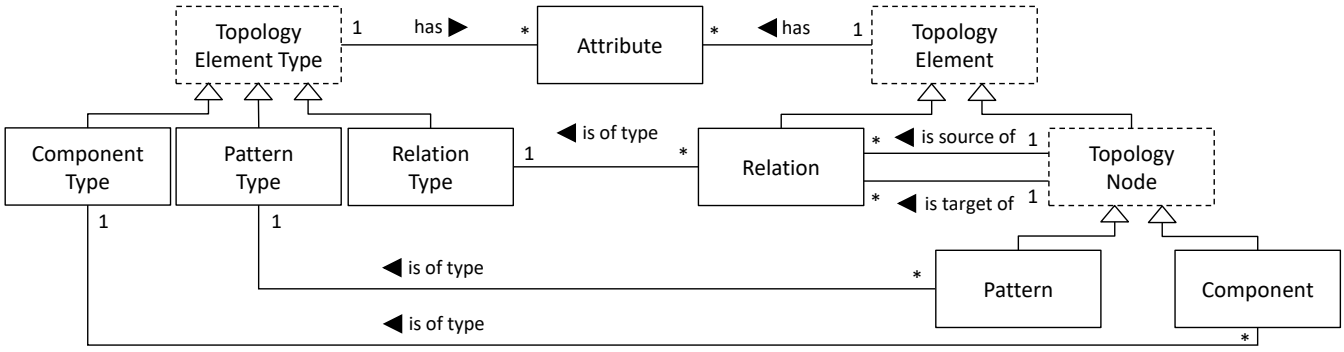

Figure 3. A metamodel for Pattern-based Deployment Models (abstract classes are depicted as rectangles with dashed border)

IV. Pattern-based Deployment Models

This section introduces a formal metamodel for Pattern-based Deployment Models, which are used in the presented method to model deployments on a conceptual level. First, we introduce the metamodel on an abstract level, independently of a concrete declarative deployment technology. Afterwards, in Sect. VI, we (i) show how the TOSCA standard can be extended to support the metamodel and (ii) how Pattern-based Deployment Models can be automatically refined to executable models using the OpenTOSCA ecosystem.

A. Metamodel for Pattern-based Deployment Models

Fig. 3 shows the metamodel as class diagram. The basis of the metamodel is the concept of *application topologies* as defined by TOSCA [7]: An application topology describes the structure of an application including its components, their relationships, and configurations in the form of a weighted directed graph. However, Pattern-based Deployment Models extend this common declarative modeling style [5] by patterns as first-class modeling elements. The advantage of using patterns in deployment models is that modelers only require to specify the *concepts* that must be realized by the deployment system, rather than forcing them to specify the technologies and configurations required to implement these concepts.

Let $\mathcal{T}$ be the set of all Pattern-based Deployment Models, then $t \in \mathcal{T}$ is defined as a nine-tuple as follows:

$$t = (C_t, P_t, R_t, CT_t, PT_t, RT_t, A_t, type_t, supertype_t) \tag{1}$$

The elements of t are defined as follows:

- C_t is the set of *Components* in t, whereby each $c_i \in C_t$ represents a component of the application to be deployed
- P_t is the set of *Patterns* in t. Each $p_i \in P_t$ represents a pattern that must be realized during deployment.
- The union of the set of Components and the set of Patterns denoted as $TN_t := C_t \cup P_t$ consequently represents the set of all *Topology Nodes* in t.
- $R_t \subseteq TN_t \times TN_t$ is the set of *Relations* in t, whereby each $r_i = (tn_s, tn_t) \in R_t$ represents the relationship between two topology nodes, where tn_s is the source node and tn_t is the target node of the relationship.
- CT_t is 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.
- PT_t is the set of *Pattern Types* in t, whereby each $pt_i \in PT_t$ describes the semantics for the Patterns that have this Pattern Type assigned.
- RT_t is the set of *Relation Types* in t, whereby each $rt_i \in RT_t$ describes the semantics for the Relations that have this Relation Type assigned.
- $type_t$ is a map that assigns all Relations, Components, and Patterns in t to their respective Relation Type, Component Type, or Pattern Type. The union set $TE_t := C_t \cup R_t \cup P_t$ contains all *Topology Elements* of t and the union set $TET_t := CT_t \cup RT_t \cup PT_t$ all *Topology Element Types* of t. Then, the map $type_t$ associates each $te_i \in TE_t$ with an $tet_j \in TET_t$ to provide the semantics for each Topology Element:

$$type_t : TE_t \to TET_t \tag{2}$$

- $supertype_t$ is the map that assigns Relation Types, Component Types, and Pattern Types to their respective supertypes. Consequently, the map $supertype_t$ associates a $tet_i \in TET_t$ with a $tet_j \in TET_t$ with $i \neq j$. This means that tet_j is the supertype of tet_i. The mapping $supertype_t$ is defined as:

$$supertype_t : TET_t \to TET_t \tag{3}$$

- $supertypes_t$ is the map that assigns a Topology Element type to all of its supertypes that can be transitively resolved. Thus, the map that associates a Topology Element Type $tet_i \in TET_t$ of t to its respective $supertype_t(tet_i)$ combined with all transitively resolvable supertypes of $supertype_t(tet_i)$ is defined as:

$$supertypes_t : TET_t \to \wp(TET_t) \tag{4}$$

- $A_t \subseteq \Sigma^+ \times \Sigma^+$ is the set of *Attributes* in t, whereby each $a_i = (Key, Value) \in A_t$ describes an attribute of a Topology Element or of a Topology Element Type.

