Institute of Architecture of Application Systems

Automated Detection of Design Patterns in Declarative Deployment Models

Lukas Harzenetter, Uwe Breitenbücher, Ghareeb Falazi,
Frank Leymann, and Adrian Wersching

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

BIBTEX

```
@inproceedings{Harzenetter2021_PatternDetection,
  author    = {Harzenetter, Lukas and Breitenb{\"u}cher, Uwe and
               Falazi, Ghareeb and Leymann, Frank and Wersching, Adrian},
  title     = {Automated Detection of Design Patterns in Declarative
               Deployment Models},
  booktitle = {Proceedings of the 2021 IEEE/ACM 14th International
               Conference on Utility Cloud Computing (UCC 2021)},
  year      = 2021,
  month     = dec,
  pages     = {36-45},
  publisher = {ACM},
  doi       = {10.1145/3468737.3494085}
}
```

© ACM 2021

This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version is available at ACM: https://doi.org/10.1145/3468737.3494085

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

University of Stuttgart
Germany

Automated Detection of Design Patterns in Declarative Deployment Models

Lukas Harzenetter, Uwe Breitenbücher, Ghareeb Falazi, Frank Leymann, and Adrian Wersching
{harzenetter,breitenbuecher,falazi,leymann,wersching}@iaas.uni-stuttgart.de
University of Stuttgart, Institute of Architecture of Application Systems (IAAS)
Stuttgart, Germany

ABSTRACT

In recent years, many different deployment automation technologies have been developed to automatically deploy cloud applications. Most of these technologies employ declarative deployment models to describe the deployment of a cloud application by modeling its components, their configurations as well as the relations between them. However, while modeling the deployment of cloud applications declaratively is intuitive, declarative deployment models quickly become complex as they often contain detailed information about the application's components and their configurations. As a result, immense technical expertise is typically required to understand the semantics of a declarative deployment model, i. e., what gets deployed and how the components behave. In this paper, we present an approach that automatically detects design patterns in declarative deployment models. This eases understanding the semantics of deployment models as only the abstract and high-level semantics of the detected patterns must be known instead of technical details about components, relations, and configurations. We demonstrate an open-source implementation based on the Topology and Orchestration Specification for Cloud Applications (TOSCA) and the graphical open-source modeling tool Winery. In addition, we present a detailed case study showing how our approach can be applied in practice using the presented prototype.

CCS CONCEPTS

• **Software and its engineering** → **Software architectures**; **Software system models**; **Design patterns**; **Cloud computing**.

KEYWORDS

Declarative Deployment Models, Design Patterns, Pattern Detection, TOSCA, Eclipse Winery

ACM Reference Format:
Lukas Harzenetter, Uwe Breitenbücher, Ghareeb Falazi, Frank Leymann, and Adrian Wersching. 2021. Automated Detection of Design Patterns in Declarative Deployment Models. In *2021 IEEE/ACM 14th International Conference on Utility and Cloud Computing (UCC'21), December 6–9, 2021, Leicester, United Kingdom.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3468737.3494085

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
UCC'21, December 6–9, 2021, Leicester, United Kingdom
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8564-0/21/12. . . $15.00
https://doi.org/10.1145/3468737.3494085

1 INTRODUCTION

Since the manual deployment and configuration of applications is error-prone and time consuming, its automation is important to ensure repeatable and reliable executions—especially in the Cloud [4, 34]. Therefore, a plethora of *deployment automation technologies*, e. g., Chef, Ansible, and Terraform have been developed [44]. These technologies typically use *deployment models* to describe the deployment and configuration of application components as well as their relations [4, 44]. Deployment models can be classified into *declarative* and *imperative* deployment models [11]. Imperative deployment models specify the actual process of the deployment, i. e., technical deployment tasks as well as their execution order. Thus, imperative models require a lot of technical expertise to implement these deployment tasks [7]. In contrast, declarative deployment models describe only *what* has to be deployed without specifying the technical execution details. Thereby, declarative models are typically structured in the form of directed, weighted graphs [44] and can be represented graphically, which eases their understandability. Since the most prominent deployment automation technologies support declarative modeling [44], we focus on them in this paper.

In general, declarative deployment models are significantly easier to create in contrast to imperative deployment models because technical details about their execution do not need to be modeled [7, 31]. However, declarative deployment models also quickly become complex as many technical details about components, relations, and their configurations must be described to enable the application's fully-automated deployment. This is especially a challenge for large-scale applications that consist of hundreds of components possibly distributed across multiple different cloud infrastructures using various middleware technologies. Thus, understanding the semantics of the modeled application requires immense technical expertise about the modeled components, relations, and their configurations. For example, to achieve a highly scalable cloud application, queues are often used to enable asynchronous communication between different components of the application. Thereby, it is often a compelling business requirement to guarantee that each message is processed exactly once. If a queue provided by a messaging service is used, the service must therefore ensure *Exactly-once Delivery* [15]. However, identifying that a queue is configured to ensure exactly-once semantics from a deployment model requires technical knowledge about the underlying technology. For instance, a queue hosted on AWS' Simple Queuing Service (SQS)[1] that realizes Exactly-once Delivery, must be of type "FIFO", whereas in Azure's Service Bus Messaging service[2], the queue must be configured to use the "duplication detection" feature. These are very specific technical details, which

[1] https://aws.amazon.com/de/sqs/
[2] https://docs.microsoft.com/en-us/azure/service-bus-messaging

readers must know to understand the semantics of the queue and how it behaves. This becomes a serious challenge for large, multi-cloud deployments where multiple different technologies and providers are involved. Moreover, as default configurations are typically not modeled explicitly, recognizing the behavior of a component is very difficult or even impossible for readers who are not aware of all the technical details. Thus, adapting such models becomes error-prone.

To tackle this issue, we present an approach to automatically detect design patterns in declarative deployment models. This eases understanding the semantics in declarative deployment models as they are explicitly described in the form of design patterns and do not need to be derived by the reader from the technical details and configurations of each component and relation. Moreover, since patterns document proven solutions to a particular recurring problem [1], the terms and concepts are clearly defined providing a common understanding for all readers. Therefore, to present the detected design patterns and where they occur in the model, we reuse a metamodel that has been previously introduced to describe *Pattern-based Deployment Models* (PbDMs) [21, 22]. Thereby, design patterns can be used as nodes to represent structural elements or attached to nodes to express their behavioral characteristics [22]. Hence, PbDMs describe the logical architecture [27] of applications and can be used to communicate the application's semantics to audiences that do not have expert knowledge in all used vendors and technologies.

In previous work [21, 22], PbDMs were introduced to model deployments in an abstract way without the need to specify concrete components, relations, and configurations. Moreover, an approach was presented to refine PbDMs automatically to executable deployment models [21]. On the contrary, in this work we turn the idea around: Instead of manually creating PbDMs that can be automatically refined, we automatically derive PbDMs from existing declarative deployment models to ease understanding their semantics—which is obviously easier to grasp from technology-independent, pattern-based models than from deep technical deployment models. We validate our approach by presenting a prototypical implementation based on Eclipse Winery [25] and applying it on a case study.

2 BACKGROUND, MOTIVATION, AND PROBLEM STATEMENT

An overview of deployment automation is given in Section 2.1, while Sections 2.2 and 2.3 motivate and outline the problem statement.

