Institute of Architecture of Application Systems

Pattern-based Deployment Models Revisited: Automated Pattern-driven Deployment Configuration

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

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

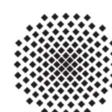
BibTeX:

```
@inproceedings{Harzenetter2020_PatternBasedConfiguration,
    Author    = {Lukas Harzenetter and Uwe Breitenb{\"u}cher and
                 Michael Falkenthal and Jasmin Guth and Frank Leymann},
    Title     = {{Pattern-based Deployment Models Revisited: Automated
                  Pattern-driven Deployment Configuration}},
    Booktitle = {Proceedings of the Twelfth International Conference on
                 Pervasive Patterns and Applications (PATTERNS 2020)},
    Publisher = {Xpert Publishing Services},
    Pages     = {40-49},
    Month     = oct,
    Year      = 2020,
    Isbn      = {978-1-61208-783-2}
}
```

© 2020 Xpert Publishing Services

The published version is available in the Thinkmind Digital Library:

http://www.thinkmind.org/index.php?view=article&articleid=patterns_2020_3_10_70011



Universität Stuttgart
Germany

Pattern-based Deployment Models Revisited: Automated Pattern-driven Deployment Configuration

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

Institute of Architecture of Application Systems (IAAS), University of Stuttgart
Universitätsstrasse 38, 70569 Stuttgart, Germany
Email: {harzenetter, breitenbuecher, falkenthal, guth, leymann}@iaas.uni-stuttgart.de

***Abstract*—The manual deployment of cloud applications is error-prone and requires significant expertise. Therefore, many deployment automation technologies have been developed that enable deploying applications fully automatically by processing deployment models. However, while these technologies substantially simplify deployment, the manual creation of deployment models ironically poses similar challenges to manually deploying applications as technical expertise about the components to be deployed and their dependencies is required. Therefore, we introduced Pattern-based Deployment Models (PbDMs) in a previous work that allow using design patterns to model components in an abstract manner, which are then automatically replaced by concrete technologies. However, in many scenarios, the resulting deployment models still have to be subsequently adapted with regard to the configuration of the selected technologies, e. g., to configure a selected Platform as a Service (PaaS) offering, such as Amazon Beanstalk, for optimal scaling. Therefore, while our previous work only enables using design patterns to model components, in this paper we extend the proposed meta-model and algorithms by the possibility to specify behavioral aspects of components and relations also in the form of patterns. Moreover, we show how these annotated patterns can be automatically transformed into concrete configurations that reflect their semantics. We present a prototype and a case study to validate the extension's practical feasibility.**

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

I. Introduction

Automating the deployment of applications is of vital importance as manual deployment is error-prone, time-consuming, and requires a significant amount of technical expertise [1]. Therefore, several *deployment automation technologies*, such as Chef, Terraform, or Kubernetes, have been developed to automate the deployment of applications. The majority of these technologies use *declarative deployment models* to describe the structure of an application to be deployed [2]. These models specify all components of the application to be deployed, their configurations, as well as their dependencies among each other [3]. For example, to describe the deployment of a Java 8 based application, a declarative deployment model may specify its components as follows: The application itself may be described as an instance of a Java 8 Web App that is hosted on an Amazon Elastic Beanstalk Environment to enable its automatic scaling. Additionally, it may be connected to a MySQL 5.7 database that is installed on an Ubuntu 18.04 Virtual Machine (VM) running on an Amazon EC2 instance.

However, while deployment technologies are an established means, the manual creation of deployment models ironically poses similar challenges to manually deploying applications: First, (i) modelers are required to have *significant technical expertise* in selecting appropriate components, such as web servers or operating systems. For example, considering the example, a modeler has to know which web servers supported by Beanstalk are able to run Java 8 Web Apps. This often results in (ii) *error-prone* modeling that requires testing the created models multiple times, which quickly becomes a (iii) *time-consuming* task. To tackle these issues, we introduced *Pattern-based Deployment Models (PbDM)* in a previous work [4], whereby we used patterns as first-class citizens in a declarative deployment model to describe components in an abstract manner. For example, instead of specifying a concrete web server for Beanstalk to run the Java 8 Web App, in a PbDM only the *Platform as a Service (PaaS)* pattern [5] needs to be modeled. Moreover, since PbDM cannot be executed as they only specify abstract semantics instead of executable technologies, we also presented algorithms to automatically refine all patterns in a PbDM to concrete technologies [4]. However, in many scenarios, the resulting deployment models have to be adapted with regard to the configuration of the refined components, e. g., to configure the scaling behavior of Beanstalk. Unfortunately, this again requires technical expertise and is error-prone. The reason for these problems is that only components are abstracted by patterns, not their configuration.

Therefore, we extend our proposed meta-model for PbDMs in this paper by the possibility to specify also *behavioral requirements* for components and relations in the form of abstract patterns. For example, instead of providing a concrete configuration of Beanstalk's scaling behavior, our new approach enables the annotation of the *Unpredictable Workload Pattern* [5] to the Java Web App, which implies that the underlying infrastructure needs to be elastic, but without the need to specify any technical configuration. Moreover, we also extend the refinement algorithms to support refining patterns annotated at components and relations. We validate the practical feasibility of the approach by a case study and a prototype.

Hereafter, Section II describes fundamentals, Sections III and IV introduce the new concepts while Sections V and VI explain our case study and prototype. Finally, Section VII describes related work and Section VIII concludes the paper.

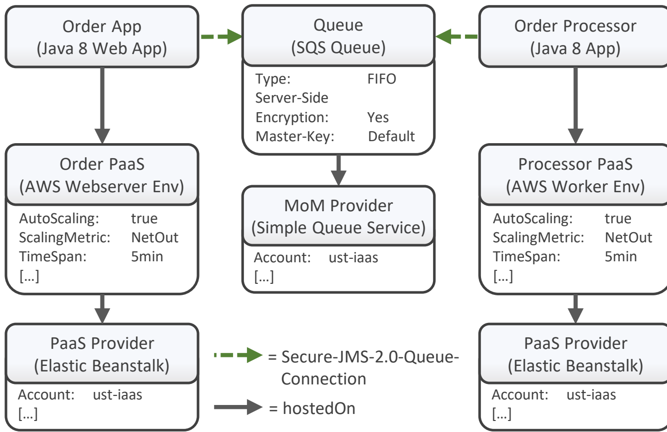


Figure 1. A declarative deployment model.

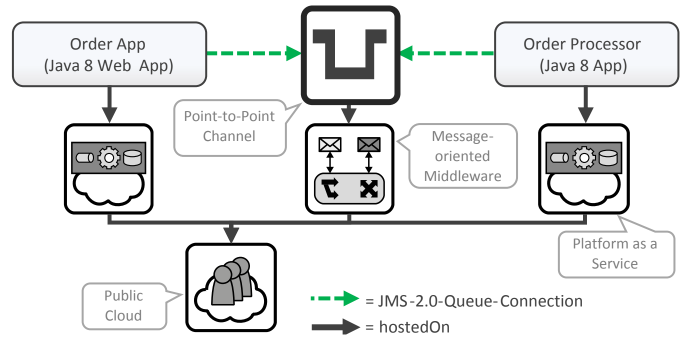


Figure 2. A Pattern-based Deployment Model (PbDM).

II. Fundamentals and Motivation

In this section, we introduce deployment automation concepts and technologies, as well as our motivating scenario.