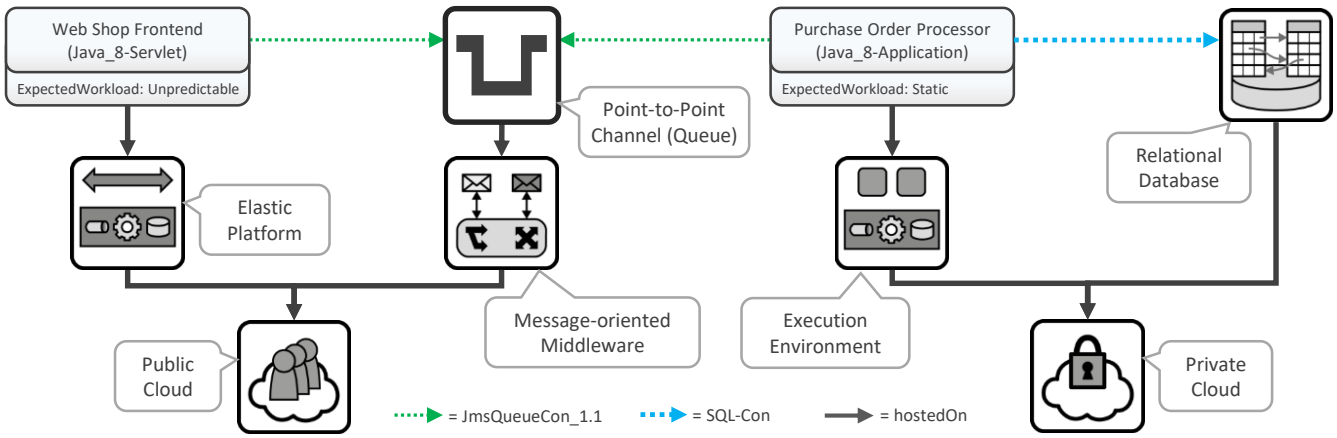

Figure 4. Pattern-based Deployment Model of the case study (speech bubbles attached to the Patterns specify the corresponding Pattern Types)

B. Applying the Approach to the Case Study

This section describes how the technology- and provider-specific deployment model introduced in Sect. II-B can be described as a Pattern-based Deployment Model. The resulting model is shown in Fig. 4 and contains only the shop frontend component and the processor component from the original model as well as their relations. However, infrastructure and middleware requirements are now described as Cloud Computing Patterns [13] and Enterprise Integration Patterns [14], which are depicted as rectangular icons.

The Frontend Component in the upper left corner has the associated Component Type *Java_8-Servlet*. It is connected via a Relation of Relation Type *hostedOn* to a Pattern of Type *Elastic Platform*. Thereby, the Component Java_8-Servlet is defined as the source and the Elastic Platform pattern is defined as the target of the relation. The Elastic Platform is again hosted on a Pattern of type *Public Cloud*. Furthermore, a stack consisting of the two patterns *Point-to-Point Channel* representing the queue concept and a corresponding *Message-oriented Middleware* are hosted on the same *Public Cloud*. On the right-hand side, a Processor Component with Component Type *Java_8-Application* is hosted on a Pattern of type *Execution Environment*. The Processor Component has a *SQL-Con* Relation to a Pattern of type *Relational Database*. The Execution Environment and Relational Database are hosted on a Pattern of Pattern Type *Private Cloud*.

Moreover, the frontend and processor components each specify an attribute *ExpectedWorkload*, which specifies the workload the modelers expect the respective component needs to serve. In this scenario, an unpredictable number of users browse the Web shop's frontend, therefore, it must scale and needs to be hosted on an *Elastic Platform*. However, the company produces and sells only a static number of exclusive products each month. This static number of sellable products thus corresponds to the maximum number of purchase orders the processor has to process, which requires only a non-scaling *Execution Environment* pattern for hosting it.

C. Benefits of Pattern-based Deployment Models

Pattern-based Models significantly reduce the amount of technical expertise modelers need to create it compared to the traditional procedure. For example, while the original model requires a technical configuration of the Amazon Beanstalk environment regarding scaling thresholds, selection of metrics, etc. to serve unpredictable workload, the Pattern-based Model only requires to model an *Elastic Platform* pattern and the specification of the expected workload attribute at the frontend component (cf. Fig. 4). Thus, no technical expertise about the scaling configuration of a certain technology is required, but only the selection of appropriate patterns and conceptual attributes. This makes the deployment model less complex and reduces the risk of incorrect configurations.

Moreover, Pattern-based Models efficiently support changing requirements. For example, if the company increases the number of sellable products, also more purchase orders need to be processed. The increased workload may overload a single instance of the processor, making it necessary to scale it, too. While the adaptation of the original model would require a significant amount of technical expertise about the scaling configuration of OpenStack, the Pattern-based model can be easily adapted: Only the non-scaling *Execution Environment* pattern needs to be replaced by a scaling *Elastic Platform* pattern and the *ExpectedWorkload* attribute of the processor component needs to be changed to *Unpredictable*.