2.1 Deployment Models and Automation

Since the manual deployment of applications is error-prone and time consuming, automating application deployments is inevitable to achieve reproducible and efficient executions [34]. Available deployment automation technologies mostly use deployment models to describe the deployment of applications [4]. Thereby, two types can be differentiated: (i) *imperative deployment models* and (ii) *declarative deployment models* [11]. Imperative deployment models are process models that explicitly describe the tasks to be executed with all technical details as well as their order and the data flow between them. Thus, imperative deployment models describe *how* the modeled application is deployed. Hence, to perform the deployment, the corresponding script or workflow is simply executed by a suitable deployment automation engine. In contrast, declarative deployment models describe only the components of an application to be deployed including their relations and configurations. Hence, declarative deployment models describe only *what* has to be deployed, but not *how*. Thus, instead of directly consuming and executing a declarative deployment model, declarative deployment automation engines must derive the tasks that need be performed to instantiate an application in the correct order. As a result, declarative deployment models are easier to create as no technical expertise about the actual deployment tasks is required [7]. Therefore, modeling application deployments declaratively has prevailed in practice: The deployment automation technologies, which are most used in industry and research, all use declarative models to describe the deployment of applications [44]. Additionally, by performing a review of the 13 most used deployment automation technologies, Wurster et al. [44] derived a meta-model, the *Essential Deployment Metamodel* (EDMM), that consists of the general modeling elements all these technologies have in common. Application deployments modeled in EDMM can be automatically transformed to models of concrete technologies, such as Terraform, Ansible, or Kubernetes [43] and executed using these production-ready technologies [45]. Therefore, we use EDMM to describe our concepts independent of a deployment technology.

2.2 Motivating Scenario and Running Example

In general, declarative deployment models can be represented as directed, weighted graphs whereby *components* are represented as nodes while edges define their *relations* [44]. In EDMM, both, components and relations, are semantically defined by reusable types, i. e., *component types* and *relation types* respectively, defining the properties and operations a component or relation has. For example, Fig. 1 shows a simplified declarative deployment model of a distributed order application consisting of a *Webshop* component that communicates with an *Order Processor* component using a queue component. The Webshop is realized in the form of an *Angular 11 Web App* as represented by the the component's type, shown in parentheses. It is publicly accessible via the internet via an *Elastic Beanstalk Webserver Environment* on which it runs. Similarly, the Order Processor, a *Java 11 App*, is hosted on *AWS Lambda* in the form of a *Java 11 Lambda Function* and is executed whenever there are new orders issued from the Webshop to the SQS Queue. To store new orders persistently, the Order Processor connects to a *MySQL 8.0 Database* that is running on a Relational Database Service (RDS) *MySQL 8.0 Environment*. Then, employees can access the database using a *Management UI* hosted on a *Ubuntu 20.04* virtual machine (VM) running on a local *OpenStack* infrastructure to process and ship the orders. In such a declarative deployment model, the components are instances of component types defining their semantics. For example, the Angular 11 Web App component type defines operations to install, start, and configure such an application, as well as its required properties. In this case, the "Context" path from which the Webshop will be reachable is "/shop", while the name of the MySQL Database, e. g., should be "webshop". Finally, to express the semantics of relations between the components, they are also instances of relation types. Thus, to indicate that the Webshop is running on an Elastic Beanstalk Webserver Environment and it connects to the *SQS Queue*, the two relations are, as illustrated in Fig. 1, instances of the relation types *hostedOn* and *SecureQueueConnection*.

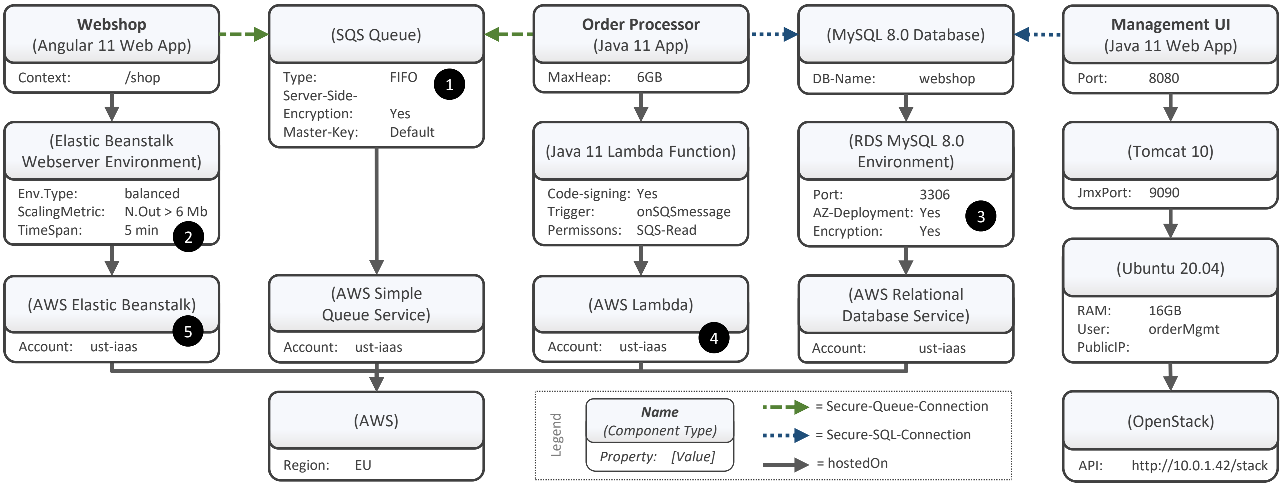

Figure 1. A simplified declarative deployment model describing a hybrid cloud application consisting of multiple distributed components.

2.3 Problem Statement

As outlined above and illustrated in Fig. 1, declarative deployment models are very useful to describe the components, relations, and configurations of an application. However, understanding the semantics hidden in such deployment models is very difficult and requires immense technical expertise. For example, to recognize that the queue used between the Webshop and the Order Processor enforces exactly-once delivery, it must be known that AWS uses the queue type "FIFO" to achieve this. In Fig. 1, the SQS Queue realizes this and is highlighted by ❶. Similarly, to model that the Webshop must be scaled horizontally, the corresponding Elastic Beanstalk Webserver Environment hosting the Order Processor is configured to be of type "balanced", instead of "single", see ❷. Thus, the Webshop is automatically scaled if the network output is constantly over 6 Mb/s for more than five minutes. Moreover, not only the behavior of components is challenging to detect from deployment models, but also the structural concepts realized by the employed components. For example, it must be known that the Database Management System (DBMS) is configured to use strict consistency since the Relational Database Service Environment (❸) uses AZ-Deployments, which is again highly vendor-specific knowledge required to identify this behavior. Similarly, identifying the used service types may be a problem if the reader is not aware of them. For example, a reader must know that AWS Lambda is a Function as a Service (FaaS) offering (❹) while AWS Elastic Beanstalk, see ❺ in Fig. 1, is a Platform as a Service (PaaS) offering. Moreover, this only holds for this particular deployment model. If another cloud provider or even several are used, understanding these deployment semantics becomes almost impossible if knowledge about the different offerings and their configurations is missing. Additionally, an application's deployment model describes its logical architecture and is therefore also suitable for communication. Thus, the following research question arises:

> *"How can the semantics of technical declarative deployment models be automatically detected and represented in a way that significantly reduces the technical expertise the reader needs to understand them?"*

3 PATTERN-BASED REPRESENTATION OF DEPLOYMENT SEMANTICS

First the idea of the proposed approach is described in Section 3.1 followed by an application to the motivation scenario in Section 3.2.

3.1 General Idea

Declarative deployment models can become very complex and contain many different components that represent various vendors and technologies as discussed in the previous section. Thus, understanding their semantics is very difficult. To tackle this, the idea of this paper is to (i) automatically detect design patterns in declarative deployment models by mapping concrete components, relations, and their configurations to abstract design patterns and (ii) representing them as *Pattern-based Deployment Models* (PbDMs)[3] [21, 22], which are deployment models that can contain structural patterns instead of concrete components and behavior patterns attached to nodes to describe their behavioral semantics. As a result, only design patterns need to be understood in contrast to deep technical details of numerous technologies and providers. Especially for deploying Cloud applications, there are different pattern languages available that can be used to describe the semantics of components, relations, and their configurations [21], e. g., the *Cloud Computing Patterns* [15], the *Enterprise Integration Patterns* [23], and the *Security Patterns* [38]. Moreover, the approach can be applied to the 13 most used deployment technologies [44] as we employ EDMM.