A. Deployment Models and Automation

To automate the deployment of applications, many deployment automation technologies have been developed. Most of these technologies use *Deployment Models* to describe the desired application [3]. Deployment models can be categorized into two types: (i) *declarative deployment models* and (ii) *imperative deployment models* [3]. An imperative deployment model describes *how* a deployment is performed as an executable process including all technical activities and their execution order [3]. In contrast, a declarative deployment model describes *what* has to be achieved but provides no executable process. Thus, a deployment technology must interpret declarative models and derive the necessary steps [3]. In this paper, we focus on declarative models as they are (i) supported by various deployment technologies [2] and (ii) can be automatically transformed to imperative deployment models [1].

Declarative models state the desired outcome of a deployment in the form of the application's structure encompassing the components of the application, their configurations, and the dependencies between them [3]. An example consisting of components, relations, and their properties is depicted in Figure 1. It illustrates a frontend component called *Order App* and a backend component called *Order Processor*, both hosted on Elastic Beanstalk Environments. The Elastic Beanstalk Environments are configured to scale automatically, which is indicated by their "AutoScaling" properties. To communicate to another, the applications use a *Queue* that is hosted on the Simple Queue Service (SQS), whereas the Order App expects that each order is delivered and processed exactly once, which is hereby ensured by a queue of type "FIFO". Finally, the types of the components and relations are shown. The component types are depicted in braces, while the relation types are encoded by their stroke type and color. Thus, the Order App, e. g., is an instance of the *Java 8 Web App* while its relation to the Queue is of type *Secure-JMS-2.0-Queue-Connection*.

B. Pattern-based Deployment Models

However, even the creation of such simple models poses several challenges as it requires technical expertise in different technologies. For instance, it must be known whether Beanstalk supports Java 8 and which environment is appropriate, or whether an SQS Queue can be used at all since the apps require JMS connections. Thus, to reduce the modeling complexity, we previously introduced *Pattern-based Deployment Models (PbDM)* [4], which use Design Patterns [6] as first class model elements. Figure 2 shows an example PbDM representing the abstract semantics of the deployment shown in Figure 1. Herein, the *Cloud Computing Patterns* [5] and *Enterprise Integration Patterns* [7] are used to represent components in an abstract, technology-agnostic way: Instead of specifying concrete services, such as Beanstalk and SQS, the applications are hosted on *Platform as a Service (PaaS)* patterns [5] while a *Point-to-Point Channel* pattern [7] is used for communication that is hosted on a *Message-oriented Middleware (MoM)* pattern [5]. Thus, this model contains no details about technologies but only specifies the *abstract semantics* of the required components in the form of patterns, which is less error-prone and requires less technical expertise. We refer to patterns that represent the semantics of components as *Component Patterns*. Moreover, we also presented *refinement algorithms* [4] that replace Component Patterns with concrete technologies and providers.

However, in many cases, the refined deployment model requires additional manual configuration: For example, as the Order App's workload is unpredictable, the Beanstalk environment must be configured for automated scaling by specifying the "ScalingMetric" and the "TimeSpan" to define when scaling will be triggered. Additionally, the Order App requires the orders to be processed exactly once by the Order Processor. Thus, the modeler has to select the correct queue type, i. e., in the context of SQS "FIFO" instead of "Standard". Another difficulty often results from compliance requirements: If the orders issued by the Order App contain sensitive data, the communication between the applications must be secured using "Server-Side Encryption" and a "Master-Key". However, to configure all components and relations correctly via properties, immense technical expertise is required on each employed technology and again results in an error-prone and time-consuming model configuration step.

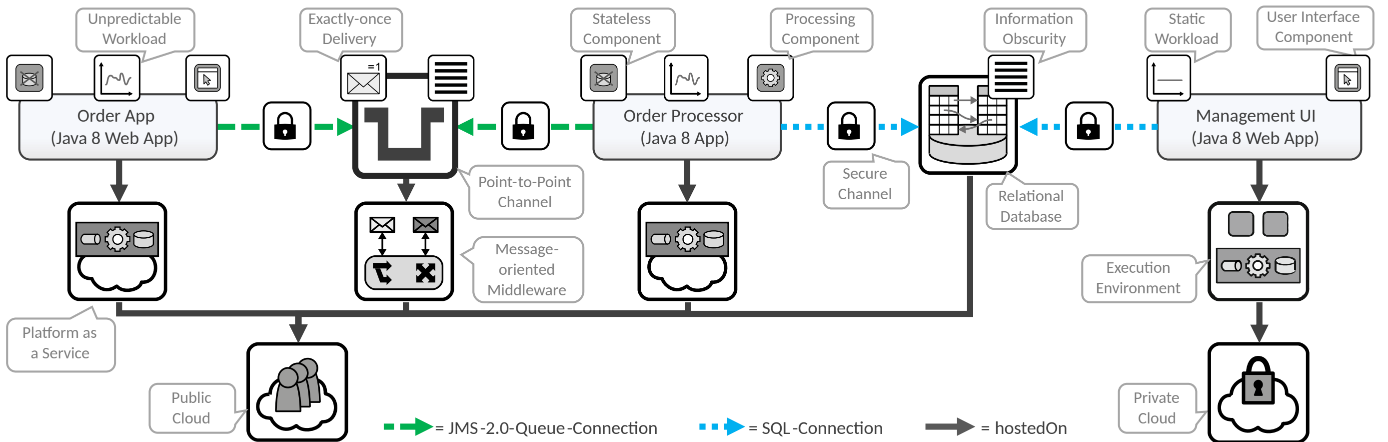


Figure 3. A Pattern-based Deployment and Configuration Model (PbDCM) following the metamodel defined in Figure 4.

III. Pattern-based Deployment and Configuration Models

To tackle the issue of subsequent manual model configuration, our first contribution is an extension of PbDMs to *Pattern-based Deployment and Configuration Models (PbDCMs)*, which support annotating *Behavior Patterns* [8] to components and relations to describe their desired behavior in an abstract way.

To provide the basis for demonstrating our new approach in a more complex case study, we first enlarge our motivating scenario as shown in Figure 3: We add the *Relational Database* Component Pattern [5] to store the results from the Processor as well as a management component called *Management UI* to maintain the database, which is hosted on *Execution Environment* [5] and *Private Cloud* [5] Component Patterns.

A. Overview of the Modeling Extension

To compensate the shortcomings, we extend PbDMs to *Pattern-based Deployment and Configuration Models*: Instead of using patterns only to abstract components by Component Patterns, we extend the metamodel to also allow annotating *Behavior Patterns* to components and relations: A Behavior Pattern abstractly describes behavioral requirements that must be respected by the deployment, e.g., that a component has to handle unpredictable workload. Figure 3 shows a PbDCM in which both apps have been annotated with Behavior Patterns, e.g., the Unpredictable Workload which implies the need for automatic scaling. To secure communication and to encrypt the storage, the relations between the applications are annotated with the *Secure Channel* pattern [9], while the Relational Database and the Point-to-Point Channel are annotated with the *Information Obscurity* pattern [9]. To ensure that the orders are processed only once, the Point-to-Point Channel is annotated with the *Exactly-once Delivery* pattern [5]. Moreover, patterns may specify additional semantics, e.g., the *Stateless Component*, the *User Interface Component* and the *Processing Component* patterns [5]. Thus, instead of specifying all technical configurations that realize these behaviors manually, our extension only requires to annotate desired behavior of components and relations in the form of Behavior Patterns.

