Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Pattern Detection in Declarative Deployment Models

Adrian Wersching

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	Lukas Harzenetter, M. Sc.
Commenced:	October 28, 2020
Completed:	April 28, 2021

Abstract

The manual deployment of cloud applications is error-prone, time-consuming, and requires considerable technical knowledge. To tackle these issues and automate the deployment of applications, multiple approaches have been proposed that enable the description of applications in the form of declarative deployment models which describe the components of an application, their desired state or configuration, and the relations among the components. The technologies and tools that support deployment models significantly ease the deployment process, however, they still require vendor and product-specific details in the deployment models. This obfuscates the underlying semantics of the deployment models. The essential architectural decisions realized in a deployment model can be stated more clearly by design patterns which describe problems, their solution, the resulting benefits, and the resulting drawbacks in an abstract and reusable format. To combine the benefits of declarative deployment models and patterns, Pattern-based Deployment and Configuration Models (PbDCMs) were defined which introduce patterns as first-class citizens in deployment models. However, manually extracting the realized patterns from deployment models of applications is a non-trivial task as it requires technical knowledge about the patterns a certain component or parts of the application realize. Furthermore, there is no automated way to use the detected patterns for the creation of PbDCMs. To tackle these issues, this thesis presents an automated approach for the detection of patterns in declarative deployment models and the generation of corresponding PbDCMs. The automated detection of patterns is enabled by introducing Pattern Detection and Refinement Models (PDRMs) which consist of two structures, one is used to determine matching subgraphs in a deployment model and the other represents the patterns realized by the first structure. The patterns detected with the support of the introduced PDRMs are used to build corresponding PbDCMs. The approach is implemented as an extension to the modeling tool Eclipse Winery and is validated by a case study. The case study highlights how the pattern detection process can be used to realize new design decisions on an abstract level.

Kurzfassung

Das manuelle Deployment von Anwendungen in der Cloud ist fehleranfällig, zeitaufwändig und erfordert erhebliche technisches Vorkenntnisse. Um diesen Probleme entgegenzuwirken und das Deployment von Anwendungen zu automatisieren, wurden unterschiedliche Ansätze eingeführt, welche die Beschreibung von Anwendungen in Form von deklarativen Deployment Modellen ermöglichen. Deklarative Deployment Modelle beschreiben dabei die Komponenten einer Anwendung, deren gewünschten Zustand beziehungsweise gewünschte Konfiguration und die Beziehungen der Komponenten zueinander. Die Technologien und Werkzeuge, welche Deployment Modelle unterstützen, erleichtern zwar den Prozess des Deployments erheblich, setzen aber dennoch hersteller- und produktspezifische Details in den Deployment Modellen voraus. Dadurch wird die zugrunde liegende Semantik von Deployment Modellen verschleiert. Die wesentlichen architektonischen Entscheidungen, die in einem Deployment Modell umgesetzt sind, können klarer durch Architekturmuster beschrieben werden. Architekturmuster beschreiben Probleme, deren Lösungen, die resultierenden Vorteile und die resultierenden Nachteile in einem abstrakten und wiederverwendbaren Format. Um die Vorteile von deklarativen Deployment Modellen und Architekturmustern zu vereinen, wurden Pattern-based Deployment and Configuration Models (PbDCMs) eingeführt, welche das Modellieren von Architekturmustern in Deployment Modellen erlauben. Das manuelle Bestimmen der Architekturmuster, welche in einem Deployment Modell umgesetzt sind, ist jedoch keine triviale Aufgabe, da es technische Vorkenntnisse über die Architekturmuster voraussetzt, die durch die jeweiligen Komponenten oder Teile der Anwendung umgesetzt werden. Es existiert weiterhin keine automatisierte Methode, die identifizierten Architekturmuster für das Erstellen von PbDCMs zu verwenden. Um diese Probleme zu lösen, wird in dieser Arbeit ein automatisierter Ansatz zur Erkennung von Architekturmustern in deklarativen Deployment Modellen vorgestellt, welcher auch die Generierung von entsprechenden PbDCMs umfasst. Die automatische Erkennung von Architekturmustern wird durch das Einführen von Pattern Detection and Refinement Models (PDRMs) ermöglicht, welche aus zwei Structuren bestehen, von denen eine für die Bestimmung von übereinstimmenden Teilgraphen in einem Deployment Modell verwendet wird und die andere die durch die erste Struktur umgesetzten Architekturmuster darstellt. Die mit Hilfe der eingeführten PDRMs erkannten Architekturmuster werden verwendet, um entsprechende PbDCMs zu erstellen. Der Ansatz ist als eine Erweiterung des Modellierungswerkzeugs Eclipse Winery implementiert und wird anhand einer Fallstudie validiert. Die Fallstudie verdeutlicht, wie der Mustererkennungsprozess genutzt werden kann, um neue Designentscheidungen auf einer abstrakten Ebene zu realisieren.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

API Application Programming Interface. 56

AST Abstract Syntax Tree. 71

AWS Amazon Web Services. 15

BLOB Binary Large Object. 65

CBPRM Component and Behavior Pattern Refinement Model. 7, 9, 16

CSAR Cloud Service Archive. 57

DBMS Database Management System. 18

DSL Domain-specific Language. 70

EDMM Essential Deployment Metamodel. 7, 21

FIFO First-In, First-Out. 16

GUI Graphical User Interface. 18

IaC Infrastructure as Code. 72

ID Identifier. 60

KMS Key Management System. 44, 66

OASIS Organization for the Advancement of Structured Information Standards. 55

OCCI Open Cloud Computing Interface. 72

OMT Object-Modeling Technique. 71

PbDCM Pattern-based Deployment and Configuration Model. 3, 4, 7, 16

PDRM Pattern Detection and Refinement Model. 3, 4, 7, 9, 16

REST Representational State Transfer. 56

SCA Service Component Architecture. 72

SDK Software Development Kit. 73

SIG Softgoal Interdependency Graph. 69

SPARQL SPARQL Protocol and RDF Query Language. 70

SQL Structured Query Language. 23

SQS Simple Queue Service. 16

SSE Server-Side Encryption. 18

SWRL Semantic Web Rule Language. 72

TFRM Topology Fragment Refinement Model. 73

TOSCA Topology and Orchestration Specification for Cloud Applications. 15

UI User Interface. 56

UML Unified Modeling Language. 19

VM Virtual Machine. 15

XML Extensible Markup Language. 72

1 Introduction

The manual deployment of cloud applications is error-prone, time-consuming, and requires considerable technical knowledge [BBK+14a; OGP03]. To tackle these issues and automate the deployment of applications, multiple approaches have been proposed that enable the description of applications in the form of deployment models [WBF+19]. There are provider-specific deployment automation technologies such as Amazon Web Services (AWS) CloudFormation[1], configuration management systems such as Puppet[2], and interoperable standards such as the Topology and Orchestration Specification for Cloud Applications (TOSCA) [OAS13; OAS20]. Most of the approaches use declarative deployment models which describe the components of an application, their desired state or configuration, and the relations among the components [EBF+17; HAW11; WBF+19]. For example, the declarative deployment model of an application for digitalizing documents may consist of an Angular[3] frontend which provides documents selected by a user to a Java[4] backend that digitalizes the documents and stores the result in a MySQL[5] database [FLR+14]. All three components may be hosted on an OpenStack[6] Virtual Machine (VM).

While technologies and tools that support deployment models significantly ease the deployment process, they still require vendor and product-specific details in the deployment models [HBF+20; HBM+18]. This obfuscates the underlying semantics of the deployment models [Feh15]. Furthermore, this obfuscation complicates the comparison of applications according to their functional and non-functional properties [Feh15]. For example, a deployment model created using the service stack of AWS[7] differs from a deployment model created using a Microsoft Azure[8] based stack in terms of used components, level of abstraction, and terminology. To capture essential architectural decisions including their resulting benefits and drawbacks independent of specific vendors, technologies, and products, different pattern languages have been proposed [FLR+14; HW04; SFH+06]. The patterns of a pattern language describe abstract and proven solutions for recurring problems in a specific domain [FLR+14; HW04]. For example, by using the Cloud Computing Patterns [FLR+14], the architectural decision behind the MySQL database of the described document processing application can be stated more clearly by the *Relational Database* [FLR+14] pattern to specify that it stores data and maintains its relations. Furthermore, as patterns like the *Relational Database* pattern define their characteristics, they can be used for a better understanding of existing applications and to guide their future development [FLR+14].

[1] https://aws.amazon.com/cloudformation/
[2] https://puppet.com/
[3] https://angular.io/
[4] https://www.java.com/
[5] https://www.mysql.com/
[6] https://www.openstack.org/
[7] https://aws.amazon.com/
[8] https://azure.microsoft.com/

However, the patterns realized by a component or part of a deployment model are not always as apparent as the *Relational Database* pattern realized by the MySQL database of the described document processor. For example, the communication between the components of the document processor could be implemented using AWS Simple Queue Service (SQS)[9] queues [FLR+14]. AWS SQS queues of type First-In, First-Out (FIFO) guarantee that each message is delivered once without duplicates [Ama21] and, therefore, realize the *Exactly-once Delivery* [FLR+14] pattern. AWS SQS queues of type standard only guarantee that each message is delivered, i.e., duplicates may occur [Ama21], and, therefore, realize the *At-least-once Delivery* [FLR+14] pattern. This difference in configuration does not convey the semantics of the queues but requires expert knowledge to understand. As one component can also realize multiple patterns and a deployment model can consist of several components and their relations [HBF+20; HBM+18], further complexity is introduced into the process of determining the abstract semantics of an application. These problems are amplified even more if the application or the deployment model needs to be changed frequently which may be the case during the initial development phase or to provide security updates. To eliminate the described challenges and reveal the underlying architectural concepts and semantics of deployment models, an automated approach for the detection of patterns is needed.