Originally, PbDMs were introduced to abstractly model applications with design patterns instead of using concrete technologies and providers to ease their creation [21]. The existing approach also enables the automated refinement of the modeled design patterns to concrete components and their corresponding configurations for each deployment of the application [21, 22]. Thereby, two types of patterns are differentiated [22]: *component patterns* which are used to describe components abstractly and *behavior patterns* which can

[3] In [22], PbDMs were extended to *Pattern-based Deployment and Configuration Models (PbDCMs)* to support pattern-based configuration of components and relations. However, for simplicity we use PbDMs to refer to PbDCMs in this paper.

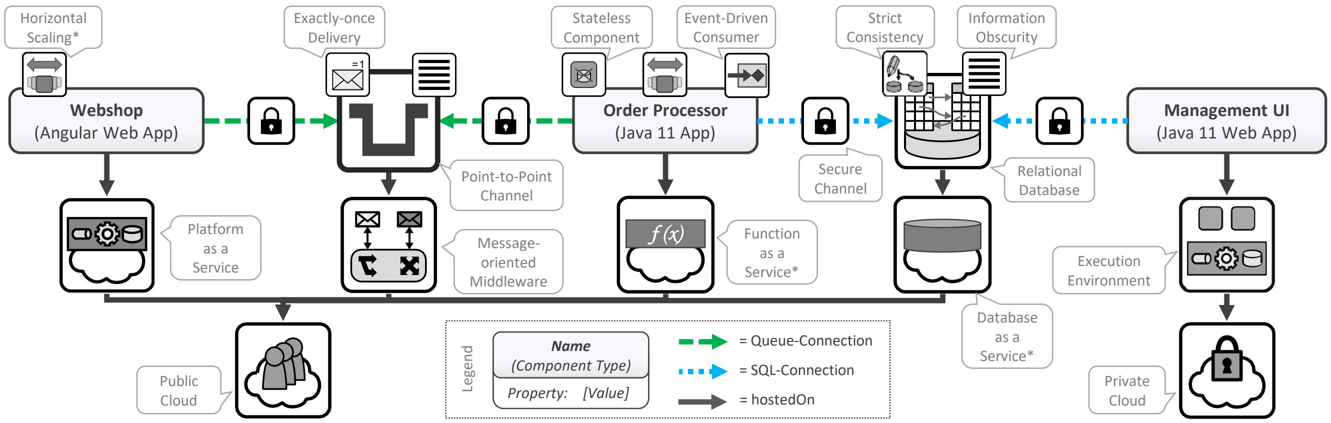

Figure 2. Pattern-based Deployment Model that is generated by applying the proposed approach to the deployment model shown in Fig. 1.
(*) *Horizontal Scaling* [15, 46], Function as a Service [47], and Database as a Service [47] are not documented as patterns but are common cloud concepts.

be annotated to components, relations, and component patterns to abstractly define their behavior. Therefore, while PbDMs not only ease the creation of deployment models but also ease understanding their semantics, we propose to reverse the original approach presented by Harzenetter et al. [21, 22] and enable the automated detection of design patterns in declarative deployment models and represent them in the form of PbDMs. Thus, the goal is to ease understanding the semantics of the modeled application deployment as components and their behavior are represented by patterns instead of technical details that need to be understood.

3.2 Motivation Scenario as a Pattern-based Deployment Model

To demonstrate how intuitive the result is if declarative deployment models are represented as PbDMs, we describe how the application shown in Fig. 1 can be represented as a PbDM. Therefore, Fig. 2 illustrates the running example application as a PbDM: Instead of describing concrete vendors and technologies, such as the Elastic Beanstalk Webserver Environment, the AWS Elastic Beanstalk, and the AWS component the Webshop is hosted on, their semantics are now abstractly represented in the form of *component patterns* [22]. Thus, the Webshop is hosted on a *Platform as a Service* [15] pattern that is provided by a *Public Cloud* [15] pattern in Fig. 2. Similarly, the SQS queue is an implementation of the *Point-to-Point Channel* pattern [23] (a.k.a. *Queue*) that is running on a *Message-oriented Middleware* [15] provided by the same Public Cloud in which the Webshop is hosted on. Moreover, the components of the application modeled in Fig. 1 are configured to behave in a particular way. To understand the behavior of a component or relation from a declarative deployment model, immense technical expertise about the corresponding vendor or technology is required. In contrast, the behavior of components and relations is explicitly described in PbDMs in the form of *behavior patterns* [22] annotated to the corresponding component, component pattern, or relation. Hence, recognizing that, e. g., the Point-to-Point Channel ensures *Exactly-once Delivery* is significantly easier. Similarly, the Order Processor is running on a *Function as a Service (FaaS)* and is annotated to be a *Stateless Component* [15] that uses *Horizontal Scaling*. To understand that the Order Processor cannot hold any state in between requests, hence realizes the Stateless Component pattern, as well as that it is automatically scaled horizontally based on the original deployment model shown in Fig. 1, many technical details about how AWS Lambda works need to be understood. In contrast, in the PbDM in Fig. 2 the Order Processor's behavior is abstractly shown. Additionally, since the Java 11 Lambda Function is executed after a new message arrives in the SQS Queue—defined by the "Trigger" property with the value "onSQSmessage"—the Order Processor realizes the *Event-Driven Consumer* [23] pattern. Furthermore, the Order Processor connects to a *Relational Database* [15], which is running in a *Database as a Service (DBaaS)* offering in the same Public Cloud. Moreover, it is also possible to detect additional semantics hidden in the relations. For example, to communicate between the Webshop and the Order Processor, both use a *Secure Queue Connection* relationship type to describe their communication with the SQS Queue. This can be abstractly represented by a relation of type *Queue Connection* that is annotated with the *Secure Channel* [38] pattern. Finally, the Management UI hosted on a Ubuntu VM running on OpenStack can be represented by an *Execution Environment* [15] that is provided by a *Private Cloud* [15] pattern. However, while *Horizontal Scaling* is not documented as a pattern, we still included it to describe the scaling behavior of an application since scaling horizontally is a common behavior in the Cloud to compensate changing workloads [15, 46]. Similarly, FaaS and DBaaS are common cloud services [46], but are not documented as design patterns.

4 THE UNDERLYING PREVIOUS APPROACH

As shown in the previous section, the technical knowledge required to understand the deployment semantics of an application modeled as a PbDM is significantly reduced in comparison to a technical declarative deployment model. Therefore, we propose an iterative approach to automatically generate PbDMs from executable declarative deployment models in the next section. But before we introduce our new approach in the next section, we describe in this section the previous work [21, 22] on which our new approach is based as it is a reversed version of this previous work.