B. Metamodel Extensions for PbDCMs

In this section, we describe the formal metamodel for PbDCMs, which is graphically illustrated in Figure 4. Hereby, the original PbDM metamodel [4] is extended by the grey elements, which provide the capabilities to define Behavior Pattern types and to annotate them to components and relations.

The new PbDCM metamodel and the original metamodel are based on the *Essential Deployment Metamodel (EDMM)* [2], which is a normalized metamodel that has been extracted from the 13 most used deployment technologies including, e. g., Terraform and the Topology Orchestration Specification for Cloud Applications (TOSCA) [10]. We use EDMM as basis metamodel to describe our approach in a technology-agnostic way instead of extending only one certain deployment technology. Since Section IV describes how PbDCMs can be automatically transformed into EDMM-compliant models containing only standard EDMM modeling constructs, the extension of EDMM only affects the design time while the refined models can be directly translated into any of the 13 supported deployment technologies that can be mapped to EDMM. To demonstrate the approach's practical feasibility, we show how the PbDCM metamodel can be realized using the TOSCA standard and how the models refined by our algorithms presented in Section IV can be consumed by a standard-compliant TOSCA runtime. Let $\mathcal{T}$ be the set of all PbDCMs, then $t \in \mathcal{T}$ is defined as a fifteen-tuple as follows:

$$t = (C_t, R_t, CP_t, CBP_t, RBP_t, CT_t, RT_t, CPT_t, CBPT_t, RBPT_t, PROP_t, type_t, supertype_t, properties_t, annotations_t)$$

1) Basis of the Metamodel: Following EDMM, a deployment model is a directed, weighted graph, in which nodes represent components, edges their relations. Components and relations are typed and specify properties to configure the deployment. EDMM defines more elements, such as Operations, which are, however, not affected by our approach. Thus, we omit them for the sake of simplicity. Our previous work of PbDMs extends this metamodel by *Component Patterns* that can be used as nodes [4]. Thus, the following elements of t are already defined by EDMM [2] and the metamodel of PbDMs [4]:

Figure 4. Metamodel of Pattern-based Deployment and Configuration Models (extensions to PbDMs are highlighted in grey).

- C_t is the set of *Components* in t. A *Component* $c_i \in C_t$ is a physical, functional, or logical unit of an application.
- CP_t is the set of *Component Patterns* in t. A $cp_i \in CP_t$ describes a *Component Pattern* that must be refined to a concrete *Component* before the application's deployment.
- The union of *Components* C_t and *Component Patterns* CP_t form the set of *Model Nodes* in t: $MN_t := C_t \cup CP_t$.
- $R_t \subseteq MN_t \times MN_t$ represents the set of *Relations* in t. A *Relation* $r_i = (mn_{source}, mn_{target}) \in R_t$ is a directed physical, functional, or logical dependency between exactly two *Model Nodes* $mn_{source}, mn_{target} \in MN_t$, where mn_{source} is the source and mn_{target} the target *Model Node* of the *Relation*.
- CT_t is the set of *Component Types* in t. A *Component Type* $ct_i \in CT_t$ specifies the semantics of a *Component* $c_j \in C_t$ that has this type assigned.
- CPT_t is the set of *Component Pattern Types* in t. A *Component Pattern Type* $cpt_i \in CPT_t$ specifies the semantics of a *Component Pattern* $cp_j \in CP_t$ that has this type assigned.
- RT_t is the set of *Relation Types* in t. A *Relation Type* $rt_i \in RT_t$ specifies the semantics of a *Relation* $r_j \in R_t$ that has this type assigned.
- $PROP_t \subseteq \Sigma^+ \times \Sigma^+$ is the set of *Properties* in t. A *Property* $pr_i = (Key, Value) \in PROP_t$ describes the configuration of a *Component*, *Relation*, *Pattern*, or their types. Its initial value is defined to be the *Empty Word* ε.

2) Extension for Behavior Patterns:

- CBP_t is the set of *Component Behavior Patterns* in t. A $cbp_i \in CBP_t$ represents a pattern annotated to a *Model Node* describing its desired behavior in an abstract way.
- RBP_t is the set of *Relation Behavior Patterns* in t. A $rbp_i \in RBP_t$ represents a pattern that is annotated to a *Relation* describing its behavior in an abstract way.
- $CBPT_t$ is the set of *Component Behavior Pattern Types* in t. A $cbpt_i \in CBPT_t$ specifies the semantics of a *Component Behavior Pattern* $cbp_j \in CBP_t$ that has this type assigend.
- $RBPT_t$ is the set of *Relation Behavior Pattern Types* in t. A $rbpt_i \in RBPT_t$ specifies the semantics of a *Relation Behavior Pattern* $rbp_j \in RBP_t$ that has this type assigned.
- The union set $SE_t := R_t \cup MN_t$ contains all *Structure Elements* in t, while the union set $BP_t := CBP_t \cup RBP_t$ contains all *Behavior Patterns* in t.
- The union of all *Structure Elements* SE_t and *Behavior Patterns* BP_t form the set of *Model Elements* ME_t in t: $ME_t := SE_t \cup BP_t$.
- The union set $MET_t := CT_t \cup RT_t \cup CPT_t \cup CBPT_t \cup RBPT_t$ contains all *Model Element Types* in t.
- $annotations_t$ is the map that assigns a *Structure Element* $se_i \in SE_t$ to its set of annotated *Behavior Patterns*.

$$annotations_t : SE_t \rightarrow \wp(BP_t) \tag{1}$$

The following maps are already defined by EDMM [2] and PbDMs [4]. Since we extended ME_t and MET_t, their mapping now include also *Behavior Patterns* and *Behavior Pattern Types*:

- $type_t$ is a map that assigns all *Model Elements* $me_i \in ME_t$ to their respective *Model Element Type* $met_j \in MET_t$ providing the semantics for the *Model Element*:

$$type_t : ME_t \rightarrow MET_t \tag{2}$$

- $supertype_t$ is the map that assigns each *Model Element Type* to its respective supertype. It associates a $met_i \in MET_t$ with a $met_j \in MET_t$ where $i \neq j$, i. e., that met_j is the supertype of met_i.

$$supertype_t : MET_t \rightarrow MET_t \tag{3}$$

- Additionally, $supertypes_t$ assigns a *Model Element Type* $met_i \in MET_t$ to all of its supertypes that can be transitively resolved. Thus, $supertypes_t$ is defined as:

$$supertypes_t : MET_t \rightarrow \wp(MET_t) \tag{4}$$

- $properties_t$ is the map that assigns each $met_i \in MET_t$ and $me_j \in ME_t$ its corresponding set of *Properties*.

$$properties_t : MET_t \cup ME_t \rightarrow \wp(PROP_t) \tag{5}$$

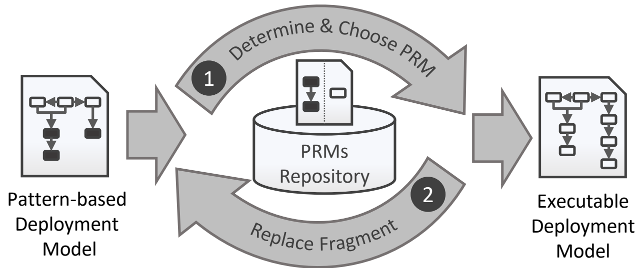


Figure 5. Refinement of PbDMs to executable models [4].