If patterns can be detected automatically in deployment models, then the detected patterns can also be used to build pattern-based representations of the deployment models. This can be achieved by utilizing Pattern-based Deployment and Configuration Models (PbDCMs) [HBF+20; HBM+18] which introduce patterns as first-class citizens in deployment models. For example, a PbDCM describing the fundamental design decisions of the aforementioned document processing application would contain the *Relational Database* pattern to abstractly define that the application must use a relational database to store its data. PbDCMs can therefore provide a comprehensive view of all patterns realized by an application and its deployment model. Furthermore, as the patterns contained in PbDCMs are independent of specific vendors and technologies, fundamental design decisions can be changed on an abstract level without considering the technical details required to realize these changes [HBF+20; HBM+18]. For example, the *Relational Database* pattern may be annotated with the *Information Obscurity* [SFH+06] pattern to specify that the stored data needs to be protected, e.g., by encryption. To execute an adapted PbDCM, i.e., obtain an executable deployment model which realizes the patterns defined by the PbDCM, all abstract patterns must be refined to concrete vendors and technologies which can be achieved by using the automated pattern refinement process introduced for PbDCMs [HBF+20; HBM+18]. However, while PbDCMs can be refined to executable deployment models in an automated fashion, there is no automated way to represent the patterns identified in an executable deployment model in the form of a PbDCM.

To tackle the described issues, this thesis presents an automated approach for the detection of patterns in declarative deployment models and the generation of corresponding PbDCMs. The automated detection of patterns is enabled by introducing Pattern Detection and Refinement Models (PDRMs) which are based on the Component and Behavior Pattern Refinement Models (CBPRMs) [HBF+20; HBM+18] defined for the pattern refinement process and are suitable to both purposes. A PDRM consists of two structures, one is used to determine matching subgraphs in a deployment model and the other represents the patterns realized by the first structure. The patterns detected with the support of the introduced PDRMs are used to build corresponding PbDCMs which provide a comprehensive view of all identified patterns and enable the application of the pattern refinement

[9] https://aws.amazon.com/sqs/

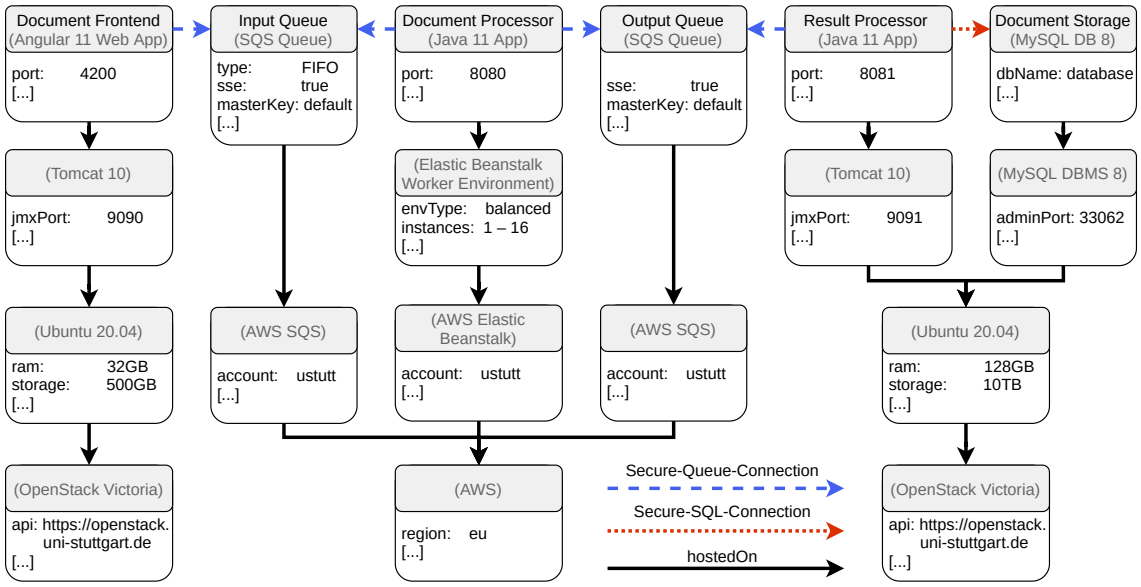

Figure 1.1: Declarative deployment model of a document processor.

process. A PbDCM for a deployment model is built by repurposing and extending the algorithms described for the refinement of patterns. The approach is implemented as an extension to the modeling tool Eclipse Winery [KBBL13] and is validated by a case study. The case study highlights how the pattern detection process can be used in combination with the pattern refinement process to realize new design decisions on an abstract level.

1.1 Running Example

In this section, the previously mentioned document processing application is extended to a running example which will be used throughout this thesis. The running example is inspired by the *Hybrid Processing* [FLR+14] cloud application design. Figure 1.1 shows the declarative deployment model of the application which was specifically developed with the digitalization of sensitive documents in mind. The components of the deployment model are visualized as boxes and the relations between the components as directed arrows. A component can have a name which is visualized at the top of the respective box and must have a type which is visualized by the respective text in brackets. The properties of a component which describe its desired state or configuration are visualized as key-value pairs below the type. The type of a relation is defined by the visual representation of the respective arrow as shown at the bottom of Figure 1.1 [BBK+12].

Because of the implementation and the intended usage of the document processor, two requirements must be fulfilled by the deployment: (i) the component implementing the digitalization procedure must be scaled automatically to adapt to different kinds of utilization and (ii) digitalized documents must be stored on-premise due to data protection regulations. To fulfill these requirements, the deployment model depicted in Figure 1.1 utilizes three cloud environments. The AWS cloud shown at the center of Figure 1.1 is used to adapt to different kinds of utilization as it offers managed services. A local OpenStack Victoria infrastructure is used to host a database which stores the

digitalized documents as depicted on the right side of Figure 1.1. A second local OpenStack Victoria infrastructure is used to host the Graphical User Interface (GUI) of the application as depicted on the left side of Figure 1.1. The AWS cloud environment and both local cloud environments provide the respective component stacks required to realize the deployment [FLR+14].

To digitalize a document, it must be selected and uploaded using the *Document Frontend* component which is depicted at the top left corner of Figure 1.1 and is implemented as an Angular 11 Web App. The *Document Frontend* component is executed by a Tomcat[10] 10 webserver running on an Ubuntu[11] 20.04 VM which is configured with fixed values for its RAM and storage and is hosted on one of the local OpenStack Victoria infrastructures.

After a document has been selected for digitalization by using the *Document Frontend* component, it is processed by the *Document Processor* component depicted at the center of Figure 1.1. To improve scalability and decouple the local cloud environment from the AWS cloud environment, the selected document is not sent to the *Document Processor* component directly but placed in a queue [FLR+14]. The result of the digitalization process is then also placed in a queue to maintain the described benefits [FLR+14]. Both the *Input Queue* component and the *Output Queue* component contained in the deployment model are implemented as SQS Queues hosted on components representing the AWS SQS service. The components representing the AWS SQS service are in turn hosted on a component representing the AWS cloud environment. Furthermore, the *Input Queue* component and the *Output Queue* component specify properties to describe their configuration. The *Input Queue* component is configured as an SQS Queue of type FIFO and uses Server-Side Encryption (SSE) with a default master key. The *Output Queue* does not specify a type for the SQS Queue but uses SSE with a default master key as well. The *Document Processor* component itself is implemented as a Java 11 App and is executed in an Elastic Beanstalk[12] Worker Environment hosted on the AWS Elastic Beanstalk service. The components representing the Elastic Beanstalk Worker Environment and the AWS Elastic Beanstalk service are also hosted on the AWS cloud component. The Elastic Beanstalk Worker Environment is configured to be balanced and specifies a minimum and a maximum number of instances to scale the *Document Processor* component in or out.

After a document has been processed by the *Document Processor* component and has been placed in the SQS Queue of the *Output Queue* component, the processed document is retrieved by the *Result Processor* component which is depicted on the right side of Figure 1.1. The *Result Processor* component writes the final result of the digitalization process to the database implemented by the *Document Storage* component. The *Result Processor* component is implemented as a Java 11 App and is hosted on another Tomcat 10 webserver. The *Document Storage* component is implemented as a MySQL DB 8 database and is hosted on a MySQL Database Management System (DBMS) 8. Both the Tomcat 10 webserver and the MySQL DBMS 8 are hosted on an Ubuntu 20.04 VM which is again configured with fixed values for its RAM and storage and is hosted on the second local OpenStack Victoria infrastructure. Finally, the communication between all components is represented by relations that guarantee security of some sort for the exchanged data.