In previous work [21, 22], Pattern-based Deployment Models were modeled by the user to describe the abstract semantics of a desired application deployment. As PbDMs cannot be executed, the

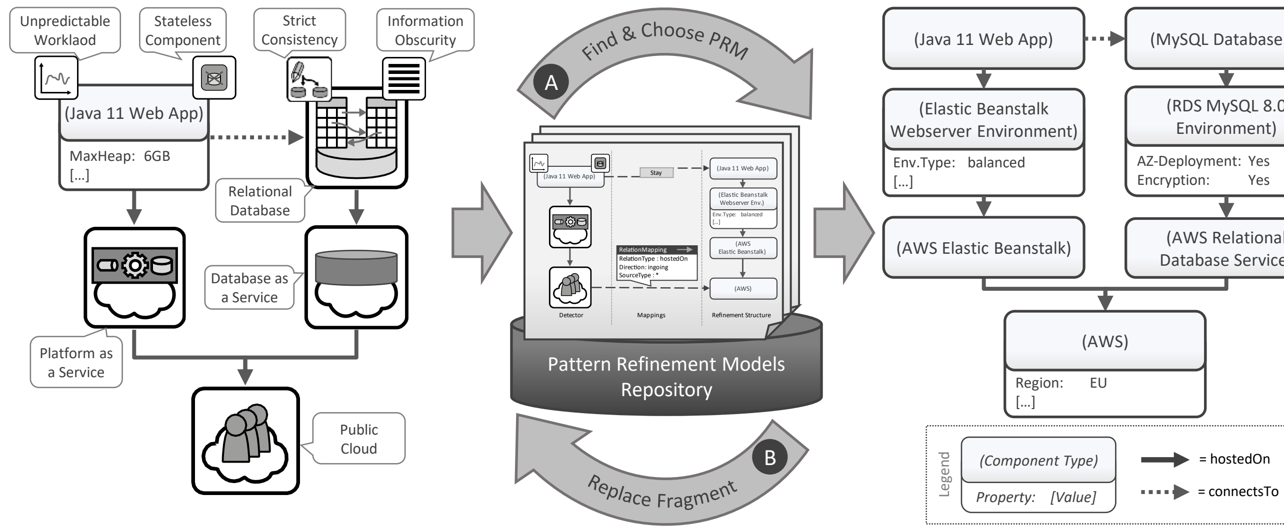

Figure 3. The underlying approach previous approach [21, 22] that refines PbDMs into executable declarative deployment models.

previous work also presented a refinement method and corresponding algorithms to automatically refine a PbDM to an executable deployment model in order to deploy the modeled application. To achieve this, the patterns in PbDMs must be replaced by concrete components, relations, and configurations. Therefore, an iterative approach has been introduced [21, 22] which is illustrated in Fig. 3.

The general idea was to use transformation models, similar to *Triple Graph Grammars (TGGs)* [39], which are used to transform a particular *left-hand-side* graph into a *right-hand-side* graph by describing correspondences between the nodes of the left-hand side and the nodes of the right-hand side in a *correspondence graph*. In the previous approach, so-called *Pattern Refinement Models (PRMs)*[4] [21] were used that define how a set of interconnected patterns can be refined to concrete components, relations, and technical configurations. For example, Fig. 3 illustrates how an application can be refined to an executable declarative deployment model. On the left, a Java 11 Web App that is hosted on the Platform as a Service pattern provided by the Public Cloud pattern, which connects to a Relational Database pattern running on a DBaaS pattern by the same Public Cloud. Additionally, the Java Web App is annotated with an *Unpredictable Workload* [15] pattern and the *Stateless Component* [15] pattern, while the Relational Database must ensure the *Strict Consistency* [15] and the *Information Obscurity* [38] patterns. Thus, to refine the PbDM a set of *applicable* PRMs is first identified whereof one is chosen by a user to be applied to the PbDM. A PRM is applicable if its left-hand side, i. e., a PbDM fragment called *Detector*, can be found as an isomorphic subgraph in the investigated PbDM. Thereby, all nodes and their relations in the detector can be found one-by-one in the PbDM [21]. Hence, if a PRM is applicable, the subgraph in the PbDM matching its detector can be refined to the graph defined in the PRM's right-hand side, i. e., one possible set of concrete components, relations, and configurations realizing the patterns described in the PRM's detector, which is called *Refinement Structure* [21]. For example, if a PRM's detector defines a Relational Database hosted on a DBaaS provided by a Public Cloud pattern, it may define its refinement structure to be a MySQL Database running on an AWS RDS environment. Thus, by finding a subgraph in a PbDM matching the PRM's detector, the PRM can be applied to the PbDM to replace the matching subgraph with the graph defined in its refinement structure. This process repeats until no more patterns can be found in the refined model.

Finally, an executable, declarative deployment model is generated as shown in Fig. 3: The Java 11 Web App is then running on an Elastic Beanstalk Webserver Environment running on AWS' Elastic Beanstalk and connects to a MySQL Database 8 provided by a RDS MySQL 8.0 Environment on AWS' RDS. To realize the annotated behavior of the Java 11 Web App, the Elastic Beanstalk Webserver Environment is configured to scale automatically, while the behavior annotated at the Relational Database is realized by enabling encryption and AZ-Deployments in the RDS MySQL 8.0 Environment.

5 OVERVIEW OF THE METHOD

To detect design patterns in technical declarative deployment models and to represent them as Pattern-based Deployment Models, we present an automated method in this section which is depicted in Fig. 4. Instead of identifying interconnected patterns as subgraphs in a given PbDM and refining them to concrete technologies as presented by the previous work (see Section 4), we now search for subgraphs matching concrete components, relations, and configurations in executable technical declarative deployment models in order to replace them with the corresponding patterns. Thereby, similar to the idea of PRMs, we use *Pattern Detection Models (PDMs)* to automatically generate a PbDM from a given declarative deployment model by replacing subgraphs in the declarative deployment models matching a PDM's left-hand side with their right-hand side in an iterative manner. PDMs are detailed in Section 6.1. Hence, in each iteration of the method, more semantics hidden in the technical

[4]In [22], PRMs were extended to *Component and Behavior Pattern Refinement Models* (CBPRMs) to enable the refinement of annotated behavior patterns to concrete configurations. For simplicity, we use PRMs to refer to CPRMs.

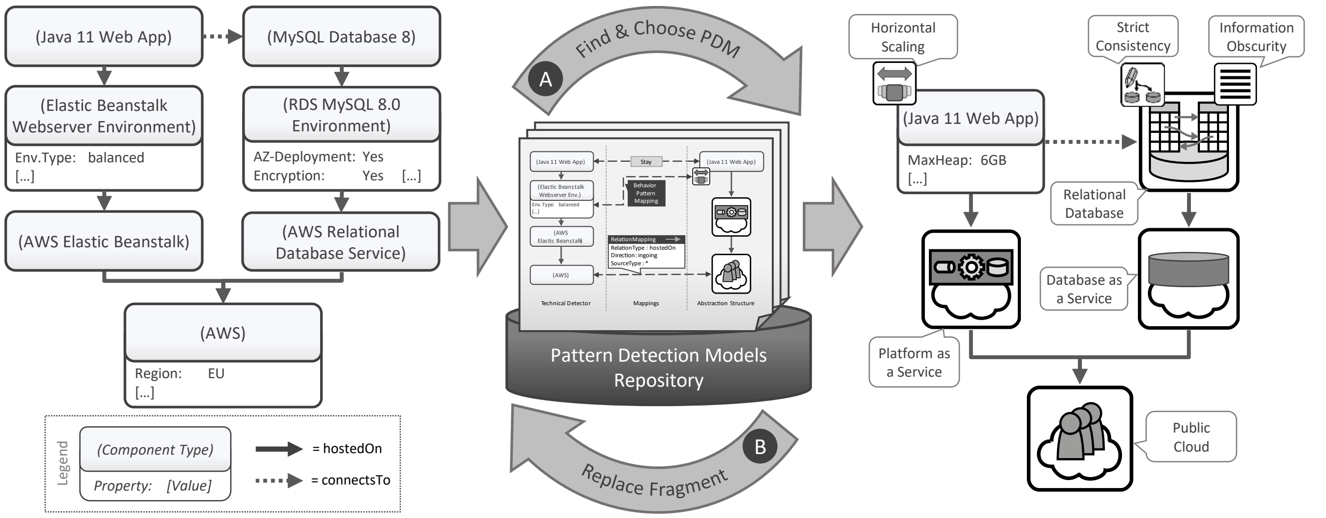

Figure 4. The new method for transforming technical declarative deployment models into Pattern-based Deployment Models.

details of the processed declarative deployment model are detected and represented by abstract design patterns in the resulting PbDM.