IV. Automatic Refinement to Executable Deployment Models

PbDCMs are not executable as the contained patterns only specify abstract semantics. Thus, to get an *Executable Deployment Model*, all *Component Patterns* need to be replaced by concrete *Components* and the additional semantics specified by the annotated *Behavior Patterns* must be considered by configuring the affected *Components* and *Relations* correctly. In our previous work [4], we presented *Pattern Refinement Models (PRMs)* and corresponding *refinement algorithms* to replace *Component Patterns* by concrete *Components*.

A. Pattern Refinement Models (PRMs)

To deploy a PbDM, we introduced algorithms to automatically replace *Component Patterns* by concrete technologies [4]. Hereby, *Pattern Refinement Models (PRMs)* define how *Component Patterns* can be refined to concrete components [4]. As illustrated in Figure 5, the refinement is an semi-automated, iterative process: All PRMs contained in a repository are analyzed whether they can refine certain *Component Patterns* contained in the PbDM to concrete *Components*. Appropriate PRMs are selected manually and automatically applied until the PbDM contains no more patterns resulting in an *Executable Deployment Model*, which requires only small manual additions.

A PRM consists of (i) a *Detector*, (ii) a *Refinement Structure*, and (iii) a set of *Relation Mappings*. The Detector is a PbDM fragment that specifies the structure of *Component Patterns* and their *Relations* the PRM can refine to concrete *Components*. Thus, if a fragment of a Detector matches a fragment in a PbDM, this PRM can refine exactly the matching subgraph. The Refinement Structure specifies how the Detector fragment can be refined to concrete *Components* and *Relations*. Hence, if a fragment in a PbDM matches a Detector fragment of a PRM, the PbDM fragment can be refined to the fragment specified in the PRM's Refinement Structure. For example, Figure 6 shows a PRM that refines the PaaS and the Public Cloud patterns to a concrete Webserver Environment hosted on AWS Beanstalk.

Moreover, to handle external relations of the mapped Detector fragment, we introduced *Relation Mappings* [4] defining which type of relations can be redirected from which *Model Node* in the Detector to which *Model Node* in the Refinement Structure. For example, the Relation Mapping in Figure 6 redirects all incoming *Relations* of type *hostedOn* that target the Public Cloud *Component Pattern* and that are not contained in the Detector to the Public Cloud *Component*.

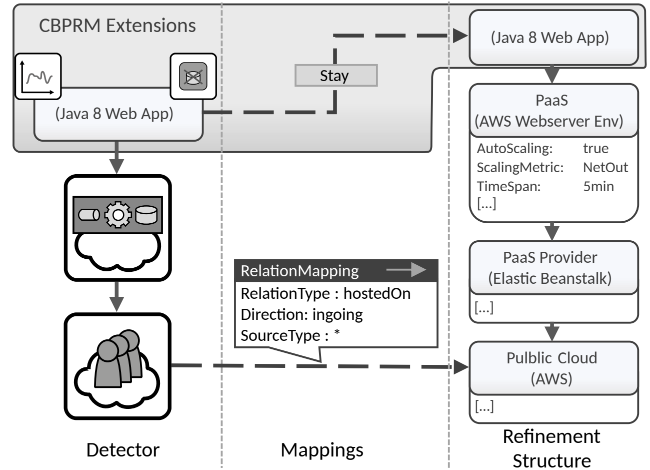


Figure 6. Exemplary CBPRM respecting Behavior Patterns.

B. Component and Behavior Pattern Refinement Models (CBPRMs)

In the original approach [4], PRMs were only used to refine *Component Patterns* by concrete *Components*. Therefore, Detector fragments of PRMs contained only *Component Patterns* and their *Relations*, but no business components as they were not affected by the refinement. Thus, only *Component Patterns* are considered if a PRM is applicable or not. However, our extended approach must consider *Behavior Patterns* that are attached to business *Components* or *Relations*, such as the Java 8 Web App shown in Figure 6. Hence, we extend PRMs to *Component and Behavior Pattern Refinement Models (CBPRMs)* to also support the refinement of *Behavior Patterns*.

The extension requires two changes: First, PRMs must be extended to use PbDCM fragments as Detector and Refinement Structure instead of PbDM fragments. Second, to consider *Behavior Patterns* during the refinement, also the affected business components must be modeled in the Detector to define which *Behavior Patterns* a CBPRM considers. This is, for example, shown in Figure 6: Herein, also the business *Component* Java 8 Web App is modeled in the CBPRM Detector including the two *Component Behavior Patterns* Unpredictable Workload and Stateless Component. This business *Component* is hosted on Platform as a Service and Public Cloud *Component Patterns*. Thus, this Detector specifies that the PaaS and Public Cloud *Component Patterns* can be refined by this CBPRM in a way that it respects the *Behavior Patterns* annotated at the Java 8 Web App, i.e. that its Refinement Structure is able to handle unpredictable workloads of stateless Java 8 Web Apps. Hence, this Detector is refined to an elastic PaaS-based solution on AWS including all required configuration properties, e. g., it specifies the "AutoScaling", "ScalingMetric", and "TimeSpan" *Properties* of Beanstalk to dynamically scale the application. Thus, when applying such a CBPRM, all annotated *Behavior Patterns* in the Detector must be considered by the Refinement Structure. Thereby, the Refinement Structure must not contain any of the Detector's or new patterns. Thus, each CBPRM removes the patterns it refines from the PbDCM model.

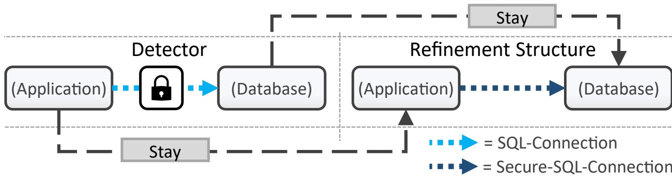


Figure 7. Exemplary CBPRM that refines Relations.

However, while business *Components* and their configuration *Properties* must not be changed during the refinement of *Component Patterns*, their annotated *Behavior Patterns* must be considered. Therefore, we introduce *Stay Mappings* as a second extension to PRMs, which state that a *Model Node* in a PbDCM mapping to a "staying" *Model Node* in the CBPRM's Detector must not be changed. For example, Figure 6 specifies that if the Detector can be mapped to a subgraph in the PbDCM, the *Model Node* of the PbDCM mapping to the Java 8 Web App of the Detector must stay as is, i.e., neither its type nor its configuration must change. Thus, business *Components* are essential to specify the pattern-component constellations a CBPRM can refine. For example, the CBPRM shown in Figure 6 states that it is able to refine Java 8 Web Apps hosted on PaaS and Public Cloud *Component Patterns* while considering the annotated Unpredictable Workload and Stateless Component *Component Behavior Patterns*. To specify where business *Components* will be located in the PbDCM after their annotated *Component Behavior Patterns* are refined, Stay Mappings are defined in CBPRMs. Hence, Stay Mappings are only required if *Behavior Patterns* are refined by a CBPRM.

Moreover, Stay Mappings enable the definition of CBPRMs that only refine a *Relation* between two *Model Nodes* that is annotated with a *Relation Behavior Pattern*. For example, as illustrated in Figure 7, it is possible to refine a *SQL-Connection* annotated with the *Secure Channel* pattern [9] between an application and a database to a *Secure-SQL-Connection* without changing neither the application nor the database. Thus, the application and the database stay in the given PbDCM while their *Relation* gets refined to a more concrete *Relation Type*.