[10] https://tomcat.apache.org/
[11] https://ubuntu.com/
[12] https://aws.amazon.com/elasticbeanstalk/

The discussed deployment model of the document processor potentially realizes a multitude of different patterns, each of which requires careful analysis of the components, their configuration, the relations among the components, and the context of the application. It should also be noted that the deployment model depicted in Figure 1.1 omits some specifics that may be required for its successful execution as indicated by the dots in square brackets in the property description of the components. For example, the different AWS services require authentication to be able to access the specified account which is not provided in the deployment model. The presence of characteristics not directly related to a pattern further complicates the manual detection of patterns. Additionally, changes to the deployment model require re-evaluation of the identified patterns and reanalysis of the deployment model. Therefore, an automated approach to detect patterns in deployment models is needed to enable more reproducible results, reduce errors, reduce the needed effort to be made, and reduce the required technical knowledge.

1.2 Outline

The remainder of this thesis is divided into the following chapters:

Chapter 2 – Foundations
In this chapter, the foundations required for the remainder of this thesis are described. This includes background information on deployment automation, patterns and pattern languages, PbDCMs, and the basics of the pattern refinement process which generates executable deployment models from PbDCMs.

Chapter 3 – Pattern Detection in Declarative Deployment Models
This chapter presents an automated approach for the detection of patterns in declarative deployment models which consists of two detection phases connected in series. The first detection phase identifies structural as well as behavioral patterns. The second detection phase identifies additional structural patterns. Both phases utilize the detected patterns to build a PbDCM for a given deployment model.

Chapter 4 – Prototypical Implementation
In this chapter, the prototypical implementation of the presented approach is described. This includes additional background information on TOSCA and Eclipse Winery which were used for the implementation of the approach.

Chapter 5 – Validation
In this chapter, a case study is conducted to validate the presented approach. Furthermore, limitations of the approach are discussed.

Chapter 6 – Related Work
Related work in different domains is discussed. This includes general pattern detection in deployment models, Unified Modeling Language (UML) models, and application code. Additionally, pattern detection approaches specific to a domain and works in a wider scope are discussed.

Chapter 7 – Conclusion and Future Work
This chapter provides a conclusion of this thesis and envisions future work.

2 Foundations

In this chapter the foundations required for the remainder of this thesis are described. Section 2.1 provides an introduction to deployment automation and describes the Essential Deployment Metamodel (EDMM) [WBF+19] which defines the fundamental elements of declarative deployment models. In Section 2.2, the notion of patterns and pattern language is introduced and the pattern languages used throughout this thesis are outlined. Section 2.3 describes the details of PbDCMs [HBF+20; HBM+18] including a formal metamodel based on EDMM. Additionally, the pattern refinement process for PbDCMs [HBF+20; HBM+18] which is repurposed and extended in this thesis to enable the detection of patterns in declarative deployment models is outlined.

2.1 Deployment Automation

A number of deployment automation approaches have been proposed to increase the reusability of deployments, reduce errors, and ease the process of automation [WBF+19]. For example, there are provider-specific deployment automation technologies such as AWS CloudFormation or Azure Resource Manager[1]. Other deployment automation technologies such as Puppet, Chef[2], or Ansible[3] enable configuration management without the restriction to a single provider or platform. There are also standards such as the Topology and Orchestration Specification for Cloud Applications (TOSCA) [OAS13; OAS20] which provides a metamodel for deployment models. The models used by such deployment automation technologies and standards can be imperative or declarative [EBF+17]. Imperative deployment models explicitly state all technical steps required for the deployment, the order these steps need to be executed in, and the data flow between the individual steps. As all of the technical details required for the deployment can be configured, imperative deployment models can be used for deployments that require individual customization. Declarative deployment models describe the structure of the application to be deployed and are interpreted by a suitable deployment and management system which executes the required technical activities for the deployment. Therefore, declarative deployment models can be used for applications which require less individual customization in their deployments. The focus of this thesis resides on declarative deployment models as they can be transformed to imperative deployment models and are considered to be the most appropriate approach for application deployment and configuration management [BBK+14a; HAW11; WBF+19].

While deployment models can be categorized into the two types of imperative deployment models and declarative deployment models, the multitude of deployment automation technologies using these models complicates the comparison of different technologies and increases the effort needed

[1] https://azure.microsoft.com/en-us/features/resource-manager/
[2] https://www.chef.io/products/chef-infra/
[3] https://www.ansible.com/

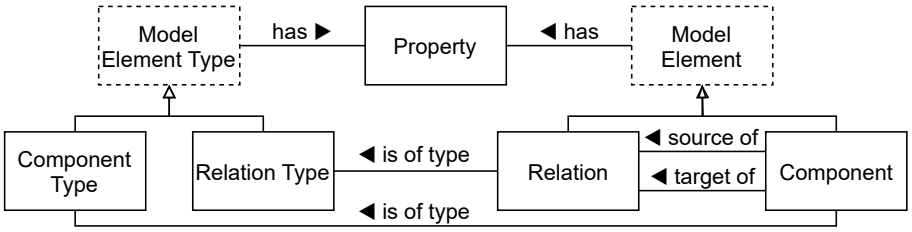

Figure 2.1: Relevant elements of EDMM based on [WBF+19].

for migration from one technology to another [WBF+19]. To describe declarative deployment models independent of specific deployment automation technologies, Wurster et al. [WBF+19] introduce EDMM which has been obtained through a systematic analysis of the most used declarative deployment technologies and can be mapped to the different approaches it is based on. Figure 2.1 depicts the elements of EDMM relevant in the context of this thesis. In Figure 2.1, inheritance is visualized by arrows with outlined heads and abstract elements are visualized by a dashed border.

EDMM defines *Components* and *Relations* as the fundamental structural elements of a declarative deployment model as depicted on the right side of Figure 2.1. A Component is a physical, functional, or logical unit of the respective application and can be the source or the target of a Relation. A Relation is a directed physical, functional, or logical dependency between exactly two Components. For example, the Java 11 App *Result Processor* and the MySQL DB 8 *Document Storage* depicted in Figure 1.1 represent Components in EDMM and the *Secure-SQL-Connection* between the two represents a Relation in EDMM. Components and Relations can have *Properties* which describe their state or configuration. For example, the MySQL DB 8 *Document Storage* depicted in Figure 1.1 specifies a Property defining the name of the MySQL database. Components and Relations are instances of a *Component Type* or a *Relation Type*, respectively, which specify the semantics of their instances in a reusable format as depicted on the left side of Figure 2.1. This includes the definition of the Properties a Component or Relation can have. For example, a Component Type for the MySQL DB 8 *Document Storage* depicted in Figure 1.1 could specify that the Property defining the name of the database is of type string [WBF+19].

2.2 Patterns and Pattern Languages

While the concept of patterns and pattern languages originates from the works of Alexander et al. [AIS77] and Alexander [Ale79] in the domain of architecture, common patterns and their interrelations have since been defined for a multitude of other domains such as cloud computing, messaging, software security, and humanities [FLR+14; HW04; SBLE12; SFH+06]. Generally, patterns describe proven solutions for recurring problems in a specific domain [FLR+14]. The problem a pattern refers to and the solution it describes are usually defined in an abstract way to increase its applicability [FLR+14; HW04]. Furthermore, the definition of patterns often follows a common format [FLR+14; HW04]. For example, the format used to define the Cloud Computing Patterns [FLR+14] consists of a *name* for the pattern, an *intent* providing a short description, a *driving question* stating the problem to be solved, an *icon*, a *context* the described problem can

Name	*Relational Database*
Intent	Structure data according to a schema and enable expressive queries.
Driving Question	How to express relations in stored data? How to enable expressive queries?
Icon	
Context	Many similar data elements, consistent relations in data are expected.
Solution	Data is stored in tables where each row represents a data element and the columns represent the well-defined attributes of the data elements.
Result	(i) Key attributes for identifying elements, (ii) foreign keys to specify relations between elements, (iii) consistency is enforced during data manipulation, and (iv) expressive queries, e.g., by using the Structured Query Language (SQL).
Variations	–
Related Patterns	*Key-Value Storage*, *Stateless Component*, ...
Known Uses	Can be used via a DBMS or as a service.

Table 2.1: Description of the *Relational Database* pattern [FLR+14].

appear in, a *solution* to the problem, a *result* describing the solution in detail, possible *variations* of the pattern, other *related patterns*, and *known uses* of the pattern. This format is used in Table 2.1 to describe the *Relational Database* [FLR+14] pattern.

The *Relational Database* pattern can be applied if many similar data elements need to be structured according to a predefined schema which enforces the consistency of the data and enables expressive queries. To realize the *Relational Database* pattern, the data is stored in tables where each row of a table represents a data element and the columns of a table represent the well-defined attributes of the data elements. The consistency of the data can be enforced based on key attributes which identify a data element and foreign keys which express relations between data elements. Furthermore, expressive queries are enabled by using a query language such as SQL. Instances of the *Relational Database* pattern can be used via a DBMS, as it is the case in the running example depicted in Figure 1.1, or as a service, e.g., by using the AWS Relational Database Service[4] [FLR+14].