For example, on the left of Fig. 4 the same Java 11 Web App that is running on an Elastic Beanstalk Webserver Environment provided by AWS Beanstalk is illustrated as shown on the right side of Fig. 3. In this case, a PDM that defines this subgraph as its left-hand side, which is called *Technical Detector Fragment*, maps the concrete technologies to a Java 11 Web App that is running on a PaaS provided by a Public Cloud, i. e., the PDM's right-hand side which we call *Pattern Abstraction Fragment*. Additionally, since the Elastic Beanstalk Webserver Environment is configured to be balanced, the Java 11 Web App gets annotated with Horizontal Scaling to describe its behavior. In contrast to the PbDM shown in Fig. 3, the generated PbDM neither contains the Unpredictable Workload pattern nor the Stateless Component pattern because it is only possible to detect the modeled scaling behavior but not that this originally resulted from an expected Unpredictable Workload. Thus, we differentiate between PRMs and PDMs in this paper and plan to identify in which cases a PRM and a PDM can work both ways in future work. Moreover, as depicted in the generated PbDM in Fig. 4, both behavior patterns annotated to the Relational Database in the PbDM shown in Fig. 3, i. e., Strict Consistency and Information Obscurity, are annotated to the Relational Database pattern detected in Fig. 4. This is possible since the corresponding configurations are directly reflected as properties in the declarative deployment model.

6 DETECTING DESIGN PATTERNS IN DECLARATIVE DEPLOYMENT MODELS

In Section 6.1, Pattern Detection Models (PDMs) are explained in detail, while Section 6.2 presents two concepts that describe how patterns can be detected in technical declarative deployment models.

6.1 Pattern Detection Models

To replace subgraphs in a declarative deployment model that match the Technical Detector Fragment of a PDM, a set of rewriting rules are needed. These rules are called *Mappings* [21] and form the correspondence graph between the left-hand side and right-hand side in PRMs and are reused and extended for PDMs. Therefore, we use the following existing mappings between patterns, components, and relations from previous work: (i) To define how relations that are in- or outgoing from a matching component can be redirected to a corresponding pattern, *Relation Mappings* [21] are used. For example, the PDM shown on the left of Fig. 5 defines a relation mapping between the AWS component and the Public Cloud pattern which redirects all relations of type *hostedOn* that are ingoing at the AWS component to the Public Cloud. (ii) To define that a component must not be replaced when a PDM is applied, *Stay Mappings* [22] are used to specify where the component must be located in the Pattern Abstraction Fragment. For example, for the right PDM in Fig. 5 to be applicable to a declarative deployment model, it must match the Java 11 Web App which is hosted on a Tomcat 10 Webserver running on a Ubuntu 20.04 in vSphere. Thus, to define that the Java 11 Web App will be hosted on the Execution Environment pattern in the PbDM, a stay mapping is used. This ensures that all the properties specified for the Java 11 Web App are still in the model after applying the PDM. (iii) Lastly, *Property Mappings* [42] enable the mapping of a property defined at a component to a property of a pattern. For example, if the "Region" property is set in an AWS component of a deployment model matching the PDM's left-hand side in Fig. 5, its value is transferred to the "Location" of the Public Cloud pattern.

In addition to the existing mappings, we introduce a new mapping to detect behavior patterns conditionally. Therefore, we introduce *Behavior Pattern Mappings* to map a set of properties with specific values to a behavior pattern. Thereby, concrete mappings between the technical configuration properties in the left-hand side of the PDM can be mapped to behavior patterns they implement in the right-hand side. For example, in the configuration of the SQS Queue illustrated in the left PDM in Fig. 5, the property "Type" is set to "FIFO" which realizes the Exactly-once Delivery pattern. Thus, by specifying a behavior pattern mapping between the SQS Queue's "Type" property and the Exactly-once Delivery pattern, the pattern

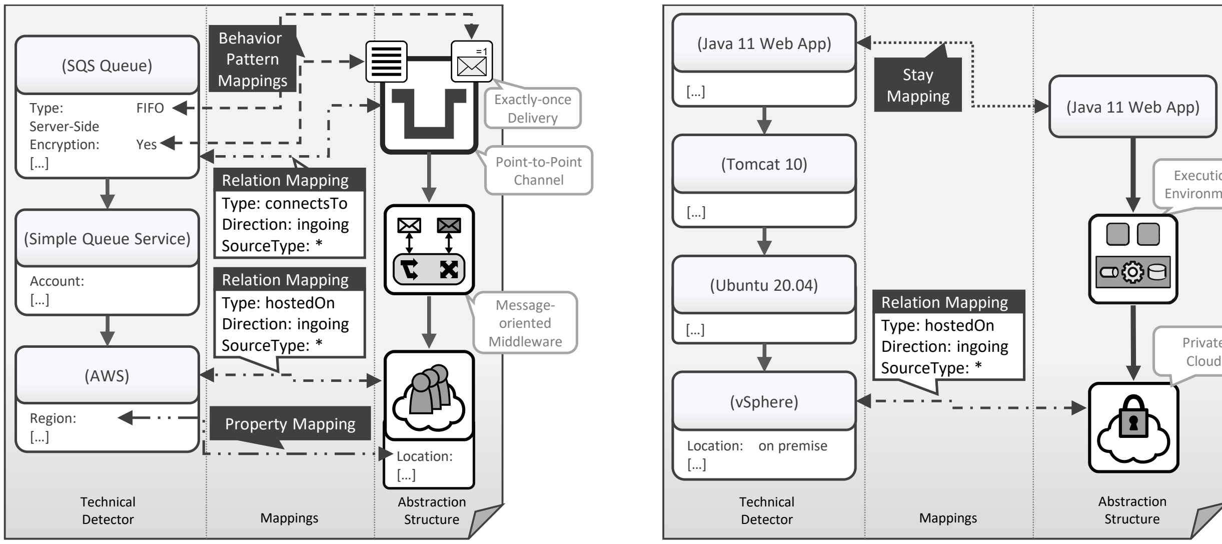

Figure 5. Two examples for Pattern Detection Models (PDMs).

can be automatically detected whenever this particular configuration of a matching SQS Queue is found in a technical declarative deployment model. Similarly, if the "Server-Side Encryption" property is set to true in a matching SQS Queue component, the Information Obscurity pattern is implemented and, thus, can be detected.

6.2 Deployment Model Abstraction Algorithm

In this section, we abstractly describe the *Deployment Model Abstraction Algorithm* which transforms a given technical deployment model into a PbDM. The algorithm is depicted in Fig. 6 which gets a declarative deployment model, *ddm*, as an input. To automatically detect patterns in declarative deployment models, we first create a copy of the model, see line 2 in Fig. 6, and then start to search for *applicable* PDMs in line 3. Thereby, a PDM is applicable if its Technical Detector fragment can be found as a subgraph in the currently investigated deployment model. If such subgraphs can be found in the PbDM, i. e., the component types and relation types of components and relations defined in the Technical Detector, as well as their properties, match a subgraph in a technical deployment model [22], we are able to replace them with the Abstraction Structure fragments defined in the corresponding applicable PDMs. Therefore, multiple PDMs can be applicable at a time whereof one must be chosen by a user to be applied to the deployment model (see line 4). Additionally, one PDM may be applicable multiple times as the investigated model may contain multiple stacks that all match the PDM's Technical Detector. Hence, all matching subgraphs are calculated in line 5 while one is chosen in line 6.