The Pattern-based Model shown in Fig. 4 eliminates technology dependencies of the original deployment model (cf. Sect. II) as all provider and middleware components are replaced by conceptual patterns. Thus, for each deployment of this model, different technologies and providers can be chosen depending on the actual requirements, for example, depending on the desired cloud provider hosting the frontend. As a result, the model can be easily reused for different deployment scenarios without the need to adapt it manually. Moreover, assumptions about technologies and versions of components, such as databases or virtual machines, are avoided.

V. Refinement to Executable Deployment Models

Besides all the benefits described in the last section, Pattern-based Models are not directly executable as the used patterns only describe concepts instead of concrete components and configurations (cf. Sect. III). Therefore, we introduce an approach that enables refining Pattern-based Deployment Models automatically into executable deployment models by replacing the used patterns by appropriate components, by handling adjacent relations, and by setting configurations.

A. Overview of the Refinement Approach

To automatically transform Pattern-based Deployment Models to executable models we introduce *Pattern Refinement Models (PRM)* in this section, which describe how a pattern or a set of interconnected patterns can be refined to concrete components, relations, and technical configurations. A PRM consists of (i) a Pattern-based Deployment Model fragment called *Detector*, (ii) a Pattern-based Deployment Model fragment called *Refinement Structure*, and (iii) a set of *Relation Mappings*. An example is shown in Fig. 5.

The *Detector* of a PRM defines a Pattern-based Deployment Model fragment, which can be refined by the PRM. For example, the Detector can define a single pattern, such as simply the Relational Database pattern, or a more complex fragment, such as the Relational Database pattern hosted on the Private Cloud pattern, as exemplarily depicted in Fig. 5. In order to detect a PRM as possible refinement of a certain part in a Pattern-based Deployment Model, the Detector must be a subgraph of this model, whereby the types and attributes of the Topology Elements defined in the Detector must match the types and attributes of the Topology Elements in the matching part of the model (Subgraph isomorphism). For instance, the Detector modeled in Fig. 5 is a subgraph of the Pattern-based Deployment Model shown in Fig. 4 as the pattern composition specified by the Detector, namely the Relational Database pattern hosted on the Private Cloud pattern, occurs identically in the model. Since the patterns in both models do not define attributes, only their types must match. However, if attributes are specified for Components, Relations, or Patterns in the Detector, the matching elements in the deployment model must provide either (i) exactly the same value or (ii) an arbitrary value in case the detector specifies a wildcard for this attribute (asterisk as value).

The *Refinement Structure* provides a Pattern-based Deployment Model fragment which replaces the subgraph matched by the Detector in the deployment model. For example, the PRM illustrated in Fig. 5 refines the Relational Database pattern hosted on a Private Cloud pattern to a MySQL database hosted on an Ubuntu virtual machine running on OpenStack. This PRM refines both patterns including their relation exclusively to concrete components, relations, and configurations. However, it is also possible to use PRMs to refine high-level patterns to more specific patterns to refine the model stepwise towards actual implementations.