Patterns are usually not defined as isolated solutions but are part of a pattern language [AIS77]. Pattern languages define the relations between the patterns in a domain and guide the navigation through the knowledge base created by the different patterns [FL17]. Relations between patterns can be specified, for example, by using the *related patterns* reference defined for the Cloud Computing Patterns [FLR+14]. The relations defined by such references can exhibit different semantics and therefore enable different decision-making processes [FL17]. For example, there are relations which define AND semantics, OR semantics, or XOR semantics and can therefore be used to express that two patterns are usually used in combination, that one pattern represents an alternative to another pattern, or that one pattern can only be used if another pattern is excluded [Rei14]. Other relations define the increase or decrease of specific quality attributes if a pattern is applied based on the pattern which is the source of the relation [Zdu07]. In the Cloud Computing Patterns, the semantics

[4] https://aws.amazon.com/rds/

of a *related patterns* reference is defined by a description [FLR+14]. For example, as described in Table 2.1, the *Relational Database* pattern references the *Key-Value Storage* [FLR+14] pattern and the *Stateless Component* [FLR+14] pattern. The *Key-Value Storage* pattern should be used instead of the *Relational Database* pattern if the represented domain model is simple, not subject to frequent changes, and scalability and performance need to be improved. The *Stateless Component* pattern can provide a coherent design in combination with the *Relational Database* pattern as stateless components do not hold application data and, therefore, rely on a data storage.

The pattern languages used for the examples described in this thesis are the aforementioned Cloud Computing Patterns introduced by Fehling et al. [FLR+14], the Enterprise Integration Patterns introduced by Hohpe and Woolf [HW04], and the Security Patterns introduced by Schumacher et al. [SFH+06]. The Cloud Computing Patterns describe problems and their proven solutions in the domain of cloud computing. The Enterprise Integration Patterns describe strategies for the integration of enterprise applications using asynchronous messaging. The Security Patterns define proven solutions to typical security problems at the enterprise level, the architectural level, and the operational level.

2.3 Pattern-based Deployment and Configuration Models

To combine the benefits of declarative deployment models and pattern languages, Pattern-based Deployment and Configuration Models (PbDCMs) were defined which introduce patterns as first-class citizens in deployment models. PbDCMs are based on EDMM [WBF+19] and can contain two types of patterns in addition to the elements defined by EDMM. Patterns specifying the abstract semantics of Components are defined as *Component Patterns*. Component Patterns, Components, and Relations can be annotated with *Behavior Patterns* which specify the abstract behavior of the respective element. Component Patterns and Behavior Patterns can therefore be used in PbDCMs instead of modeling concrete elements and their configuration. This enables a modeling approach focused on the abstract semantics of a deployment instead of its technical details and, thus, reduces the technical knowledge and the time needed to create the deployment while also reducing the risk of errors. Furthermore, as the patterns contained in a PbDCMs are independent of particular providers and technologies, the vendor lock-in effect is avoided [HBF+20; HBM+18].

Figure 2.2 shows an exemplary PbDCM. The left side of Figure 2.2 shows a subgraph of the running example depicted in Figure 1.1 containing the *Result Processor* Component which retrieves the digitalized documents from the *Output Queue* Component and writes them to the MySQL DB 8 database of the *Document Storage* Component. The right side of Figure 2.2 shows a PbDCM which represents the abstract design decisions realized in the subgraph by means of Component Patterns and Behavior Patterns. As the local OpenStack Victoria infrastructure is running on-premise, it realizes the *Private Cloud* [FLR+14] Component Pattern and is therefore represented by this element in the PbDCM. The Tomcat 10 Component is used to execute the Java 11 App *Result Processor* and, therefore, realizes the *Execution Environment* [FLR+14] Component Pattern. The *Document Storage* Component is implemented as a MySQL DB 8 database and therefore realizes the *Relational Database* Component Pattern. The Java 11 App *Result Processor* contains business functionality not representable by a pattern and is therefore also contained in the PbDCM. As the Java 11 App *Result Processor* of the deployment model is expected to experience a constant amount of utilization and the Ubuntu 20.04 Component is configured with fixed values for its RAM and storage, the

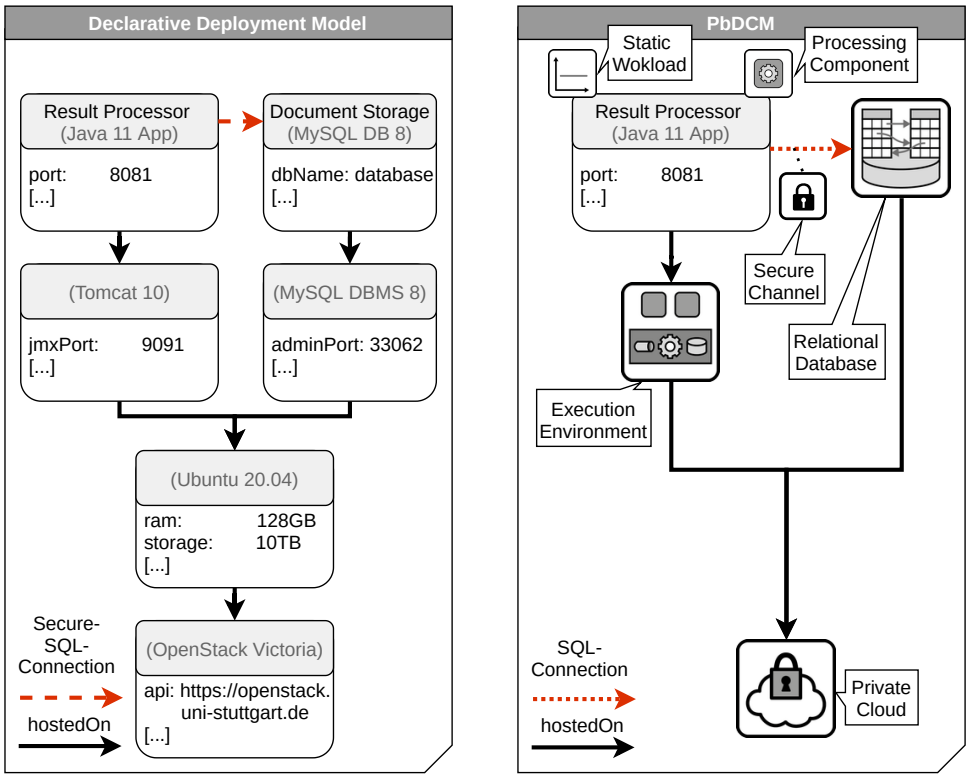

Figure 2.2: Subgraph of the running example depicted in Figure 1.1 and a corresponding PbDCM.

Java 11 App *Result Processor* of the PbDCM is annotated with the *Static Workload* [FLR+14] Behavior Pattern. Additionally, it is annotated with the *Processing Component* [FLR+14] Behavior Pattern as the Java 11 App *Result Processor* of the deployment model implements a self-contained part of functionality and is decoupled from the digitalization procedure via the *Output Queue* Component. Finally, as the Relation between the Java 11 App *Result Processor* and the MySQL DB 8 *Document Storage* in the deployment model is of type *Secure-SQL-Connection*, it realizes the *Secure Channel* [SFH+06] Behavior Pattern and is therefore represented by a *SQL-Connection* Relation annotated with this pattern in the PbDCM.

2.3.1 Metamodel for Pattern-based Deployment and Configuration Models

As described in Section 2.3, PbDCMs are based on EDMM [WBF+19]. Figure 2.3 depicts the metamodel for PbDCMs where elements already present in EDMM are visualized by rectangles with a white background and the extensions to the model are visualized by rectangles with a grey background. The newly introduced elements are the described Component Patterns and Behavior Patterns including their respective types. Behavior Patterns are divided into elements that are applicable to Components or Component Patterns and elements that are applicable to Relations. This is visualized in Figure 2.3 by the *Component Behavior Pattern* and the *Relation Behavior Pattern* elements. All newly introduced elements are instances of their respective types which is visualized in Figure 2.3 by the *Component Pattern Type*, the *Component Behavior Pattern Type*,

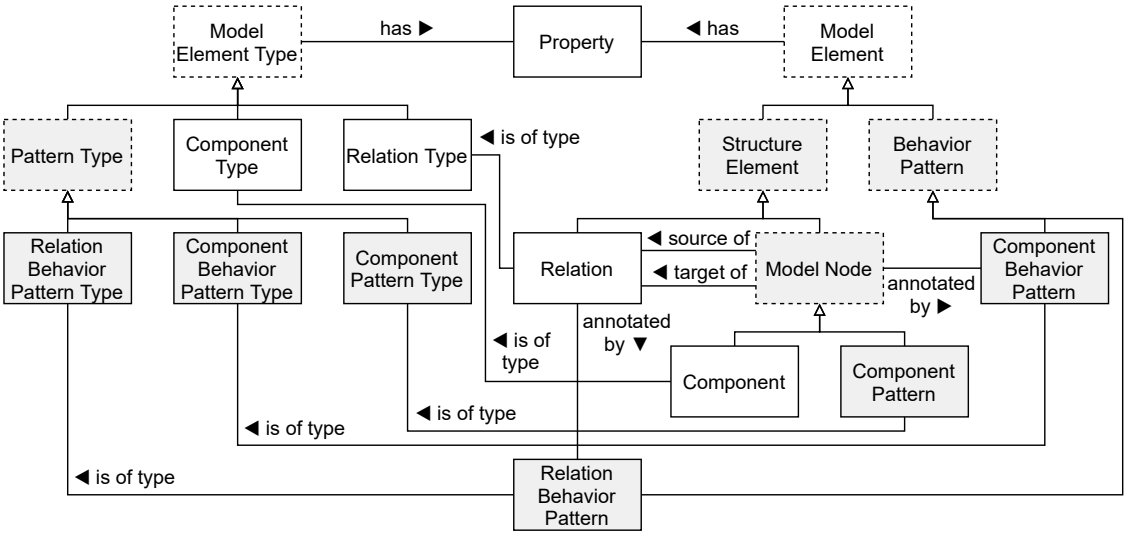

Figure 2.3: Metamodel for PbDCMs [HBF+20; HBM+18].

and the *Relation Behavior Pattern Type* elements. It should be noted that a PbDCM can contain patterns but is not required to. Therefore, the declarative deployment models presented in this thesis are by definition also PbDCMs [HBF+20; HBM+18].