In the next step, the PDM is applied to the declarative deployment model whereby the selected matching subgraph is replaced with the PDM's Abstraction Structure. For example, the Technical Detector shown in the left PDM in Fig. 5 can be found in the deployment model shown in Fig. 1. Hence, the SQS Queue provided by AWS can be replaced with the Abstraction Structure defined in the left PDM of Fig. 5. In contrast, the Technical Detector of the right PDM cannot be found as a subgraph in Fig. 1. To apply a selected PDM, first all component patterns and their relations among each other are added to the copy of the declarative deployment model (see line 7 in Fig. 6). Then, all relations that are in- and outgoing of the matching subgraph are redirected to the added pattern structure fragment in line 8, while the property mappings are applied by moving the corresponding values from the matching components in line 9.

In addition, to enable a more generic matching, behavior pattern mappings can be used to conditionally detect the components' behavior. For example, to detect that a SQS Queue is ensuring exactly-once delivery, it must be configured to be of type "FIFO". Hence, if a subgraph in a declarative deployment model can be found that matches the left PDM's Technical Detector shown in Fig. 5, the Exactly-once Delivery pattern is annotated to the Point-to-Point Channel representing the semantics of a queue only if the SQS Queue is configured accordingly. Otherwise, the Pattern Abstraction Fragment of the PDM is added to the declarative deployment model without the Exactly-once Delivery pattern. This is realized in the algorithm shown in Fig. 6 between lines 10 and 15: Hereby, all behavior patterns defined in the PDM's Abstraction Structure are investigated if they are part of a behavior pattern mapping. If there are behavior pattern mappings for a behavior pattern, it is only added to the generated model if all properties are set in the matching subgraph as defined by the behavior pattern mappings. In contrast, if a PDM does not define behavior pattern mappings for the annotated patterns the annotated behavior patterns are considered to be realized by detecting the components defined in the Technical Detector and, thus, are added to the generalized model. However, in this case, the PDM is only applicable if all properties defined in the PDM's Technical Detector are defined in the exact same way as they are defined in a declarative deployment model. The only exceptions are if (i) the value of a component's or relation's property is empty or (ii) defines a wildcard value, i. e., an asterisks, in the PDM's Technical Detector. Hence, the corresponding value in a declarative deployment model may be (i) any value or must be (ii) any non-empty value.

As a last step, the detected subgraph is removed from the deployment model in line 16 of the algorithm. Then, everything between the lines 3 to 17 is repeated until no more PDMs can be applied.

```
function DETECTPATTERNS(ddm):
  pbdm := CREATECOPY(ddm)
  while (EXISTSAPPLICABLEPDM(pbdm)) do
    pdm := SELECTPDM(pbdm)
    matches := CALCULATEISOMORPHISMS(pbdm, pdm)
    isomorphism := SELECTSUBGRAPH(matches)
    ADDABSTRACTIONFRAGMENT(pbdm, pdm)
    REDIRECTRELATIONS(pbdm, pbdm, isomorphism)
    MOVEPROPERTIES(pbdm, pdm, isomorphism)
    for all (bp ∈ BEHAVIORPATTERNS(pdm)) do
      bpms := BEHAVIORPATTERNMAPPINGS(pdm, bp)
      if (FULFILLSALLPROPERTIES(isomorphism, bpms)
          ∨ (bpms = ∅)) then
        ADDBEHAVIORPATTERN(pbdm, isomorphism, bp)
      end if
    end for
    REMOVESUBGRAPH(pbdm, isomorphism)
  end while
  return pbdm
end function
```

Figure 6. Pseudo-code of the Technical Deployment Model Abstraction.

7 PROTOTYPICAL REALIZATION IN TOSCA

To prove the technical feasibility of our approach, we present a prototypical implementation based on the *Topology and Orchestration Specification for Cloud Applications* (TOSCA) [32, 33] and the open-source TOSCA modeling tool Winery [25].

7.1 The TOSCA Standard

TOSCA is a standardized language to describe the orchestration and management of cloud applications in a vendor- and technology-independent way. To model applications, TOSCA defines *Service Templates*. Thereby, a Service Template contains a description of the application's structure, referred to as its *Topology Template*, which contains (i) *Node Templates* that represent the components of the application and that correspond to *components* in EDMM and (ii) *Relationship Templates* that correspond to *relations* in EDMM. Thus, a Topology Template defines, similarly to EDMM, applications in the form of a directed, weighted graph. In fact, all modeling elements defined by EDMM can be mapped one-on-one to TOSCA [45]. Similar to EDMM, TOSCA defines the semantics of Node Templates and Relationship Templates by *Node Types* and *Relationship Types* respectively. Hence, the Node Types and Relationship Types can be reused when modeling new applications.

7.2 Realizing PbDMs in TOSCA

Since EDMM can be directly mapped one-on-one to TOSCA, we describe in the following how PbDMs can be realized in TOSCA according to previous work [21, 22]. Moreover, since TOSCA models conforming to EDMM are automatically transformable to production ready deployment technologies [45], the approach can be applied to most of these technologies. To model components and relations in TOSCA, Node Templates and Relationship Templates are used respectively. However, to differentiate components from component patterns in TOSCA, the Node Types defining component pattern types are annotated with a flag identifying them as patterns. In contrast, behavior patterns are realized in the form of *Policy Types* since TOSCA allows Node Templates to be annotated with *Policies* that are defined by Policy Types. Additionally, to also enable the annotation of behavior patterns to relations, we extended TOSCA to support the annotation of Policies to Relationship Templates. Since PbDMs are abstract models, the contained patterns must be refined to concrete components to be executable by a TOSCA orchestrator. Hence, the extension is only used during modeling time and does not interfere with the standard-compatibility of executable models [22].

To enable the generalization of deployment models to TOSCA-based PbDMs, we define Pattern Detection Models as a separate model element that uses Topology Templates to define the Technical Detector as well as the Abstraction Structure. Hence, to realize the mappings between the Technical Detector and the Abstraction Structure, separate elements for each mapping type, i. e., relation mappings [21], stay mappings [22], property mappings [42], and behavior pattern mappings, can be defined between the corresponding Node Templates. Moreover, to also provide the description of patterns and how they are linked to each other, we integrated the *Pattern Atlas* [30], which provides a way to capture pattern languages.

Lastly, to enable the graphical modeling and automated generalization of deployment models to PbDMs, we extended Winery [25] to support the modeling of PDMs according to the extensions described above. Thus, we extended the web-based modeling tool Winery, which is part of the OpenTOSCA ecosystem [6], to support the modeling of PDMs. In addition, we implemented the automated generalization of deployment models as described in Section 4.

7.3 Case Study

Based on the deployment model of the motivation scenario shown in Fig. 1, we used our implementation in Winery to automatically identify the abstract semantics hidden in the deployment model by generating a PbDM as illustrated in Fig. 2. We recorded a video [19] that shows (i) how PDMs can be defined in Winery as well as (ii) how they can be used in the generalization process to generate a PbDM from a declarative deployment model automatically.

8 DISCUSSION

The approach described in this paper enables the automated identification of design patterns in technical deployment models and represents the semantics of the architecture in the form of a Pattern-based Deployment Model. However, it is not always possible to detect all patterns realized in a technical deployment model as (i) the repository containing PDMs may not be complete. (ii) Patterns may describe architectural aspects of the application that are not mappable to concrete components, relations, or configurations. (iii) Moreover, there are patterns which require the absence of other patterns or components; this cannot be detected using our subgraph-based approach. Finally, (iv) there are patterns that cannot be detected based on the deployment model but only by using runtime measurements. For example, to determine whether the workload of an application is static, continuously changing, or even unpredictable, the workload can only be measured and cannot be derived from configurations.