C. Metamodel for CBPRMs

In the following, the metamodel for *Component and Behavior Pattern Refinement Models* is defined based on PRMs [4]. Let $CBPRM$ be the set of all Component and Behavior Pattern Refinement Models, then a $cbprm \in CBPRM$ is a four-tuple:

$$cbprm = (d_{cbprm}, rs_{cbprm}, RM_{cbprm}, S_{cbprm}) \qquad (6)$$

The original metamodel of PRMs [4] is adopted for CBPRMs by exchanging PbDMs by PbDCMs as follows:

- $d_{cbprm} \in \mathcal{T}$ is a PbDCM fragment describing the *Detector* which can be refined by this CBPRM.
- $rs_{cbprm} \in \mathcal{T}$ is a PbDCM fragment that describes the *Refinement Structure* that refines the Detector fragment.
- RM_{cbprm} is the set of *Relation Mappings* describing the rules how external relations of *Model Nodes* in the Detector must be redirected to *Model Nodes* in the Refinement Structure. A $rm_i \in RM_{cbprm}$ is defined as:

$$rm_i = (mn_1, mn_2, rt, direction_{rt}, vt) \qquad (7)$$

Herein, $mn_1 \in MN_{d_{cbprm}}$ and $mn_2 \in MN_{rs_{cbprm}}$ are *Model Nodes* of the Detector d_{cbprm} and the Refinement Structure rs_{cbprm}. $rt \in RT$ is the *Relation Type* of an external *Relation* that targets or sources the *Model Node* matching mn_1, while the *Relation*'s direction is defined as $direction_{rt} \in \{ingoing, outgoing\}$. $vt \in CT \cup CPT$ specifies the valid type of the *Relation*'s source *Model Node*, if it is ingoing, or target *Model Node* otherwise. To also refine *Behavior Patterns*, we introduce *Stay Mappings* in CBPRMs:

- S_{cbprm} is the set of Stay Mappings. A Stay Mapping $s_i = (mn_1, mn_2) \in S_{cbprm}$ is a pair of *Model Modes*, whereby $mn_1 \in MN_{d_{cbprm}}$ is matching a *Model Node* in a PbDCM that must stay as is at the place of the Refinement Structure's *Model Node* $mn_2 \in MN_{rs_{cbprm}}$.

D. Refinement Step 1: CBPRM Selection

Following our original approach [4], to refine *Component Patterns* in a PbDM, first all applicable PRMs are determined. A PRM is applicable iff (i) its Detector fragment can be found as a subgraph of *compatible Structure Elements* in the PbDM and (ii) all external *Relations* of all mapped *Model Nodes* in the PbDM can be redirected by the CBPRM's Relation Mappings [4]. To find two compatible *Structure Elements* their types and annotated *Behavior Patterns* must be considered. Thus, a *Structure Element* in a CBPRM's Detector is only compatible to a *Structure Element* in a PbDCM iff (i) their types are compatible and (ii) all *Behavior Patterns* the CBPRM defines in its Detector are also annotated at a matching *Structure Element* in the PbDCM. To determine the compatibility of two *Structure Elements* in our algorithms, we introduce a formal *Compatibility Rule* similarly to Breitenbücher [11]: A *Structure Element* $se_1 \in SE_{d_{cbprm}}$ of a Detector d_{cbprm} is matching a *Structure Element* $se_2 \in SE_t$ in a PbDCM t iff (i) the type of se_2 or one of its *supertypes* is equal to the type of se_1, (ii) all *annotations* defined at se_1 are also annotated at se_2, (iii) all *Properties* that are set in se_1 are equally set in se_2 or se_1 specifies wildcard values "$*$", which means that any non-empty value is allowed for a *Property*. Based on this, we define the *Compatibility Operator* "$\vec{\approx}$" as follows:

$$\begin{aligned} se_1 \vec{\approx} se_2 :\Leftrightarrow \Big(& type_{d_{cbprm}}(se_1) \in supertypes_t(se_2) \\ & \wedge \big(\forall b_x \in annotations_{d_{cbprm}}(se_1) \exists b_y \in annotations_t(se_2) \\ & \quad \big(type_{d_{cbprm}}(b_x) = type_t(b_y)\big)\big) \\ & \wedge \big(\forall p_i \in properties_{d_{cbprm}}(se_1) \\ & \quad \exists p_j \in properties_t(se_2)\ \big(\pi_1(p_i) = \pi_1(p_j) \wedge \\ & \quad \big(\pi_2(p_i) = \pi_2(p_j) \vee (\pi_2(p_i) = \text{"}*\text{"} \wedge \pi_2(p_j) \neq \varepsilon)\big)\big)\big)\Big) \end{aligned}$$

The set of $SubgraphMappings_{d_{cbprm},t}$ contains all possible subgraph mappings that exist between a CBPRM's Detector d_{cbprm} and a PbDCM t. A Subgraph Mapping $sm_i \in SubgraphMappings_{d_{cbprm},t}$ is defined as the set of *Element Mappings* between *Structure Elements*: An *Element Mapping* $em_j = (se_1, se_2) \in sm_i$ is defined as a tuple of *Structure Elements*, where the *Structure Elements* $se_1 \in SE_{d_{cbprm}}$ and $se_2 \in SE_t$ are compatible, i. e., $se_1 \vec{\approx} se_2$ holds.

E. Refinement Step 2: CBPRM Application

The refinement of an applicable CBPRM $cbprm$ that has been selected to refine a matching subgraph in a PbDCM t is described in Figure 8. Thus, to apply the $cbprm$ to t, they are both passed alongside the Subgraph Mapping sm containing the Element Mappings between the $cbprm$'s Detector and t. Hereby, Lines 1, 2, 11–20, and 23 are used from the original algorithm [4] and are adapted to support *Behavior Patterns* and *Stay Mappings*: First, all *Structure Elements* defined in the Refinement Structure are added to t (Line 1), then all affected *Relations* must be redirected to their new source or target (Lines 3–20). Therefore, all *Relations* that are in- or outgoing of a *Model Node* in t that is part of the subgraph sm (Line 2) must be investigated to redirect (i) *Relations* between added and staying *MNs* (Lines 4–10), and (ii) external *Relations* according to the Relation Mappings defined in the $cbprm$ (Lines 12–19).

To redirect the *Relations* that are in- or outgoing of staying *Model Nodes*, the *Relations* added by the $cbprm$'s Refinement Structure must be considered as the type of the *Relation* can change. Therefore, all *Relations* that are in- or outgoing of a *Model Node* that has been added from the Refinement Structure and that is part of a Stay Mapping (Line 4) must be redirected to the existing *Model Node* in t. For example, by adding the Refinement Structure defined in the CBPRM illustrated in Figure 6 to the PbDCM shown in Figure 3, the Java 8 Web App, the PaaS environment, the PaaS Provider, and the Public Cloud *Components*, as well as all three *Relations* are added. However, as there is a *Component* in the PbDCM that maps to the Java 8 Web App, the added application only serves as a placeholder, where the actual application must be located. Hence, the *Relation* between the placeholder and the PaaS environment must be redirected to the actual Java 8 Web App *Component* in the PbDCM. Thus, all *Relations* in t that have been added from $cbprm$'s Refinement Structure and are either the source or the target of a *Model Node* that is part of a Stay Mapping must be redirected to the corresponding staying *Model Node* that already exists in t. Hence, if the staying *Model Node* was the source, the *Relation's* source must be changed, or its target otherwise (Lines 5 to 9).