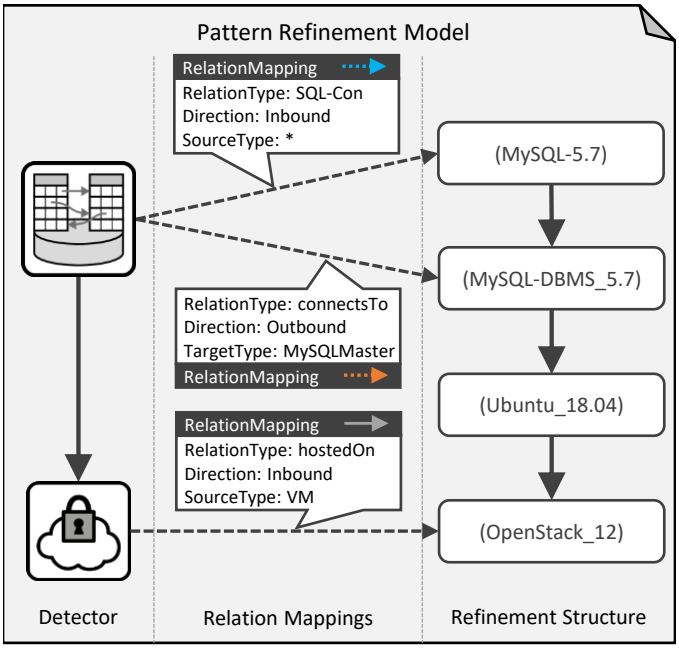

Figure 5. Exemplary Pattern Refinement Model

Lastly, Relation Mappings dictate the rules how ingoing and outgoing relations in a matching subgraph of a Pattern-based Deployment Model can be redirected to the refined fragment represented by the Refinement Structure. For example, the top most Relation Mapping shown in Fig. 5 defines a Relation Mapping for incoming relations of type *SQL-Con* at the Relational Database pattern. Since the Relation Mapping points to the *MySQL-5.7* component in the Refinement Structure, Relations of type SQL-Con targeting the Relational Database pattern are redirected to the MySQL-5.7 Component during refinement. Additionally, depending on the Relation's direction, a Relation Mapping can define a valid source or target type, restricting ingoing or outgoing Relations. For example, the Relation Mapping for Relations of type SQL-Con defines an asterisk as source type, indicating that any type of Topology Node can be the source of the Relation. However, as the other two Relation Mappings in Fig. 5 show, the valid source or target type, can be set to a specific Topology Element Type. If so, the respective source or target Topology Node must be of the defined type, i.e., that only a Topology Node of type *VM*, or which has VM as an element of it's supertypes, can be hosted on the Private Cloud, or ultimately on the OpenStack instance. Similarly, only an outgoing *connectsTo* Relation at the Relational Database pattern, or *MySQL-DBMS_5.7* Component respectfully, can target a Topology Node of type *MySQLMaster*. However, if, for example, a given Pattern-based Deployment Model incorrectly connects Patterns or Components, no, or only a few, PRMs may be applied resulting in an incompletely or incorrectly refined Deployment Model Candidate.

Algorithm 1 refineTopology($t \in T$)

1: **while** $P_t \neq \varnothing$ **do**
2: $candidates := \{(prm_i, sm_j)\} | prm_i \in PRM \wedge sm_j \in SubgraphMappings_{t,\pi_1(prm_i)} \wedge \texttt{applicable}(t, prm_i, sm_j)$
3: **if** $candidates \neq \varnothing$ **then**
4: $(prm_{chosen}, sm_{chosen}) := \texttt{choosePatternRefinementModel}(candidates)$
5: $\texttt{applyRefinement}(prm_{chosen}, t, sm_{chosen})$
6: **else**
7: **return** t
8: **end if**
9: **end while**
10: **return** t

In the following, we introduce a metamodel for PRMs and present algorithms for refining a given Pattern-based Deployment Model by the very same.

B. Metamodel for Pattern Refinement Models

In this section, we define a metamodel for PRMs as basis for the algorithms presented in the next section. As introduced above, a PRM consists of a Detector modeled as Pattern-based Deployment Model fragment, a Refinement Structure which is also a Pattern-based Deployment Model fragment, as well as a set of Relation Mappings.

Let PRM be the set of all Pattern Refinement Models, then $prm \in PRM$ is defined as a three-tuple:

$$prm = (d_{prm}, rs_{prm}, RM_{prm}) \tag{5}$$

The elements of prm are defined as follows:

- $d_{prm} := t_i \in T$: A Pattern-based Deployment Model fragment defining the Detector, which can be replaced by the Refinement Structure fragment.
- $rs_{prm} := t_i \in T$: A Pattern-based Deployment Model fragment defining the Refinement Structure, which replaces the Detector fragment.
- RM_{prm}: The set of Relation Mappings defining redirection rules for ingoing and outgoing relations of Topology Nodes in the Detector to Topology Nodes in the Refinement Structure.

During the refinement of a Pattern-based Deployment Model, the ingoing and outgoing Relations of all Topology Nodes corresponding to a Topology Node in the Detector fragment must be redirected to a Topology Node in the Refinement Structure fragment. The rules dictating which types of a Relation referencing a specific Detector Topology Node must be redirected to which Topology Node in the Refinement Structure, are defined in the Relation Mappings. Let RM_{prm} be the set of Relation Mappings in prm, then $rm \in RM_{prm}$ is defined as a five-tuple:

$$rm = (dn_{rm}, rsn_{rm}, rt_{rm}, direction_{rm}, vt_{rm}) \tag{6}$$

- $dn_{rm} \in TN_{\pi_1(prm)}$: The Topology Node of the prm's Detector whose Relation in a matching Pattern-based Deployment Model must be redirected
- $rsn_{rm} \in TN_{\pi_2(prm)}$: The Topology Node of the prm's Refinement Structure which will be the new source or target of the redirected Relation
- $rt \in RT$: The type of Relations that can be redirected from the matching Topology Node dn_{rm}
- $direction_{rt} \in \{ingoing, outgoing\}$: Specifies which direction of the Relation Type rt can be redirected
- $vt_{rt} \in CT \cup PT$: The valid type or supertype of the considered Relation Type's source or target node

A *Subgraph Mapping* contains all valid subgraph mappings between two given Pattern-based Deployment Models. Let $SubgraphMappings_{t_1,t_2}$ be the set of all subgraph mappings between two Pattern-based Deployment Models $t_1, t_2 \in T$, then $sm_i \in SubgraphMappings_{t_1,t_2}$ is defined as a set of *Element Mappings*. Further, an Element Mapping $em_{te_1,te_2} \in sm_i$ is defined as a tuple of Topology Elements, whereby $te_n \in TE_{t_1}$ and $te_m \in TE_{t_2}$:

$$em_{t_1,t_2} = (te_n, te_m) \tag{7}$$

C. Refinement Algorithms

In the following, we introduce the algorithms to (semi-) automatically refine a Pattern-based Deployment Model into an executable deployment model.

The overall refinement of a Pattern-based Deployment Model is described in Algorithm 1, which gets a Pattern-based Deployment Model $t \in T$ as input. Then, while the Pattern-based Deployment Model t contains patterns (line 1), the algorithm starts by iterating over the following four steps: (i) retrieve all PRM candidates (line 2), (ii) if the set of PRM candidates is not empty (line 3), choose a PRM (line 4), and (iii) apply the chosen PRM to t (line 5). Otherwise, if no more Patterns are contained in t, or no more PRM candidates can be found, (iv) the refined Pattern-based Deployment Model t' is returned (lines 7 and 10).