As the approach is based on rewriting rules, i. e., Pattern Detection Models, a large number of them are required to detect design patterns in declarative deployment models. Additionally, as outlined in Section 2.3, immense technical expertise is required to create Pattern Detection Models as the abstract concepts described by design patterns need to be mapped to concrete components and their configurations of various technologies and providers. Therefore, experts are required that are familiar with (i) these technologies and how they can be configured as well as (ii) with the appropriate design patterns. As a result, it is a major challenge to create and maintain the large number of PDMs that are required for our approach to abstract as much as possible of the technical information in a declarative deployment model. On the other hand, design patterns are often used in application architectures, which are finally realized technically when the application gets implemented, or in our case, deployed. Thus, the knowledge of refining design patterns to concrete technical details is always required when realizing the patterns. As a result, this knowledge could be documented for deployment-related patterns in the form of PDMs as they exactly define what pattern can be realized by which technical setting and configuration, which provides one opportunity to achieve the number of PDMs required for our approach in practice. Since our approach just applies the PDMs created by experts, the quality and correctness of the resulting PbDMs directly result from the quality and correctness of the PDMs. Thus, if PDMs are created correctly, the detected patterns exactly represent the technical settings that are captured by the PDMs.

9 RELATED WORK

The Software Architecture Reconstruction (SAR) [8] research area aims at reconstructing one or more of the architectural views [24] of an application system using artifacts like source code files, makefiles, and certain designs documents such as UML diagrams. Thus, the presented approach falls under the umbrella of SAR, with the inputs being executable deployment models and the output being Pattern-based Deployment Models. Guamán et al. [17] provide a thorough review on previous SAR approaches. Their review shows that most approaches use source code files that describe the application itself, rather than its infrastructure or deployment, as an input to the reconstruction process. In fact, to the best of our knowledge, no other approach uses deployment models as inputs for the SAR process. Moreover, Guamán et al. [17] found the traceability between a product and its deployment to be an open issue. Our approach contributes to solving this issue since the concepts realized in deployment models can be automatically identified and described using design patterns, which is a first step towards mapping the patterns actually realized and the patterns defined in the application's requirements. Furthermore, although many approaches detect patterns as intermediary or final results, most approaches detect the Gang of Four (GoF) Design Patterns [16] describing the application itself. In contrast, our approach detects component and behavior patterns describing whole application components and their relationships from established pattern languages [1] like the Cloud Computing patterns [15], the Enterprise Integration Patterns [23], as well as the Security Patterns [38] and combines them in PbDMs.

Detecting patterns in deployment models is done in multiple other works: For example, Saatkamp et al. [36, 37] formalize architectural patterns and detect them in deployment models using first-order-logic. In contrast to our approach, they are not identifying patterns realized by the modeled application but identify problems existing in the deployment model that are solved by the formalized patterns. Afterwards, a user can choose a preferred pattern, which solves the detected problem by automatically applying it to the model [35]. Similarly, Borovits et al. [5] and Kumara et al. [28] propose an approach to detect code smells and anti-patterns in IaC-models.

To describe cloud application abstractly, Di Martino et al. [9] also use the Cloud Computing patterns [15] to model their component's composition and map them to provider specific patterns. However, this is a manual process and their approach does not detect patterns automatically in an existing deployment model. Weigold et al. [41] introduce a concept to link patterns of different pattern languages in "views" but they are not used to describe applications.

Similar to our approach, which uses subgraph isomorphism to identify applicable PDMs, Krieger et at. [26] search for required subgraphs in a deployment model to detect whether the deployment model conforms to required compliance rules. Similar approaches are presented by Zimmermann et al. [48], Eilam et al. [10], and Arnold et al. [2, 3] who identify special structures in deployment models. However, none of the approaches is detecting design patterns that are realized in the modeled applications.

Similar to the concept of *Solution Implementations* [12, 13], the refinement of patterns to sub-patterns [18], and *Architectural Templates* [29], PDMs define how patterns are realized by concrete technologies, but can be used in both directions.

10 CONCLUSION AND FUTURE WORK

Deployment models describe an application and its components usually in a very detailed and technology- as well as vendor-specific way. Thus, to identify the realized patterns and semantics is difficult and requires immense technical expertise. Therefore, we presented an approach that is able to detect design patterns in declarative deployment models and thereby eases the understanding of the semantics hidden in such technically detailed models. Thus, instead of having to know all technical variations of each vendor or technology, only abstract design patterns need to be known to understand the semantics of an application. Moreover, by combining the approach with our instance retrieval approach [20], we are able to detect design patterns in running applications that have been deployed using production-ready technologies like Puppet or Kubernetes. However, one limitation of the approach is that the detection and rewriting rules, i. e., PDMs, must be defined by experts once manually.

In future work, we plan to extend the approach by a traceability method to enable users to understand how a specific pattern was detected. Additionally, we plan to apply compliance approaches to ensure that the application is compliant to its specification, for example, by identifying that the application follows the Cloud Data Patterns for confidentiality [40]. Moreover, we want to extend our approach to support different levels of pattern abstraction [14].

ACKNOWLEDGMENT

This work was partially funded by the German Research Foundation (DFG) projects SustainLife (379522012) and IAC^2 (314720630), as well as by the BMWi project *PlanQK* (01MK20005N).

REFERENCES

[1] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. 1977. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press.

[2] William Arnold, Tamar Eilam, Michael Kalantar, Alexander V. Konstantinou, and Alexander A. Totok. 2007. Pattern Based SOA Deployment. In *ICSOC 2007*. Springer, 1–12.

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

[4] Alexander Bergmayr, Uwe Breitenbücher, Nicolas Ferry, Alessandro Rossini, Arnor Solberg, Manuel Wimmer, Gerti Kappel, and Frank Leymann. 2018. A Systematic Review of Cloud Modeling Languages. *ACM Computing Surveys (CSUR)* 51, 1 (Feb. 2018), 1–38.

[5] Nemania Borovits, Indika Kumara, Parvathy Krishnan, Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, Damian A. Tamburri, and Willem-Jan van den Heuvel. 2020. DeepIaC: Deep Learning-Based Linguistic Anti-Pattern Detection in IaC *(MaLTeSQuE 2020)*. Association for Computing Machinery, NY, USA, 7–12.

[6] Uwe Breitenbücher, Christian Endres, Kálmán Képes, Oliver Kopp, Frank Leymann, Sebastian Wagner, Johannes Wettinger, and Michael Zimmermann. 2016. The OpenTOSCA Ecosystem - Concepts & Tools. *European Space project on Smart Systems, Big Data, Future Internet - Towards Serving the Grand Societal Challenges - Volume 1: EPS Rome 2016* (Dec. 2016), 112–130.

[7] Uwe Breitenbücher, Kálmán Képes, Frank Leymann, and Michael Wurster. 2017. Declarative vs. Imperative: How to Model the Automated Deployment of IoT Applications?. In *Proceedings of the 11th Advanced Summer School on Service Oriented Computing*. IBM Research Division, 18–27.

[8] S. Jeromy Carrière and Rick Kazman. 1998. The perils of reconstructing architectures. In *ISAW '98*. ACM Press, New York, New York, USA, 13–16.

[9] Beniamino Di Martino, Giuseppina Cretella, and Antonio Esposito. 2017. Cloud services composition through cloud patterns: a semantic-based approach. *Soft Computing* 21, 16 (2017), 4557–4570.

[10] T. Eilam, M.H. Kalantar, A.V. Konstantinou, G. Pacifici, J. Pershing, and A. Agrawal. 2006. Managing the configuration complexity of distributed applications in Internet data centers. *Communications Magazine* 44, 3 (March 2006), 166–177.

[11] Christian Endres, Uwe Breitenbücher, Michael Falkenthal, Oliver Kopp, Frank Leymann, and Johannes Wettinger. 2017. Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications. In *PATTERNS 2017*. Xpert Publishing Services, 22–27.