Similarly, external *Relations* that are in- and outgoing from the mapped subgraph in t must be redirected to the new *Model Nodes* that have been added from the $cbprm$. For example, based on the Relation Mapping defined for ingoing *Relations* of type *hostedOn* at the Public Cloud in Figure 6, all of these *Relations* must be redirected to the new Public Cloud *Component* of type AWS. Therefore, all *Relations* in t that are the source or the target of the currently processed *Model Node*, and that are not part of the subgraph (Line 12) must be redirected to the added *Model Node* as dictated by the $cbprm$'s Relation Mappings. Thus, for each *Relation* r_j that is in- or outgoing of the current *Model Node* mn_2 in t the following conditions must hold: (i) the *Relation Type* defined in the Relation Mapping must be in the supertypes of r_j, (ii) the direction defined in the Relation Mapping must be equal to the direction of r_j, and (iii) the corresponding source or target *Model Node*, depending on the direction, must be of the same type as defined in the Relation Mapping (Line 13). Then, the *Relation* r_j is redirected to its new source or target *Model Node* which has been added from the $cbprm$ (Lines 14 to 19). For example, if the CBPRM shown in Figure 6 is applied to the Order App in Figure 3, all *Relations* of type *hostedOn* that are ingoing at the Public Cloud *Component Pattern* are redirected to the Public Cloud *Component* of type AWS. Hence, the Message-oriented Middleware, the Platform as a Service the Order Processor is hosted on, and the Relational Database are hosted on AWS after the CBPRM has been applied.

Finally, all *Model Elements* that are part of the subgraph must be deleted as they have been refined to concrete technologies (Lines 22 to 23). This also includes all *Behavior Patterns* annotated at any mapped *Structure Elements*. However, all staying *Model Nodes* mapped by the $cbprm$'s Detector must not be deleted, while all placeholder *Model Nodes* added from $cbprm$'s Refinement Structure must be removed from t.

1: $SE_t := SE_t \cup SE_{rs_{cbprm}}$
2: **for all** $\big((mn_1, mn_2) \in sm : mn_2 \in MN_t\big)$ **do**
3:   *// Redirect added Relations of the RS to staying MNs*
4:   **for all** $\big(r_y \in R_t : r_y \in R_{rs_{cbprm}} \wedge \exists mn_i \in MN_{rs_{cbprm}} \big((mn_1, mn_i) \in S_{cbprm} \wedge (mn_i = \pi_1(r_y) \vee mn_i = \pi_2(r_y))\big)\big)$ **do**
5:     **if** $\big(mn_i = \pi_1(r_y)\big)$ **then**
6:       $\pi_1(r_y) := mn_2$ *// update the source of* r_y
7:     **else**
8:       $\pi_2(r_y) := mn_2$ *// update the target of* r_y
9:     **end if**
10:   **end for**
11:   *// Apply Relation Mappings: redirect external Relations*
12:   **for all** $\big(r_j \in R_t : (mn_2 = \pi_1(r_j) \vee mn_2 = \pi_2(r_j)) \wedge \nexists r_z (r_z, r_j) \in sm\big)$ **do**
13:     $relationMapping := rm_x \in RM_{cbprm} : \big(\pi_1(rm_x) = mn_1 \wedge \pi_3(rm_x) \in supertypes_t(r_j) \wedge \pi_4(rm_x) = \text{DIRECTION}(r_j) \wedge \pi_5(rm_x) \in supertypes_t(\texttt{sourceTarget}(r_j, mn_2))\big)$
14:     **if** $\big(\text{DIRECTION}(r_j) = outgoing\big)$ **then**
15:       $\pi_1(r_j) := \pi_2(relationMapping)$ *// update the source*
16:     **else if** $\big(\text{DIRECTION}(r_j) = ingoing\big)$ **then**
17:       $\pi_2(r_j) := \pi_2(relationMapping)$ *// update the target*
18:     **end if**
19:   **end for**
20: **end for**
21: *// Collect all Model Elements to remove from* t
22: $ME_{del} := \{se_i \in SE_t : \exists se_1 \in SE_{d_{cbprm}} \big((se_1, se_i) \in sm \big(\nexists mn_3 \in MN_{rs_{cbprm}} (se_1, mn_3) \in S_{cbprm}\big)\big)\} \cup \{bp_j \in BP_t : \big(\exists (se_1, se_2) \in sm \big(\exists bp_x \in annotations_{d_{cbprm}}(se_1) \big(type_t(bp_j) = type_{d_{cbprm}}(bp_x)\big)\big)\big)\} \cup \{mn_k \in MN_{rs_{cbprm}} : \exists mn_1 \in MN_{d_{cbprm}} ((mn_1, mn_k) \in S_{cbprm})\}$
23: $ME_t := ME_t \setminus ME_{del}$

Figure 8. The extended apply refinement algorithm. It gets the following inputs: $(cbprm \in CBPRM, t \in T, sm \in SubgraphMappings_{d_{cbrm}, t})$.

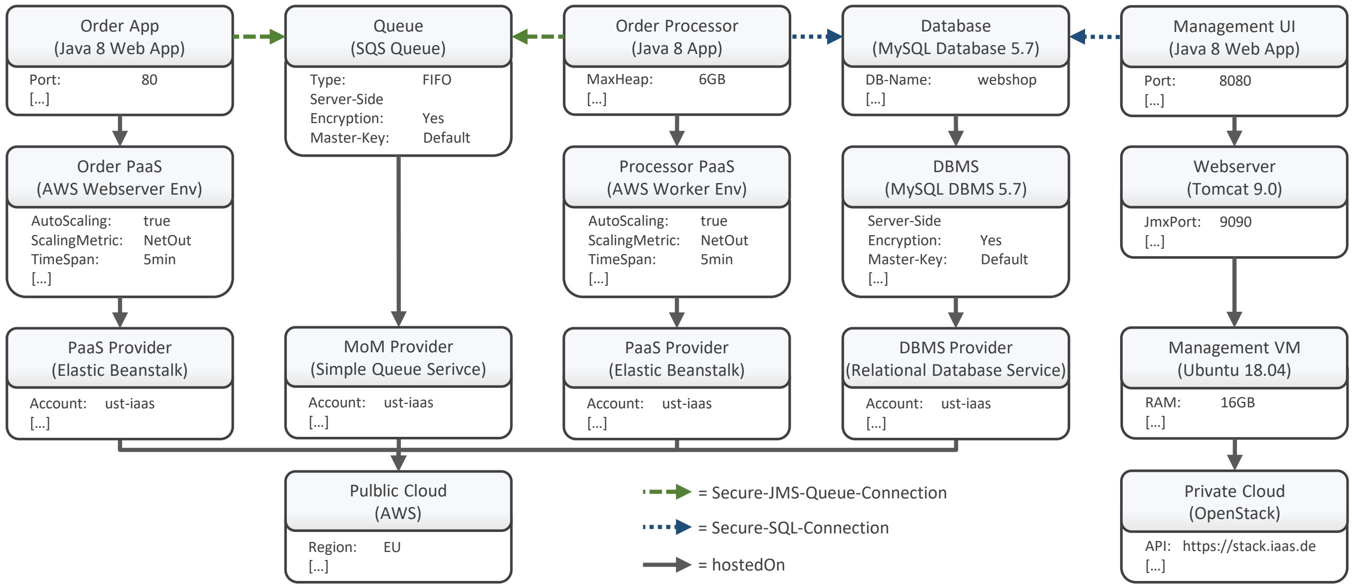


Figure 9. Executable EDMM deployment model, which results from refining the PbDCM shown in Figure 3.