In the following, the metamodel shown in Figure 2.3 is formally defined analogously to the definitions introduced by Harzenetter et al. [HBF+20; HBM+18]. A PbDCM is a fifteen-tuple $t \in \mathcal{T}$ with $\mathcal{T}$ being the set of all PbDCMs. The tuple t consists of eleven sets and four maps defined with the support of multiple union sets as follows [HBF+20; HBM+18]:

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

The sets of t are defined as follows [HBF+20; HBM+18]:

- C_t is the set of Components in t as described by EDMM. A Component $c_i \in C_t$ is a physical, functional, or logical unit of the respective application and can be the source or target of a Relation.
- CP_t is the set of Component Patterns in t. A Component Pattern $cp_i \in CP_t$ is a pattern specifying the abstract semantics of a Component and can be the source or target of a Relation.
- $MN_t := C_t \cup CP_t$ is the set of *Model Nodes* in t. As depicted in Figure 2.3, a Model Node can be a Component or a Component Pattern.
- $R_t \subseteq MN_t \times MN_t$ is the set of Relations in t as described by EDMM. A Relation $r_i = (mn_{source}, mn_{target}) \in R_t$ is a directed physical, functional, or logical dependency between a Model Node $mn_{source} \in MN_t$ representing the source of the Relation and a Model Node $mn_{target} \in MN_t$ representing the target of the Relation.
- CBP_t is the set of Component Behavior Patterns in t, i.e., the Behavior Patterns that are applicable to Components and Component Patterns, or in other terms to Model Nodes. A Component Behavior Pattern $cbp_i \in CBP_t$ specifies the abstract behavior of a Model Node $mn_j \in MN_t$ annotated with it.

- RBP_t is the set of Relation Behavior Patterns in t, i.e., the Behavior Patterns that are applicable to Relations. A Relation Behavior Pattern $rbp_i \in RBP_t$ specifies the abstract behavior of a Relation $r_j \in R_t$ annotated with it.
- CT_t is the set of Component Types in t as described by EDMM. A Component Type $ct_i \in CT_t$ specifies the semantics of a Component $c_j \in C_t$ which has this type assigned to it.
- 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 $cp_j \in CP_t$ which has this type assigned to it.
- RT_t is the set of Relation Types in t as described by EDMM. A Relation Type $rt_i \in RT_t$ specifies the semantics of a Relation $r_j \in R_t$ which has this type assigned to it.
- $CBPT_t$ is the set of Component Behavior Pattern Types in t. A Component Behavior Pattern Type $cbpt_i \in CBPT_t$ specifies the semantics of a Component Behavior Pattern $cbp_j \in CBP_t$ which has this type assigned to it.
- $RBPT_t$ is the set of Relation Behavior Pattern Types in t. A Relation Behavior Pattern Type $rbpt_i \in RBPT_t$ specifies the semantics of a Relation Behavior Pattern $rbp_j \in RBP_t$ which has this type assigned to it.
- $PROP_t \subseteq \Sigma^+ \times \Sigma^+$ is the set of Properties in t. A Property $pr_i = (Key, Value) \in PROP_t$ describes the configuration of a Component, Component Pattern, Relation, Component Behavior Pattern, Relation Behavior Pattern, or their types. The initial value of a Property is the *Empty Word* ε which is equivalent to the Property not being set.

The following union sets are defined based on the sets of t as follows [HBF+20; HBM+18]:

- $SE_t := R_t \cup MN_t$ is the set of all *Structure Elements* in t. As shown in Figure 2.3, SE_t includes all Relations and Model Nodes, i.e., all Relations, Components, and Component Patterns.
- $BP_t := CBP_t \cup RBP_t$ is the set of all Behavior Patterns in t. As depicted in Figure 2.3, BP_t includes all Component Behavior Patterns and Relation Behavior Patterns. For brevity, Component Behavior Patterns and Relation Behavior Patterns will not be differentiated in this thesis but collectively referred to as Behavior Patterns.
- $ME_t := SE_t \cup BP_t$ is the set of all *Model Elements* in t. As depicted in Figure 2.3, ME_t includes all Structure Elements and Behavior Patterns.
- $MET_t := CT_t \cup CPT_t \cup RT_t \cup CBPT_t \cup RBPT_t$ is the set of all *Model Element Types* in t. As depicted in Figure 2.3, MET_t includes all Component Types, Relation Types, and *Pattern Types*, i.e., all Component Types, Relation Types, Component Pattern Types, Component Behavior Pattern Types, and Relation Behavior Pattern Types.

The maps of t are defined as follows [HBF+20; HBM+18]:

- $type_t$ is the map that assigns each Model Element $me_i \in ME_t$ to its corresponding Model Element Type $met_j \in MET_t$ specifying the semantics of me_i:

$$type_t : ME_t \rightarrow MET_t$$

- $supertype_t$ is the map that assigns each Model Element Type $met_i \in MET_t$ to its corresponding supertype $met_j \in MET_t$ where $i \neq j$. Additionally, the map $supertypes_t$ assigns each Model Element Type $met_k \in MET_t$ to its set of transitive supertypes. The two maps are defined as follows:

$$supertype_t : MET_t \rightarrow MET_t$$
$$supertypes_t : MET_t \rightarrow \wp(MET_t)$$

- $properties_t$ is the map that assigns each Model Element $me_i \in ME_t$ and Model Element Type $met_j \in MET_t$ to its corresponding set of Properties:

$$properties_t : ME_t \cup MET_t \rightarrow \wp(PROP_t)$$

- $annotations_t$ is the map that assigns each Structure Element $se_i \in SE_t$ to to its set of annotated Behavior Patterns:

$$annotations_t : SE_t \rightarrow \wp(BP_t)$$

2.3.2 Refinement of Pattern-based Deployment and Configuration Models

While PbDCMs reduce the technical knowledge required to create a deployment and avoid the vendor lock-in effect, they do not represent executable deployment models. The Component Patterns contained in a PbDCM describe the abstract semantics of Components but do not provide concrete implementations. The Behavior Patterns a Component, Component Pattern, or Relation can be annotated with describe the abstract behavior of the respective elements but do not specify the configuration needed to achieve said behavior. Therefore, to obtain an executable deployment model which realizes the patterns contained in a PbDCM, Component Patterns need to be replaced by concrete Components and the abstract behavior specified by Behavior Patterns needs to be realized by the configuration of the Components and Relations. To facilitate this refinement process of PbDCMs, Component and Behavior Pattern Refinement Models (CBPRMs) were introduced [HBF+20; HBM+18].

A CBPRM consist of two PbDCMs and two types of mappings [HBF+20; HBM+18]. The *Detector* of a CBPRM represents a PbDCM which defines abstract design decisions by means of Component Patterns and Behavior Patterns [HBF+20; HBM+18]. The *Refinement Structure* of a CBPRM represents a PbDCM which realizes the patterns contained in the Detector, i.e., contains concrete Components realizing the abstract semantics specified by the Component Patterns and is configured to realize the abstract behavior specified by the Behavior Patterns [HBF+20; HBM+18]. Figure 2.4 depicts an exemplary CBPRM where the Detector is shown on the left side and the Refinement Structure is shown on the right side. In this CBPRM, the Detector consists of a Java 11 App Component hosted on an *Execution Environment* Component Pattern which is in turn hosted on a *Private Cloud* Component Pattern. The Java 11 App Component is annotated with the *Static Workload* Behavior Pattern and the *Processing Component* Behavior Pattern. The Refinement Structure of the CBPRM consists of a Java 11 App Component which is executed by a Tomcat 10 Component running on an Ubuntu 20.04 Component which is configured with fixed values for its RAM and storage and is hosted on an OpenStack Victoria Component.