[12] Michael Falkenthal, Johanna Barzen, Uwe Breitenbücher, Christoph Fehling, and Frank Leymann. 2014. Efficient Pattern Application: Validating the Concept of Solution Implementations in Different Domains. *International Journal On Advances in Software* 7, 3&4 (Dec. 2014), 710–726.

[13] Michael Falkenthal, Johanna Barzen, Uwe Breitenbücher, Christoph Fehling, and Frank Leymann. 2014. From Pattern Languages to Solution Implementations. In *PATTERNS 2014*. Xpert Publishing Services, 12–21.

[14] Michael Falkenthal, Johanna Barzen, Uwe Breitenbücher, Christoph Fehling, Frank Leymann, Aristotelis Hadjakos, Frank Hentschel, and Heizo Schulze. 2015. Leveraging Pattern Application via Pattern Refinement. In *PURPLSOC 2015*. epubli.

[15] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. 2014. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer. 367 pages.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley.

[17] Daniel Guamán, Jennifer Pérez, Jessica Diaz, and Carlos E. Cuesta. 2020. Towards a reference process for software architecture reconstruction. *IET Software* 14, 6 (dec 2020), 592–606.

[18] Jason Hallstrom and Neelam Soundarajan. 2009. Reusing patterns through design refinement. In *International Conference on Software Reuse*. Springer, 225–235.

[19] Lukas Harzenetter. 2021. *Video: Automated Detection of Design Patterns in Declarative Deployment Models*. https://vimeo.com/543231040 2021-04-30.

[20] Lukas Harzenetter, Tobias Binz, Uwe Breitenbücher, Frank Leymann, and Michael Wurster. 2021. Automated Generation of Management Workflows for Running Applications by Deriving and Enriching Instance Models. In *CLOSER 2021*. SciTePress, 99–110.

[21] Lukas Harzenetter, Uwe Breitenbücher, Michael Falkenthal, Jasmin Guth, Christoph Krieger, and Frank Leymann. 2018. Pattern-based Deployment Models and Their Automatic Execution. In *UCC 2018*. IEEE, 41–52.

[22] Lukas Harzenetter, Uwe Breitenbücher, Michael Falkenthal, Jasmin Guth, and Frank Leymann. 2020. Pattern-based Deployment Models Revisited: Automated Pattern-driven Deployment Configuration. In *PATTERNS 2020*. Xpert Publishing Services, 40–49.

[23] Gregor Hohpe and Bobby Woolf. 2004. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley.

[24] Rick Kazman and S. Jeromy Carriere. 1998. View extraction and view fusion in architectural understanding. In *International Conference on Software Reuse*.

[25] Oliver Kopp, Tobias Binz, Uwe Breitenbücher, and Frank Leymann. 2013. Winery – A Modeling Tool for TOSCA-based Cloud Applications. In *ICSOC 2013*. Springer, 700–704.

[26] Christoph Krieger, Uwe Breitenbücher, Kálmán Képes, and Frank Leymann. 2018. An Approach to Automatically Check the Compliance of Declarative Deployment Models. In *SummerSoC 2018*. IBM Research Division, 76–89.

[27] Philippe B Kruchten. 1995. The 4+ 1 view model of architecture. *IEEE software* 12, 6 (1995), 42–50.

[28] Indika Kumara, Zoe Vasileiou, Georgios Meditskos, Damian A Tamburri, Willem-Jan Van Den Heuvel, Anastasios Karakostas, Stefanos Vrochidis, and Ioannis Kompatsiaris. 2020. Towards Semantic Detection of Smells in Cloud Infrastructure Code. In *Proceedings of the 10th International Conference on Web Intelligence, Mining and Semantics*. 63–67.

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

[30] Frank Leymann and Johanna Barzen. 2021. Pattern Atlas. *Lecture Notes in Computer Science* 12521 (April 2021), 67–76.

[31] Kief Morris. 2016. *Infrastructure as code: managing servers in the cloud*.

[32] OASIS. 2013. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0*. Organization for the Advancement of Structured Information Standards (OASIS).

[33] OASIS. 2020. *TOSCA Simple Profile in YAML Version 1.3*. Organization for the Advancement of Structured Information Standards (OASIS).

[34] David Oppenheimer. 2003. The importance of understanding distributed system configuration. In *CHI 2003*. ACM.

[35] Karoline Saatkamp, Uwe Breitenbücher, Michael Falkenthal, Lukas Harzenetter, and Frank Leymann. 2019. An Approach to Determine & Apply Solutions to Solve Detected Problems in Restructured Deployment Models Using First-Order Logic. In *CLOSER 2019*. SciTePress, 495–506.

[36] Karoline Saatkamp, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. 2018. Application Scenarios for Automated Problem Detection in TOSCA Topologies by Formalized Patterns. In *SummerSOC 2018*. IBM Research Division, 43–53.

[37] Karoline Saatkamp, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. 2019. An approach to automatically detect problems in restructured deployment models based on formalizing architecture and design patterns. *SICS Software-Intensive Cyber-Physical Systems* (Feb. 2019), 1–13.

[38] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. 2006. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons, Inc. 565 pages.

[39] Andy Schürr. 1995. Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science*. Springer, 151–163.

[40] Steve Strauch, Uwe Breitenbücher, Oliver Kopp, Frank Leymann, and Tobias Unger. 2012. Cloud Data Patterns for Confidentiality. In *CLOSER 2012*. SciTePress, 387–394.

[41] Manuela Weigold, Johanna Barzen, Uwe Breitenbücher, Michael Falkenthal, Frank Leymann, and Karoline Wild. 2020. Pattern Views: Concept and Tooling of Interconnected Pattern Languages. In *SummerSOC 2020*. Springer International Publishing, 86–103.

[42] Karoline Wild, Uwe Breitenbücher, Lukas Harzenetter, Frank Leymann, Daniel Vietz, and Michael Zimmermann. 2020. TOSCA4QC: Two Modeling Styles for TOSCA to Automate the Deployment and Orchestration of Quantum Applications. In *EDOC 2020*. IEEE, 125–134.

[43] Michael Wurster, Uwe Breitenbücher, Antonio Brogi, Ghareeb Falazi, Lukas Harzenetter, Frank Leymann, Jacopo Soldani, and Vladimir Yussupov. 2019. The EDMM Modeling and Transformation System. In *Service-Oriented Computing – ICSOC 2019 Workshops*. Springer.

[44] Michael Wurster, Uwe Breitenbücher, Michael Falkenthal, Christoph Krieger, Frank Leymann, Karoline Saatkamp, and Jacopo Soldani. 2019. The Essential Deployment Metamodel: A Systematic Review of Deployment Automation Technologies. *Software-Intensive Cyber-Physical Systems* 35 (Aug. 2019), 63–75.

[45] Michael Wurster, Uwe Breitenbücher, Lukas Harzenetter, Frank Leymann, Jacopo Soldani, and Vladimir Yussupov. 2020. TOSCA Light: Bridging the Gap between the TOSCA Specification and Production-ready Deployment Technologies. In *CLOSER 2020*. SciTePress, 216–226.

[46] Vladimir Yussupov, Jacopo Soldani, Uwe Breitenbücher, Antonio Brogi, and Frank Leymann. [n.d.]. From Serverful to Serverless: A Spectrum of Patterns for Hosting Application Components. In *CLOSER 2021*. SciTePress. to appear.

[47] Vladimir Yussupov, Jacopo Soldani, Uwe Breitenbücher, Antonio Brogi, and Frank Leymann. 2021. FaaSten your decisions: A classification framework and technology review of Function-as-a-Service platforms. *Journal of Systems and Software* 175 (May 2021).

[48] Michael Zimmermann, Uwe Breitenbücher, Christoph Krieger, and Frank Leymann. 2018. Deployment Enforcement Rules for TOSCA-based Applications. In *SECURWARE 2018*. Xpert Publishing Services, 114–121.