To check whether a PRM is an applicable candidate, the PRM's Detector must be an isomorphic subgraph of t and the PRM must be applicable to the current Pattern-based Deployment Model t (line 2). If the set of candidates is not empty (line 3), one PRM must be chosen (line 4). This can be implemented in two ways: First, the modeler has to choose

Algorithm 2 applicable($t \in T, prm \in PRM, sm \in SubgraphMappings_{t,\pi_1(prm)}$)

1: **for all** $em_i \in sm : \pi_1(em_i) \in TN_t$ **do**
2: // check if a relation of a Topology Node exists which cannot be handled by a Relation Mapping
3: **if** $\exists r_j \in \texttt{relations}(\pi_1(em_i)) : (\nexists em_y \in sm : \pi_1(em_y) = r_j) \wedge (\nexists rm_x \in \pi_3(prm) :$
$(\pi_1(rm_x) = \pi_2(em_i) \wedge \pi_3(rm_x) \in supertypes(r_j) \wedge \pi_4(rm_x) = \texttt{direction}(r_j) \wedge$
$\pi_5(rm_x) = supertypes(\texttt{sourceOrTarget}(r_j))))$ **then**
4: **return** false
5: **end if**
6: **end for**
7: **return** true

Algorithm 3 applyRefinement($prm \in PRM, t \in T, sm \in SubgraphMappings_{t,\pi_1(prm)}$)

1: $C_t := C_t \cup C_{\pi_2(prm)}; \quad P_t := P_t \cup P_{\pi_2(prm)}; \quad R_t := R_t \cup R_{\pi_2(prm)}$
2: // apply Relation Mappings: redirect external relations
3: **for all** $em_i \in sm : \pi_1(em_i) \in TN_t$ **do**
4: // iterate over all relations of a Topology Node in the detected subgraph ignoring the Relations in this subgraph
5: **for all** $r_j \in \texttt{relations}(\pi_1(em_i)) : (\nexists em_y \in sm : \pi_1(em_y) = r_j)$ **do**
6: // change the source or target of r_j according to the Relation Mapping defined in prm
7: $relationMapping := rm_x \in \pi_3(prm) : (\pi_1(rm_x) = \pi_2(em_i) \wedge \pi_3(rm_x) \in supertypes(r_j) \wedge$
$\pi_4(rm_x) = \texttt{direction}(r_j) \wedge \pi_5(rm_x) = supertypes(\texttt{sourceOrTarget}(r_j)))$
8: **if** $\texttt{direction}(r_j) = ingoing$ **then**
9: $\pi_2(r_j) := \pi_2(rm_x)$ // set the target to the current Topology Node
10: **else if** $\texttt{direction}(r_j) = outgoing$ **then**
11: $\pi_1(r_j) := \pi_2(rm_x)$ // set the source to the current Topology Node
12: **end if**
13: **end for**
14: **end for**
15: // remove all Topology Elements that are mapped by the Detector's fragment
16: $TE_{delete} := \{te_1 \in TE_t : \exists (te_1, te_2) \in sm\}$
17: $C_t := C_t \setminus \{c_i \in TE_{delete} : c_i \in C_t\}; \quad P_t := P_t \setminus \{p_i \in TE_{delete} : p_i \in P_t\}; \quad R_t := R_t \setminus \{r_i \in TE_{delete} : r_i \in R_t\}$
18: **return** t

a PRM (semi-automatic approach) or, second, the PRM is selected in an automated fashion (fully automatic approach). However, regardless of how the refinement candidate is chosen, it gets applied to the Pattern-based Deployment Model t (line 5). Finally, the output of the algorithm is a Deployment Model Candidate t' containing the Refinement Structures of the chosen PRMs.

Algorithm 2 describes how it can be decided if a PRM $prm \in PRM$ can be applied to a given Pattern-based Deployment Model $t \in T$. Additionally, the algorithm requires a subgraph mapping $sm \in SubgraphMappings_{t,\pi_1(prm)}$ between t and the prm's Detector as input. The output is a boolean value that indicates whether the prm is applicable to the Pattern-based Deployment Model t at the given subgraph mapping sm (line 7) or not (line 4). A PRM is applicable to a Pattern-based Deployment Model if all ingoing and outgoing Relations of all matching Topology Nodes can be mapped to a Topology Node of the PRM's Refinement Structure by applying any Relation Mapping contained in the PRM's set of Relation Mappings. Therefore, for each element mapping em_i of two Topology Nodes (line 1), it is checked if an ingoing or outgoing Relation of the Topology Node in t exists that is not contained in the subgraph mapping sm, i.e., that all Relations, which are part of the subgraph, are ignored (line 3). If one of these external relations cannot be redirected using a Relation Mapping defined by the prm (line 3), it is considered inapplicable (line 4). To determine if a Relation Mapping rm redirects a Relation r_j, the following four conditions must hold: (i) the current Topology Node must be the Detector Node in rm, (ii) the type or any of r's supertypes must be equal to the Relation Type defined in rm, (iii) the direction of r must be equal to the direction in rm, and (iv) the type or any supertypes of r's source or target, depending on the direction of r, must be equal to the valid source or target defined in rm.