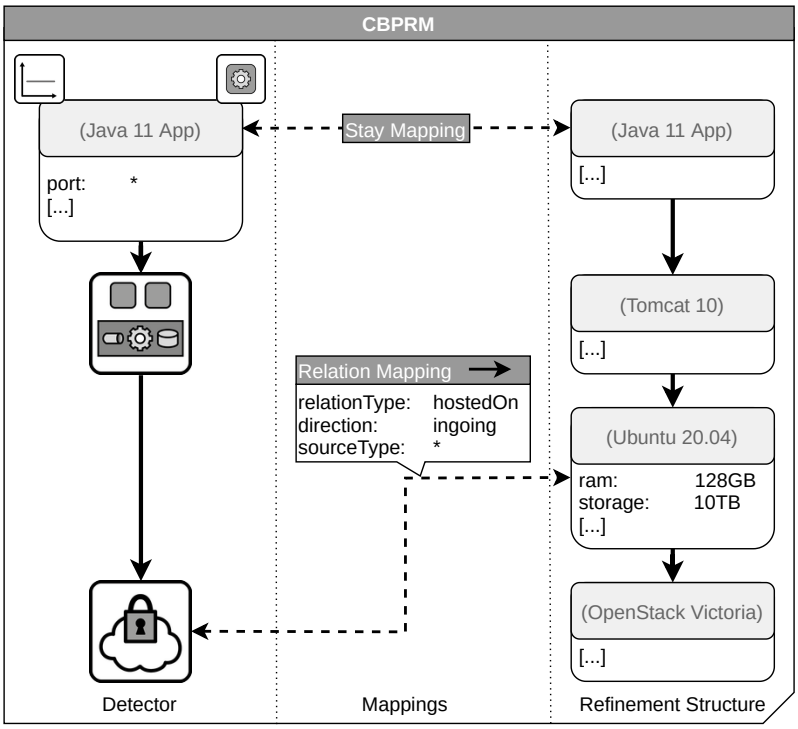

Figure 2.4: CBPRM applicable to a subgraph of the PbDCM depicted in Figure 2.2.

In addition to the two PbDCMs, CBPRMs can specify *Stay Mappings* and *Relation Mappings* which define the correspondence between elements of the Detector and elements of the Refinement Structure [HBF+20; HBM+18]. A Stay Mapping links a Model Node of the Detector to a Model Node of the Refinement Structure defining that the Model Node is equally present in both structures [HBF+20; HBM+18]. A Relation Mapping links a Model Node of the Detector to a Model Node of the Refinement Structure defining the redirection of external Relations, i.e., Relations not included in a CBPRM, between the respective Model Nodes [HBF+20; HBM+18]. In Figure 2.4, the Java 11 App Component of the Detector is connected to the Java 11 App Component of the Refinement Structure by a Stay Mapping to define that this Component is equally present in both structures. Furthermore, the *Private Cloud* Component Pattern of the Detector is connected to the Ubuntu 20.04 Component of the Refinement Structure by a Relation Mapping. This Relation Mapping is defined to apply to all ingoing external Relations of type *hostedOn* with no restriction on the type of the Model Node the Relation originates from. Other Relation Mappings which may be required to complete the CBPRM are omitted in this example for brevity.

During the pattern refinement process, CBPRMs are used in a semi-automatic and iterative fashion to obtain an executable deployment model from a given PbDCM. Each iteration can be divided into three steps. First, the available CBPRMs are compared to the subgraphs of the given PbDCM. In this comparison, a CBPRM is referred to as *applicable* if the Detector of the CBPRM matches a subgraph of the given PbDCM in terms of structure, types and Properties of the contained Structure Elements, and types of the annotated Behavior Patterns. Next, a user can select one of the CBPRMs which were determined to be applicable to the given PbDCM. Finally, the selected CBPRM is used to replace the matching subgraph of the given PbDCM with the Refinement Structure of the CBPRM.

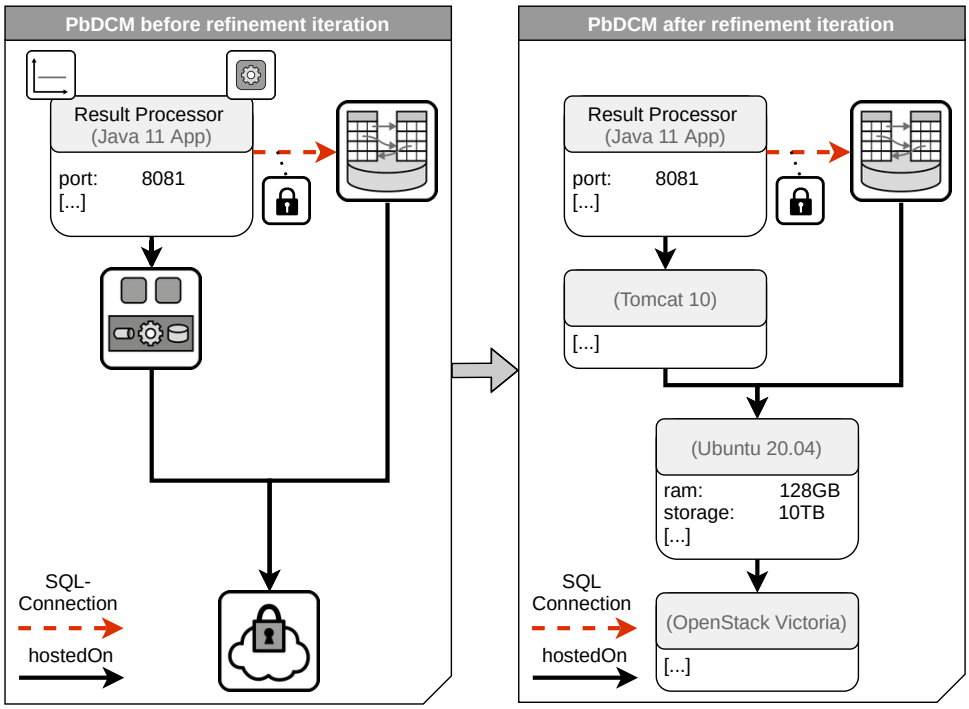

Figure 2.5: Partial refinement of the PbDCM depicted in Figure 2.2 using the CBPRM shown in Figure 2.4.

The replacement procedure is supported by the Stay Mappings and Relation Mappings defined for the respective CBPRM. The pattern refinement process stops if there are no more patterns in the PbDCM or no more applicable CBPRMs can be determined [HBF+20; HBM+18].

Figure 2.5 shows one iteration of the pattern refinement process for the PbDCM depicted in Figure 2.2 using the CBPRM illustrated in Figure 2.4. The left side of Figure 2.5 shows the PbDCM which was already discussed in the beginning of Section 2.3. It is assumed that all applicable CBPRMs have been determined by comparing the respective Detectors to the subgraphs of the PbDCM and a user has selected one of the applicable CBPRMs. The selected CBPRM is the CBPRM depicted in Figure 2.4. This CBPRM is applicable to the subgraph of the *Result Processor* depicted on the right side of Figure 2.2 as its Detector (i) matches the structure of the subgraph, (ii) matches the types of the contained Structure Elements and Behavior Patterns, and (iii) matches the Properties of the contained Structure Elements as the Detector specifies the wildcard value “*" for the *port* Property, i.e., any non-empty value is allowed for this Property. The matching subgraph of the PbDCM is now refined using the Refinement Structure of the CBPRM. The result of the refinement iteration is depicted on the right side of Figure 2.5. To achieve this result, the Refinement Structure of the CBPRM is inserted into the original PbDCM shown on the left side of Figure 2.5. The *Execution Environment* Component Pattern, the *Private Cloud* Component Pattern, and the *hostedOn* Relation between the two can now be removed as these patterns are realized by the inserted Tomcat 10 Component, the Ubuntu 20.04 Component, the OpenStack Victoria Component and the Relations between the Components [HBF+20; HBM+18].

The Java 11 App Component of the original PbDCM is required without adaptation in the final executable deployment model and, therefore, neither its type nor its configuration must change during the refinement. To retain the Java 11 App Component of the original PbDCM, it must not be replaced by the Java 11 App Component of the inserted Refinement Structure although it matches the respective element of the Detector of the CBPRM. However, the Java 11 App Component must still be modeled in the Detector of the CBPRM to match the annotated Behavior Patterns and must also be modeled in the Refinement Structure to specify where the Java 11 App Component will be located after the refinement. The Java 11 App Component of the Detector and the Java 11 App Component of the Refinement Structure are therefore connected by a Stay Mapping in the CBPRM as depicted in Figure 2.4. In the pattern refinement process, a Model Node of a PbDCM is not replaced if the matching Model Node of the Detector of a CBPRM is associated with a Stay Mapping. Instead, all Relations originating from or targeted at Model Nodes associated with a Stay Mapping in a CBPRM are redirected to the respective Model Nodes in the given PbDCM. Therefore, the *hostedOn* Relation between the inserted Java 11 App Component and the Tomcat 10 Component is redirected to the Java 11 App Component of the original PbDCM. Additionally, all Behavior Patterns contained in the Detector of the CBPRM are removed from the matching subgraph of the original PbDCM as they are now realized by the inserted Components and their configuration. Furthermore, the Java 11 App Component which was inserted as part of the Refinement Structure of the CBPRM is removed again [HBF+20; HBM+18].