V. Case Study

In the following, we describe a possible refinement of the PbDCM introduced in Figure 3 to an executable deployment model which is shown in Figure 9: All *Component Patterns* hosted on the Public Cloud pattern have been refined to concrete services offered by Amazon (AWS). Thus, the PaaS patterns hosting the *Order App* and the *Order Processor* have been refined to *Elastic Beanstalk Environments* that are preconfigured for automatic scaling to realize the Unpredictable Workload and Stateless Component patterns. Moreover, the types of the *Order App* and the *Order Processor* ensured that appropriate CBPRMs were chosen to refine the PaaS pattern to appropriate Beanstalk environments, i. e., to an *AWS Webserver Env* and to an *AWS Worker Env* respectively.

To realize the Exactly-Once Delivery pattern, the Point-to-Point Channel has been refined to a pre-configured "FIFO" *SQS Queue*, which is hosted on the *Simple Queue Service* offered by AWS. Further, the *Relational Database* pattern was refined to (i) a *MySQL Database 5.7*, (ii) a *MySQL Database Management System (DBMS) 5.7*, and (iii) the *Relational Database Service* offered by AWS. This is required, since a DBMS is obligatory to run a database, while the Relational Database Service provides and maintains the DBMS. To compensate the annotated Information Obscurity pattern, the SQS Queue and the DBMS are configured to use "Server-Side Encryption". Similarly to the Relational Database pattern, the Execution Environment pattern hosting the Management UI has been refined to multiple *Components*: An *Ubuntu 18.04* and a *Tomcat* webserver are needed since the Management UI is a Java 8 Web App and requires an underlying webserver.

Moreover, we created a video [12][13] showing the described case study in detail, i. e., how the PbDCM shown in Figure 3 can be refined to the executable deployment model illustrated in Figure 9 in an automated manner using our prototype.

VI. Prototypical Validation

To prove the practical feasibility of the extended modeling concept, we implemented a prototype based on the *Topology Orchestration Specification for Cloud Applications (TOSCA)* [10] and the open-source ecosystem OpenTOSCA [14][15] TOSCA is a standardized modeling language for automating the deployment and management of cloud applications in a portable way. We chose TOSCA as our basis as it is ontologically extensible [16] and can be mapped to EDMM as follows:

In TOSCA, a declarative deployment model can be expressed by a so-called *Topology Template*. Thereby, *Components* and *Relations* in a PbDCM are represented in TOSCA as *Node Templates* and *Relationship Templates*, which are instances of *Node Types* and *Relationship Types*, respectively. Thus, similar to our extended metamodel where, e. g., *Component Types* define the semantics for *Components*, Node Types and Relationship Types are defining the semantics for the Node and Relationship Templates. We realize *Component Patterns* and *Component Pattern Types* also as Node Templates and Node Types. To differentiate "Pattern Node Types" from "normal" Node Types in TOSCA, a *Tag* in the Node Type is used. Thus, all instances of Node Types having this pattern-tag identifies the corresponding Node Templates as *Component Patterns*.

To annotate Node Templates in a Topology Template by *Behavior Patterns*, *Policies* can be used in TOSCA. The semantics of a Policy is hereby defined by a *Policy Type*. Hence, *Relation Behavior Patterns* and *Component Behavior Patterns* can be mapped to Policies, while *Relation Behavior Pattern Types* and *Component Behavior Pattern Types* are represented by Policy Types in TOSCA. However, according to the TOSCA Specification [10], Relationship Templates cannot be annotated using Policies. Thus, we extended the TOSCA metamodel to support annotating Policies at Relationship Templates during modeling time. This, however, does not influence the standard

compatibility of our implementation since Topology Templates that contain patterns are abstract PbDCMs, and, hence, cannot be deployed directly. By refining a PbDCM in TOSCA, a standard compliant model is generated as the refined Topology Template does not contain any more patterns, i. e., all Policies attached to Node and Relationship Templates have been removed. Thus, a refined Topology Template is standard conform and can be automatically deployed. Moreover, since we only use elements of a Topology Template that can be mapped to an EDMM-based deployment model, a refined Topology Template conforms to EDMM as no policies are contained.

Our prototype is part of the OpenTOSCA ecosystem [15]. OpenTOSCA is an implementation of the TOSCA standard and consists of three components: (i) Winery [17], which provides modeling capabilities, (ii) the OpenTOSCA Container [18], which enables automated orchestration and provisioning, and (iii) the OpenTOSCA UI, offering management functionality to the user. Since our concept focuses on modeling, we extended Winery to support the modeling of *Behavior Patterns*, and the creation and refinement of PbDCMs and CBPRMs.

VII. Related Work

Diverse approaches in the context of MDA and deployment models mention patterns, nevertheless we present selected related work sharing the definition of patterns by Alexander et al. [6] as proven solutions solving recurring problems.

PbDCMs and their refinement is based on the concept of *Model-driven Architecture (MDA)* [19]: A PbDCM represents a *Platform Independent Model (PIM)*, which is, in the context of MDA, transformed into a *Platform Specific Model (PSM)*, represented by the refinement to an executable deployment model. There are diverse approaches to transform a PIM into a PSM present. For instance, the approach of Mellor et al. [20] requires a definition and implementation of a mapping between the abstract metamodel and the metamodel of the target platform. Within our approach, the CBPRMs can be automatically applied, and, thus, combine the mapping and implementation.

Multiple approaches address the transformation of deployment models. The approach of Breitenbücher [11] enables the management of composite cloud applications by an automated relization of management patterns in topologies. Furthermore, Saatkamp et al. [21][22] use logic programming to formalize the problem and context domain of patterns enabling an automated detection and resolving of problems within deployment models. Moreover, the approaches of Eilam et al. [23][24] and Arnold et al. [25][26] focus on an automated transformation of deployment models using predefined transformation steps. Nevertheless, within all of the mentioned approaches patterns are not used to model and define the deployment model.

Hallstrom and Soundarajan [27] refine patterns into subpatterns representing realization variants of abstract patterns which leads to a hierarchy of patterns. Falkenthal et al. [28] introduce a similar approach, which refines patterns into concrete technologies. They further present concrete solutions of patterns capturing reusable implementation realizations, such as code snippets [29][30], as well as aggregation operators which allow the combination of multiple concrete solutions into an overall solution [31]. The introduced CBPRMs can be considered as concrete solutions, a combination through aggregation operators will be part of future work. Eden et al. [32] present an approach for an automated application of patterns to add source code to a given program. Within this work, patterns are specified on an abstract level and realized in a specific program in advance. Even though, those works do not focus on modeling and defining deployment models by a pattern application, a combination of our approach with the presented ones will be considered in future work.

Schürr [33] presents *Triple Graph Grammars (TGGs)* to define graph transformations in a general manner. Therefore, correspondence graphs in TGGs specify correspondences between nodes. In contrast, the presented Relation Mappings in CBPRMs focus on redirecting external relations to the exchanged graph fragment. Bolusset and Oquendo [34] introduce a software architecture refinement approach using *transformation patterns* based on rewriting logic. Similarly, Lehrig [35] introduces the *Architectural Template (AT)* method to apply patterns in terms of reusable modeling templates to software architectures. In contrast, transformation patterns and ATs define rewriting rules of architectures and do not use patterns as components to be refined to concrete technologies.

Di Martino et al. [36] describe the composition of cloud services to cloud applications using patterns. Further, they introduce a semantic model of patterns describing business processes, cloud applications, and mappings to required cloud resources for their implementation [37]. Contrary, those mappings cannot be used to describe or refine deployment models.