Lastly, the steps to apply an applicable PRM to a Pattern-based Deployment Model are described in Algorithm 3. The algorithm gets the selected Pattern Refinement Model $prm \in PRM$, a Pattern-based Deployment Model $t \in T$, and the selected subgraph mapping $sm \in$

$SubgraphMappings_{t,\pi_1(prm)}$. Then, in the first step, the Pattern-based Deployment Model defined in the prm's Refinement Structure is imported into t (line 1). Therefore, all Patterns, Components, and Relations defined in the Refinement Structure are added to their equivalents in t. Similar as in Algorithm 2, we again iterate over all element mappings defined in sm which are Topology Node elements (line 3). In the following step, for each of these Topology Nodes, the algorithm iterates over all Relations which are not part of the subgraph mapping, i.e., they are external (line 5). Using the prm's Relation Mappings, the corresponding Relation Mapping dictating the redirect for the current Relation is retrieved (line 7). Then, depending on the direction of the Relation, the source or target is updated to the new source or target element (lines 8-12). Finally, after all Relations were redirected to their new source, or target respectively, the Topology Elements of the subgraph mapping are deleted from t, and t is returned (line 16-18).

D. Discussion and Limitations

The main idea of Pattern-based Deployment Models is to describe several components to be deployed on an abstract level in the form of patterns. For example, Fig. 4 shows the Web Shop Frontend connecting to a queue modeled as pattern. However, of course the frontend's implementation expects a certain messaging technology to connect to. Thus, the refinement algorithm must select PRMs that refine such patterns to appropriate concrete technologies. Our approach ensures this correct refinement by requiring the modeler to define detailed relationships between patterns and concrete components as shown in Fig. 4: The frontend connects to the queue by using a *JmsQueueCon_1.1*, which restricts our algorithms to select PRMs that inject compatible messaging technologies. However, this requires a detailed modeling of all relations between technical components and patterns.

Furthermore, the approach is limited by the applicability of available PRMs. To refine a Pattern-based Deployment Model a possibly huge set of suitable PRMs must be available since parts of the detector may already be refined in a previous refinement step or the allowed Relation Mappings are highly specific. For example, if the Private Cloud pattern illustrated in Fig. 4 was already refined during a previous refinement iteration, the example PRM described in Fig. 5 would not be applicable because the Detector subgraph cannot be found in the Pattern-based Deployment Model. This quickly results in a huge number of different PRM variants describing the same technology configurations using multiple different Detector and Refinement Structure fragments. For example, two similar PRMs may define its Detector as a Relational Database directly hosted on an (i) OpenStack or (ii) Ubuntu instance. In future work, we plan to tackle this issue by generating PRM variants automatically based on comparing different completely refined deployment models and their corresponding original Pattern-based Deployment Models.

VI. Prototype

This section introduces our prototype implementing the presented concepts. It is based on the OASIS standard Topology Orchestration Specification for Cloud Applications (TOSCA) [7], [18], which allows to define the structure and orchestration of cloud applications in a standardized and vendor independent manner. Therefore, we briefly introduce TOSCA and it's mapping to our metamodel.

In TOSCA, cloud applications are described in *Service Templates*. A Service Template describes its deployment model in a directed, weighted and possibly disconnected graph, also referred to as a topology, which is described in the Service Template's *Topology Template*. The Components of a Topology Template are represented by *Node Templates* and Relations by *Relationship Templates* [7]. Similar to our metamodel, both, Node Templates and Relationship Templates are instances of a type specifying their semantics. Hereby, *Node Types* correspond to Component Types, while Relation Types are represented by *Relationship Types*. For example, these types define the attributes for their templates in *Properties Definition* elements, whereby their names and types are specified. Additionally, Patterns and Pattern Types are expressed via annotated Node Templates and Node Types.

As an extension to TOSCA, we introduce the *Pattern Refinement Model* consisting of a *Detector*, a *Refinement Structure*, and a set of *Relation Mappings*. While the Detector and the Refinement Structure reuse the Topology Template definition, the Relation Mappings are new elements and are defined according to their structure described in Sect. V-B. To define a Relation Mapping, users must select a component of the Detector, the corresponding component of the Refinement Structure, and a Relationship Type which can be redirected. However, since Pattern Refinement Models are only used for refining Topology Templates internally, the extension does not interfere with other TOSCA compliant implementations.

The prototype builds upon Winery [19] [20], which is a web-based tool for modeling TOSCA-based applications graphically. To show the feasibility of the presented concepts, the extension of Winery enables the modeling of Pattern-based Deployment Models, as well as Pattern Refinement Models. Moreover, Winery realizes the pattern refinement described in Sect. V-C. It hereby employs a semi-automatic approach in which the user must decide which applicable PRM has to be applied to which subgraph mapping. Therefore applicable PRMs are presented to the user in a list. To ease the selection of the subgraph mapping to be refined, Winery highlights the affected Node Templates when hovering over an applicable PRM. By combining both approaches, Winery eases the development of deployment models and helps to avoid vendor- and technology-lock-ins. The extension is available as open source and demonstrated in a video[1].