As the *Private Cloud* Component Pattern was removed from the original PbDCM, the *hostedOn* Relation originating from the *Relational Database* Component Pattern currently has no target and needs to be redirected. This is achieved by applying the Relation Mapping which is visualized in Figure 2.4. As previously described, Relation Mappings are used to redirect Relations external to a CBPRM. The *hostedOn* Relation originating from the *Relational Database* Component Pattern is such an external Relation as it is not included in the CBPRM depicted in Figure 2.4. The Relation Mapping defined for the CBPRM is applicable to ingoing Relations of type *hostedOn* and defines the wildcard value “*" for the source type, i.e., it is applicable to Relations originating from Model Nodes of any type. Therefore, this Relation Mapping is applicable to the Relation originating from the *Relational Database* Component Pattern as (i) the Relation is of type *hostedOn*, (ii) the Relation originally was of ingoing direction to the *Private Cloud* Component Pattern, and (iii) the type of the *Relational Database* Component Pattern fulfills the wildcard condition. By applying the Relation Mapping, the *hostedOn* Relation originating from the *Relational Database* Component Pattern is redirected to the inserted Ubuntu 20.04 Component. This concludes one iteration of the pattern refinement process with the resulting PbDCM shown on the right side of Figure 2.5. As this PbDCM still contains the *Secure Channel* Behavior Pattern and the *Relational Database* pattern, additional iterations are required to refine all patterns. Furthermore, additional manual adaptations may be necessary after all patterns have been refined, e.g., to configure the inserted Tomcat 10 Component and the OpenStack Victoria Component and, thus, obtain an executable deployment model [HBF+20; HBM+18].

3 Pattern Detection in Declarative Deployment Models

This chapter presents an automated approach for the detection of patterns in declarative deployment models. Section 3.1 provides an overview of the approach and describes the phases of the pattern detection process. In Section 3.2, the CBPRMs introduced for the pattern refinement process are repurposed and extended to support the pattern detection approach. Section 3.3 and Section 3.4 present the main contribution of this thesis. Section 3.3 presents the first phase of the pattern detection process which detects Behavior Patterns as well as Component Patterns. Section 3.4 presents the second phase of the pattern detection process which detects additional Component Patterns based on the results of the first phase.

3.1 Overview of the Pattern Detection Process

The main idea of the approach presented in this thesis is to use the CBPRMs defined by Harzenetter et al. [HBF+20; HBM+18] for the detection of patterns in declarative deployment models. During the pattern refinement process, the Detector of a CBPRM is used to determine matching subgraphs in a PbDCM which can then be replaced by the Refinement Structure of the CBPRM [HBF+20; HBM+18]. For the detection of patterns, this process is inverted, i.e., the Refinement Structure of a CBPRM is used to determine matching subgraphs in a deployment model and the respective Detector then represents the patterns realized by the subgraph. Furthermore, the detected patterns are used to build a PbDCM for a given deployment model in a similar way as the pattern refinement process builds an executable deployment model for a given PbDCM. In fact, the algorithms presented in this chapter and the implementation presented in Chapter 4 reuse the algorithms and the implementation of the pattern refinement process. This is possible as the elements of CBPRMs introduced in Section 2.3.2 are also applicable to the detection of patterns and the subsequent creation of PbDCMs. Specifically, Stay Mappings are also applicable in the inverse direction to specify that a Model Node must not be replaced during the creation of a PbDCM and Relation Mappings are also applicable in the inverse direction to redirect the Relations of an executable deployment model.

While the inverted process can already be used to detect patterns, it still has room for improvement. A key difference between the pattern refinement process and the approach presented in this thesis is the level of abstraction before and after the execution of the respective process. The pattern refinement process generates an executable deployment model based on an abstract PbDCM [HBF+20; HBM+18]. Therefore, the conditions for applying a CBPRM during pattern refinement must be strict to ensure that the final result is an executable combination of Components, Relations, and Properties [HBF+20; HBM+18]. In contrast, the approach presented in this thesis generates an abstract PbDCM based on the patterns detected in an executable deployment model. As by definition

CBPRM	PDRM
Detector	Pattern Structure
Refinement Structure	Executable Structure
Relation Mappings	Relation Mappings
Stay Mappings	Stay Mappings
–	Property Mappings
–	Behavior Pattern Mappings
–	Component Pattern Mappings

Table 3.1: Comparison of CBPRMs and PDRMs.

PbDCMs are not executable [HBF+20; HBM+18], the conditions for applying a CBPRM can be less strict. The final result must still be a valid PbDCM but the Components, Relations, Properties, and especially the pattern elements do not need to fit together as coherently as in an executable deployment model. Therefore, the focus of this thesis is the development of less strict conditions which enable applying a CBPRM during pattern detection even if the Refinement Structure of the CBPRM does not completely match the subgraph of a deployment model. These less strict conditions are enabled by repurposing and extending CBPRMs. The repurposed and extended CBPRMs will be introduced in Section 3.2 as Pattern Detection and Refinement Models (PDRMs) which can be used for the detection of patterns as well as the refinement of patterns. Table 3.1 shows a comparison between the elements of CBPRMs and PDRMs. In PDRMs, the Detector of CBPRMs is called *Pattern Structure* and the Refinement Structure of CBPRMs is called *Executable Structure*. The Relation Mappings and the Stay Mappings introduced for CBPRMs are adopted for PDRMs. Furthermore, PDRMs contain *Property Mappings* [WBH+20], *Behavior Pattern Mappings*, and *Component Pattern Mappings* to support the pattern detection process.

As the described PDRMs and the pattern detection process introduced in this thesis are based on the CBPRMs introduced for the pattern refinement process, the fundamentals of the respective models and the two processes are the same. The pattern refinement process is executed in a semi-automatic and iterative fashion [HBF+20; HBM+18]. As a result, the pattern detection process presented in this thesis is also a semi-automatic and iterative process. Specifically, the pattern detection process consists of the following two phases which are depicted in Figure 3.1: (i) *Detection of Behavior Patterns and Component Patterns* and (ii) *Detection of Additional Component Patterns*. The input of the process is a declarative deployment model in which patterns are to be detected and the final result of the process is a PbDCM which provides a comprehensive view of the detected patterns.

To increase the effectiveness of the pattern detection process, two separate detection loops are used as indicated by the revolving arrows in Figure 3.1 for the first and second phase, respectively. Both phases utilize the described PDRMs which are provided by a *PDRM Repository*. PDRMs are visualized in Figure 3.1 above the PDRM Repositories with the Executable Structure shown on the left and the Pattern Structure shown on the right side of the PDRMs. The two detection loops are executed in a semi-automatic and iterative fashion with each iteration of a loop consisting of the following three steps: (i) automatic determination all applicable PDRMs, (ii) manual selection of one of the determined PDRMs, and (iii) automatic application of the selected PDRM to the given deployment model. A PDRM is *applicable* to a given deployment model if its Executable Structure matches a subgraph of the deployment model in terms of structure and contained elements. To

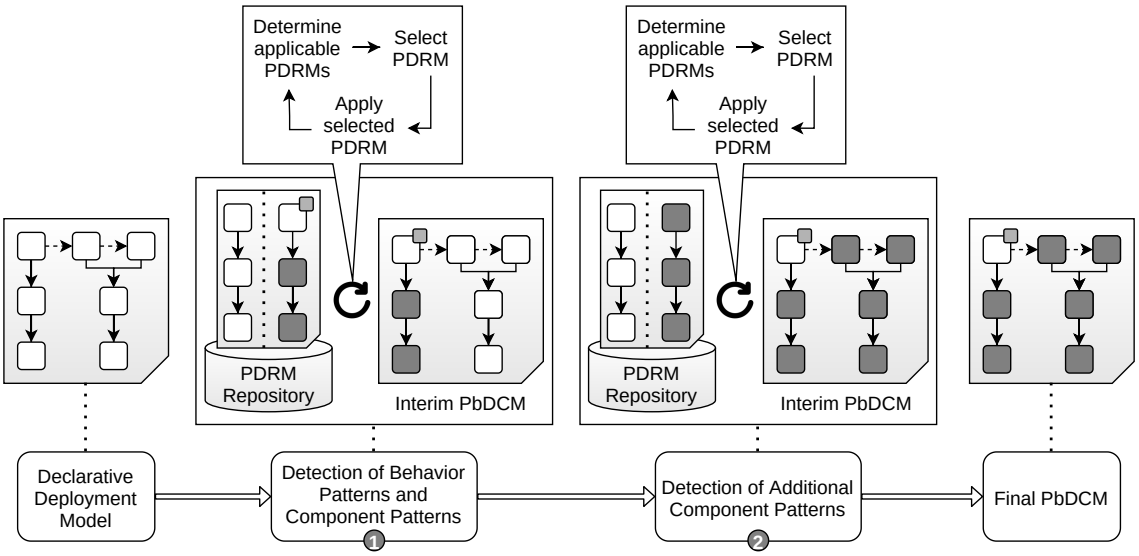

Figure 3.1: Overview of the pattern detection process.

apply a selected PDRM to a given deployment model, the matching subgraph is replaced with the Pattern Structure of the PDRM, Relation Mappings are used to redirect external Relations to the inserted elements of the Pattern Structure, and Stay Mappings are used to retain elements that must not be replaced. Furthermore, the first phase of the pattern detection process utilizes the Property Mappings and Behavior Pattern Mappings of PDRMs and the second phase of the pattern detection process utilizes the Component Pattern Mappings of PDRMs. After a PDRM was applied, the next iteration of the respective loop begins. During each iteration of each loop, a user can choose to stop the pattern detection process. To be precise, a user can choose to stop the pattern detection process during an iteration after all applicable PDRMs have been determined and before he or she selects one of the PDRMs for application.