VIII. Conclusion & Future Work

Using the Pattern-based Deployment and Configuration Model (PbDCM) approach, the deployment becomes more variable as *Component Patterns* can be automatically refined to different technologies and vendors for each deployment. For example, while one modeler chooses AWS as a public cloud provider, a second one may choose the Google Cloud. Moreover, PbDCMs reduce the required knowledge how technologies must be configured to meet non-functional requirements as *Behavior Patterns* can be annotated at *Structure Elements* to abstractly specify their requirements. Thus, if an application experiences Unpredictable Workload [5], the pattern can be annotated to the corresponding *Components*, which are then automatically refined to an appropriate configuration. For example, configurations required by the General Data Protection Regulation (GDPR) can be realized by an appropriate selection of behavioral patterns, such as the Secure Channel pattern [9].

However, to detect applicable CBPRMs that are able to refine patterns in a PbDCM, our approach builds upon isomorphic subgraph matching. Thus, if any *Structure Element* in a PbDCM is changed by applying a CBPRM, another CBPRM, which may have been applicable before, may not be applicable anymore as the detector subgraph cannot be found. Hence, the order in which CBPRMs are applied is important and may result in different solutions. We plan to tackle this issue in

future work by generating possible permutations of CBPRMs. Moreover, to close the gap between abstract architectures and deployment models we plan to combine the approach of Guth and Leymann [38] with the presented one. Thereby, first architectures are described using abstract patterns [38] which are then refined to more concrete patterns [28], and replaced by concrete technologies using the presented approach. Finally, we plan to use the *Cloud Data Patterns for Confidentiality* [39] to enhance the security of data stored in cloud environments.

Acknowledgment

This work was partially funded by the German Research Foundation (DFG) project SustainLife (379522012).

References

[1] U. Breitenbücher *et al.*, "Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA," in *International Conference on Cloud Engineering (IC2E 2014)*. IEEE, Mar. 2014, pp. 87–96.

[2] M. Wurster *et al.*, "The Essential Deployment Metamodel: A Systematic Review of Deployment Automation Technologies," *Software-Intensive Cyber-Physical Systems (SICS)*, Aug. 2019.

[3] C. Endres *et al.*, "Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications," in *Proceedings of the 9th International Conference on Pervasive Patterns and Applications (PATTERNS)*. Xpert Publishing Services, Feb. 2017, pp. 22–27.

[4] L. Harzenetter *et al.*, "Pattern-based Deployment Models and Their Automatic Execution," in *11th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2018)*. IEEE Computer Society, Dec. 2018.

[5] 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.

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

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

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1994.

[9] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad, *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons, Inc., Jan. 2006.

[10] OASIS, *TOSCA Simple Profile in YAML Version 1.3*, Organization for the Advancement of Structured Information Standards (OASIS), 2019.

[11] U. Breitenbücher, "Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements," Dissertation, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, 2016.

[12] L. Harzenetter, "Demonstration Video," 2019, URL: https://youtu.be/zmU35Detr60 [accessed: 2020-02-18].

[13] ——, "Demonstration TOSCA Repository," 2019, URL: https://github.com/lharzenetter/tosca-definitions [accessed: 2020-02-18].

[14] University of Stuttgart, "OpenTOSCA," 2019, URL: https://github.com/OpenTOSCA [accessed: 2020-02-18].

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

[16] A. Bergmayr *et al.*, "A Systematic Review of Cloud Modeling Languages," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, Feb. 2018.

[17] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, "Winery – A Modeling Tool for TOSCA-based Cloud Applications," in *Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013)*. Springer, Dec. 2013, pp. 700–704.

[18] T. Binz *et al.*, "OpenTOSCA – A Runtime for TOSCA-based Cloud Applications," in *Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013)*. Springer, Dec. 2013, pp. 692–695.

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

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

[21] K. Saatkamp, U. Breitenbücher, O. Kopp, and F. Leymann, "Application Scenarios for Automated problem Detection in TOSCA Topologies by Formalized Patterns," in *Papers From the 12th Advanced Summer School on Service Oriented Computing*. IBM Research Division, Oct. 2018, pp. 43–53.

[22] ——, "An Approach to Automatically Detect Problems in Restructured Deployment Models based on Formalizing Architecture and Design Patterns," *SICS Software-Intensive Cyber-Physical Systems*, pp. 1–13, 2019.

[23] T. Eilam *et al.*, "Managing the configuration complexity of distributed applications in Internet data centers," *Communications Magazine*, vol. 44, no. 3, pp. 166–177, Mar. 2006.

[24] T. Eilam, M. Elder, A. V. Konstantinou, and E. Snible, "Pattern-based Composite Application Deployment," in *Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011)*. IEEE, May 2011, pp. 217–224.

[25] W. Arnold, T. Eilam, M. Kalantar, A. Konstantinou, and A. Totok, "Pattern Based SOA Deployment," in *Proceedings of the Fifth International Conference on Service-Oriented Computing (ICSOC 2007)*. Springer, Sep. 2007, pp. 1–12.

[26] ——, "Automatic Realization of SOA Deployment Patterns in Distributed Environments," in *Proceedings of the 6th International Conference on Service-Oriented Computing (ICSOC 2008)*. Springer, Dec. 2008, pp. 162–179.

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

[28] M. Falkenthal *et al.*, "Leveraging Pattern Application via Pattern Refinement," in *Proceedings of the International Conference on Pursuit of Pattern Languages for Societal Change (PURPLSOC 2015)*. epubli, Jun. 2015, pp. 38–61.

[29] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, and F. Leymann, "From Pattern Languages to Solution Implementations," in *Proceedings of the Sixth International Conferences on Pervasive Patterns and Applications (PATTERNS 2014)*. Xpert Publishing Services, May 2014, pp. 12–21.

[30] M. Falkenthal and F. Leymann, "Easing pattern application by means of solution languages," in *Proceedings of the 9th International Conferences on Pervasive Patterns and Applications (PATTERNS)*. Xpert Publishing Services (XPS), 2017, pp. 58–64.

[31] M. Falkenthal, J. Barzen, U. Breitenbücher, and F. Leymann, "On the Algebraic Properties of Concrete Solution Aggregation," *SICS Software-Intensive Cyber-Physical Systems*, Aug. 2019.

[32] A. Eden, A. Yehudai, and J. Gil, "Precise Specification and Automatic Application of Design Patterns," in *Proceedings of the 12th IEEE International Conference Automated Software Engineering (ASE 1997)*. IEEE, Nov. 1997, pp. 143–152.

[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] T. Bolusset and F. Oquendo, "Formal Refinement of Software Architectures Based on Rewriting Logic," in *Proceedings of the International Workshop on Refinement of Critical Systems*, 2002, pp. 200–202.

[35] S. M. Lehrig, "Efficiently Conducting Quality-of-Service Analyses by Templating Architectural Knowledge," Dissertation, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, 2018.

[36] 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.

[37] 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.

[38] J. Guth and F. Leymann, "Pattern-based rewrite and refinement of architectures using graph theory," *Software-Intensive Cyber-Physical Systems (SICS)*, pp. 1–12, Aug. 2019.

[39] S. Strauch, U. Breitenbücher, O. Kopp, F. Leymann, and T. Unger, "Cloud Data Patterns for Confidentiality," in *Proceedings of the 2nd International Conference on Cloud Computing and Services Science (CLOSER 2012)*. SciTePress, Apr. 2012, pp. 387–394.