[1] https://github.com/OpenTOSCA/winery/releases/tag/paper%2Flh-pattern-based-modeling

VII. Related Work

Our approach for Pattern-based Deployment Models is based on the idea of Platform Independent Models (PIMs) in the context of Model-driven Architecture (MDA) presented by Soley et al. [6]. The main idea of MDA is to iteratively detail an application's architecture with more concrete elements upon each iteration. Therefore, a PIM is created to generically specify the architecture of an application which is then refined into (possibly multiple) Platform Specific Models (PSMs). In our case, the PIM is represented by a Pattern-based Deployment Model since patterns represent abstract and vendor-independent solutions, while an Executable Deployment Model can be referred to as a PSM containing specific technologies. In order to refine the PIM into a PSM, Mellor et al. [21] define steps in their software development process where mappings from an abstract metamodel to the metamodels of the target platforms must be defined and implemented. Pattern Refinement Models combine the mapping and implementation since they can be automatically applied to a Pattern-based Deployment Model. Further, since PRMs can also define Pattern-based Deployment Model fragments in their Refinement Structure, our refinement is able to increase the details on a platform independent level.

Fehling et al. [22] envision a cloud pattern framework, which contains a decision tool and a provisioning tool. The framework allows to enrich patterns with runtime annotations that can be utilized by a so-called *provisioning flow* for deploying software components in cloud environments. Although the framework addresses the provisioning of cloud applications, it does not introduce patterns as first level deployment modeling entities nor does it show how such models can be executed However, while their approach is formulated from the perspective of application development our approach can be grasped as an implementation of their framework from the perspective of deployment models.

Different approaches focus on the transformation of deployment models. Saatkamp et al. [23] formalize architecture and design patterns by means of logic programming to detect and resolve problems in deployment models. Eilam et al. [24], [25] as well as Arnold et al. [26], [27] present approaches to detect structures in application topologies, respectively, deployment models to automatically transform them by predefined transformation steps similar to PRMs. However, these approaches do not introduce patterns as first-class modeling elements in deployment models to enable conceptual, i. e., technology- and provider-independent, modeling.

Hallstrom and Soundarajan [28] introduce design refinement via patterns. Thereby, sub-patterns refine the solution concepts of more abstract patterns. This leads to specialized variants of the abstract solution concepts of coarse-grained patterns. A similar approach that especially addresses to close the abstraction gab between the solution concepts described in patterns and refinements of them towards concrete technologies is presented by Falkenthal et al. [11]. Both approaches can be combined with the presented work to allow the refinement of patterns used in deployment models towards specific implementation variants and technologies.

Di Martino et al. [29] use patterns to describe the composition of cloud services to overall cloud applications. They further present a semantic model of patterns, which can be used to describe business processes, cloud applications and mappings to required cloud resources for their implementation [30]. Those mappings are similar to Pattern Refinement Models, however, they cannot be directly used to describe abstract deployment models for later reuse, nor can they be automatically refined by concrete deployment model fragments as presented in this work.

Falkenthal et al. [31] introduce *concrete solutions* as reusable implementations of patterns [9]. Furthermore, they introduce *aggregation operators* [32] capable of combining concrete solutions of different patterns to be combined. Thus, the introduced PRMs can be seen as concrete solutions that act in combination with other PRMs as aggregation operators.

Schürr [33] introduced *Triple Graph Grammars (TGGs)* which can be used to generically define transformations of graphs from one model to another. In contrast to correspondence graphs in TGGs, Relation Mappings in PRMs do not specify correspondences between nodes directly, but rather how external relations, i.e., ingoing or outgoing relations which are not part of the detected subgraph, can be redirected to the exchanged graph fragment. Thereby, each Relation Mapping defines the Relation Type, the direction, and the source or target node of an external relation. To check the applicability of a PRM to a Pattern-based Deployment Model, its Relation Mappings are taken into account.

VIII. Conclusion and Future Work

The modeling of deployment models requires deep technical knowledge, especially considering conceptual architectural decisions. In this paper, we introduced a method for (semi-) automatically refine Pattern-based Deployment Models into executable ones, to enable their automated deployment. We further presented a formal metamodel and showed the feasibility using a TOSCA-based prototype. Our method leads to a simplified modeling of deployment models, avoiding vendor- and technology-lock-ins. Thus, the approach applies the concepts of Model-driven Architecture [21] to the domain of deployment modeling and deployment automation by describing the deployment of an application in a Platform Independent Model (PIM) which is refined in a Platform Specific Model (PSM) in an (semi-) automatic fashion. One limitation of our approach is the assumption that only correct Pattern-based Deployment Models can be processed. To achieve a correct architecture to form the input of our method the approach by Guth and Leymann [34] can be used. In future work, we will close the gap between such architectures and the Pattern-based Deployment Models.

Acknowledgment

This work was partially funded by the DFG projects *SustainLife* (641730), *ADDCompliance* (636503) and the BMWi project SePiA.Pro (01MD16013F).

References

[1] F. Leymann, "Cloud Computing: The Next Revolution in IT," in *Proceedings of the 52th Photogrammetric Week*. Wichmann Verlag, Sep. 2009, pp. 3–12.

[2] D. Oppenheimer, A. Ganapathi, and D. A. Patterson, "Why do internet services fail, and what can be done about it?" in *USITS 2003*. USENIX, Jun. 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*. Springer, Sep. 2013, pp. 130–148.

[4] A. Bergmayr *et al.*, "A Systematic Review of Cloud Modeling Languages," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 22:1–22:38, 2 2018.

[5] C. Endres *et al.*, "Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications," in *PATTERNS 2017*. Xpert Publishing Services (XPS), Feb. 2017, pp. 22–27.

[6] R. Soley *et al.*, "Model driven architecture," *OMG white paper*, vol. 308, no. 308, p. 5, 2000.

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

[8] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, Aug. 1977.

[9] M. Falkenthal and F. Leymann, "Easing Pattern Application by Means of Solution Languages," in *PATTERNS 2017*. Xpert Publishing Services (XPS), 2017.

[10] R. Reiners, "An Evolving Pattern Library for Collaborative Project Documentation," Ph.D. dissertation, RWTH Aachen University, 2013.

[11] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann, A. Hadjakos, F. Hentschel, and H. Schulze, "Leveraging Pattern Application via Pattern Refinement," in *PURPLSOC 2015*. epubli, Jun. 2015.

[12] M. Falkenthal, J. Barzen, U. Breitenbücher, and F. Leymann, "Solution Languages : Easing Pattern Composition in Different Domains," *International Journal On Advances in Software*, vol. 10, no. 3&4, pp. 263–274, 2017.

[13] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, Jan. 2014.

[14] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004.

[15] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, and A. Riegg, "Internet of things patterns," in *Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPLoP)*. ACM, 2016.

[16] Reinfurt, Lukas and Breitenbücher, Uwe and Falkenthal, Michael and Leymann, Frank and Riegg, Andreas, "Internet of Things Patterns for Devices: Powering, Operating, and Sensing," *International Journal on Advances in Internet Technology, IARIA*, pp. 106–123, 2017.

[17] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," 2011.

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

[19] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, "Winery – A Modeling Tool for TOSCA-based Cloud Applications," in *ICSOC 2013*. Springer, Dec. 2013, pp. 700–704.

[20] Eclipse. (2018) Eclipse winery. [Online]. Available: https://eclipse.github.io/winery/

[21] S. J. Mellor, K. Scott, A. Uhl, and D. Weise, "Model-driven architecture," in *Advances in Object-Oriented Information Systems*. Springer Berlin Heidelberg, 2002, pp. 290–297.

[22] C. Fehling, F. Leymann, R. Retter, D. Schumm, and W. Schupeck, "An Architectural Pattern Language of Cloud-based Applications," in *PLoP 2011*. ACM, Oct. 2011.

[23] K. Saatkamp, U. Breitenbücher, O. Kopp, and F. Leymann, "An Approach to Automatically Detect Problems in Restructured Deployment Models based on Formalizing Architecture and Design Patterns," *Computer Science - Research and Development*, 2018, to appear.

[24] T. Eilam, M. Kalantar, A. Konstantinou, G. Pacifici, J. Pershing, and A. Agrawal, "Managing the configuration complexity of distributed applications in Internet data centers," *Communications Magazine*, vol. 44, no. 3, pp. 166–177, Mar. 2006.

[25] T. Eilam, M. Elder, A. V. Konstantinou, and E. Snible, "Pattern-based Composite Application Deployment," in *IM 2011*. IEEE, May 2011, pp. 217–224.

[26] W. Arnold, T. Eilam, M. Kalantar, A. V. Konstantinou, and A. A. Totok, "Pattern Based SOA Deployment," in *ICSOC 2007*. Springer, Sep. 2007, pp. 1–12.

[27] Arnold, William and Eilam, Tamar and Kalantar, Michael and Konstantinou, Alexander V. and Totok, Alexander A., "Automatic Realization of SOA Deployment Patterns in Distributed Environments," in *ICSOC 2008*. Springer, Dec. 2008, pp. 162–179.

[28] J. O. Hallstrom and N. Soundarajan, "Reusing Patterns through Design Refinement," in *Formal Foundations of Reuse and Domain Engineering*. Springer, 2009, pp. 225–235.

[29] B. Di Martino, G. Cretella, and A. Esposito, "Cloud services composition through cloud patterns," in *Adaptive Resource Management and Scheduling for Cloud Computing*. Springer, 2015, pp. 128–140.

[30] B. Di Martino, A. Esposito, S. Nacchia, and S. A. Maisto, "A semantic model for business process patterns to support cloud deployment," *Computer Science - Research and Development*, vol. 32, no. 3, pp. 257–267, 2017.

[31] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, and F. Leymann, "From Pattern Languages to Solution Implementations," in *PATTERNS 2014*. Xpert Publishing Services, May 2014, pp. 12–21.

[32] M. Falkenthal, J. Barzen, U. Breitenbücher, and F. Leymann, "On the Algebraic Properties of Concrete Solution Aggregation," *Computer Science - Research and Development*, 2018.

[33] A. Schürr, "Specification of graph translators with triple graph grammars," in *Graph-Theoretic Concepts in Computer Science*. Springer Berlin Heidelberg, 1995, pp. 151–163.

[34] J. Guth and F. Leymann, "Towards Pattern-based Rewrite and Refinement of Application Architectures," in *Proceedings of the 12th Advanced Summer School on Service Oriented Computing*. IBM Research Division, 2018.