The detection loop of the first phase detects Behavior Patterns and Component Patterns in a given declarative deployment model and is executed in a semi-automatic and iterative fashion with each iteration consisting of the three described steps. First, all applicable PDRMs are determined. In this phase, a PDRM is applicable to a given deployment model if its Executable Structure matches a subgraph of the deployment model in terms of structure, types of the contained elements, and Properties of the contained elements. To conditionally detect Behavior Patterns based on the Properties configured in a subgraph, Behavior Pattern Mappings are utilized. The specifics of these compatibility conditions between the Executable Structure of a PDRM and the subgraph of a deployment model will be explained in Section 3.3.2. Next, one of the determined PDRMs is selected which is then applied to the deployment model and, as a result, creates an *interim PbDCM*, i.e., a PbDCM which is not the final result of the pattern detection process. To apply the selected PDRM, the matching subgraph of the deployment model is replaced with the Pattern Structure of the PDRM supported by the defined Relation Mappings and Stay Mappings. Additionally, Property Mappings are used to transfer Property values from the matching subgraph to the Pattern Structure. Furthermore, if a Behavior Pattern annotated at the Pattern Structure of the PDRM was determined to not be realized by the Properties configured in the matching subgraph, it is not added while applying the PDRM. The details of the replacement procedure will be described in Section 3.3.3. After a PDRM was applied, the next iteration of the loop begins which receives the created interim

PbDCM from the previous iteration as input. If, during this iteration, a user chooses to stop the pattern detection process, the second phase is omitted and the created interim PbDCM is returned as the final result. In other words, the second phase of the pattern detection process is optional. If no more applicable PDRMs can be determined, the first phase of the pattern detection process is completed and the created interim PbDCM is passed to the second phase as depicted in Figure 3.1.

The detection loop of the second phase detects additional Component Patterns in the interim PbDCM created by the first phase and is executed in a semi-automatic and iterative fashion with each iteration consisting of the three described steps. First, all applicable PDRMs are determined. To determine if a PDRM is applicable, the second phase does not consider the detection of Behavior Patterns but only the detection of Component Patterns. As the detection of Behavior Patterns is not considered, the second phase of the process uses less strict conditions for a successful match of a PDRM. Therefore, in this phase, a PDRM is applicable to a given deployment model if its Executable Structure matches a subgraph of the deployment model in terms of structure and types of the contained elements but all Properties are ignored. Furthermore, the supertypes of the contained elements and the Component Patterns realized by the contained Components are considered during the compatibility check. To determine the Component Patterns realized by the contained Components, the Component Pattern Mappings of PDRMs are utilized. The specifics of these compatibility conditions between the Executable Structure of a PDRM and the subgraph of a deployment model will be explained in Section 3.4.2. Next, one of the determined PDRMs is selected which is then applied to the deployment model and, as a result, creates another interim PbDCM. To apply the selected PDRM, the matching subgraph of the deployment model is replaced with the Pattern Structure of the PDRM supported by the defined Relation Mappings and Stay Mappings. The details of the replacement procedure will be described in Section 3.4.3. After a PDRM was applied, the next iteration of the loop begins which receives the created interim PbDCM from the previous iteration as input. If, during this iteration, a user chooses to stop the pattern detection process or no more applicable PDRMs can be determined, the second phase of the pattern detection process is completed and the created interim PbDCM is returned as the final result.

The final result of the pattern detection process is a PbDCM representing the Behavior Patterns and the Component Patterns realized by the application. The detected patterns can be used for a better understanding of the application and to guide future development [FLR+14]. The generated PbDCM provides a comprehensive view of the detected patterns and enables the application of the pattern refinement process.

3.2 Metamodel for Pattern Detection and Refinement Models

As described in Section 3.1, the approach presented in this thesis utilizes PDRMs for the detection of patterns in deployment models and the creation of corresponding PbDCMs. In this section, all elements of PDRMs are formally defined. PDRMs are based on CBPRMs and can therefore be used for the detection of patterns as well as the refinement of patterns. To emphasize that PDRMs can be used for both purposes, the Refinement Structure and the Detector of CBPRMs are renamed in PDRMs to Executable Structure and Pattern Structure, respectively. The Relation Mappings and Stay Mappings of CBPRMs are adopted without changes for PDRMs. Furthermore, Property Mappings [WBH+20], Behavior Pattern Mappings, and Component Pattern Mapping are defined to support the pattern detection process.

Let *PDRM* be the set of all Pattern Detection and Refinement Models, then a $pdrm \in PDRM$ is a seven-tuple defined as follows:

$$pdrm = (es_{pdrm}, ps_{pdrm}, RM_{pdrm}, S_{pdrm}, PM_{pdrm}, BPM_{pdrm}, CPM_{pdrm})$$

The following elements of *pdrm* are adopted from CBPRMs and were, with the exception of Property Mappings [WBH+20], already informally described in Section 2.3.2 [HBF+20; HBM+18]:

- $es_{pdrm} \in \mathcal{T}$ is a PbDCM representing the Executable Structure in *pdrm* which realizes some of the patterns contained in the Pattern Structure.
- $ps_{pdrm} \in \mathcal{T}$ is a PbDCM representing the Pattern Structure in *pdrm* which defines abstract design decisions by means of Component Patterns and Behavior Patterns.
- RM_{pdrm} is the set of Relation Mappings in *pdrm*. Relation Mappings link Model Nodes of the Executable Structure es_{pdrm} to Model Nodes of the Pattern Structure ps_{pdrm} defining the redirection of external Relations between the respective Model Nodes. A Relation Mapping $rm_i \in RM_{pdrm}$ is defined as follows:

  $$rm_i = (mn_{es}, mn_{ps}, rt, direction_{rt}, vt)$$

  Herein, $mn_{es} \in MN_{es_{pdrm}}$ is a Model Node in the Executable Structure es_{pdrm} and $mn_{ps} \in MN_{ps_{pdrm}}$ is a Model Node in the Pattern Structure ps_{pdrm}. $rt \in RT$ is the Relation Type of a Relation to be redirected and $direction_{rt} \in \{ingoing, outgoing\}$ is the direction of this Relation. $vt \in CT \cup CPT$ specifies the valid type allowed for the source of the Relation if $direction_{rt} = ingoing$ or the valid type allowed for the target of the Relation if $direction_{rt} = outgoing$. If $vt =$ "$*$", i.e., *vt* represents a wildcard, then any type is valid as the source or the target of the Relation.
- S_{pdrm} is the set of Stay Mappings in *pdrm*. Stay Mappings link Model Nodes of the Executable Structure es_{pdrm} to Model Nodes of the Pattern Structure ps_{pdrm} defining that the Model Nodes are equally present in both structures. A Stay Mapping $s_i \in S_{pdrm}$ is a pair of Model Nodes and is defined as follows:

  $$s_i = (mn_{es}, mn_{ps})$$

  Herein, $mn_{es} \in MN_{es_{pdrm}}$ is a Model Node in the Executable Structure es_{pdrm} and $mn_{ps} \in MN_{ps_{pdrm}}$ is a Model Node in the Pattern Structure ps_{pdrm}.
- PM_{pdrm} is the set of Property Mappings [WBH+20] in *pdrm*. Property Mappings define how the Properties specified in the Executable Structure es_{pdrm} correspond to the Properties specified in the Pattern Structure ps_{pdrm}. This enables transferring the value of a Property from the subgraph that is to be replaced to the structure that replaces it. A Property Mapping $pm_i \in PM_{pdrm}$ is defined as follows:

  $$pm_i = (se_{es}, pr_x, se_{ps}, pr_y)$$

  Herein, $pr_x \in properties_{es_{pdrm}}(se_{es})$ is a Property of a Structure Element $se_{es} \in SE_{es_{pdrm}}$ in the Executable Structure es_{pdrm} which corresponds to a Property $pr_y \in properties_{ps_{pdrm}}(se_{ps})$ of a Structure Element $se_{ps} \in SE_{ps_{pdrm}}$ in the Pattern Structure ps_{pdrm}.

As new contribution, the following elements of *pdrm* are introduced to support the pattern detection process presented in this thesis:

- BPM_{pdrm} is the set of Behavior Pattern Mappings in *pdrm*. Behavior Pattern Mappings define the correspondence between a Behavior Pattern and the Properties realizing it. A Behavior Pattern Mapping $bpm_i \in BPM_{pdrm}$ is defined as follows:

  $$bpm_i = (se_{es}, pr_x, se_{ps}, bp_y)$$

  Herein, $pr_x \in properties_{es_{pdrm}}(se_{es})$ is a Property of a Structure Element $se_{es} \in SE_{es_{pdrm}}$ in the Executable Structure es_{pdrm} which is needed to realize a Behavior Pattern $bp_y \in annotations_{ps_{pdrm}}(se_{ps})$ annotated at a Structure Element $se_{ps} \in SE_{ps_{pdrm}}$ in the Pattern Structure ps_{pdrm}.

- CPM_{pdrm} is the set of Component Pattern Mappings in *pdrm*. Component Pattern Mappings define the correspondence between a Component Pattern and the Components realizing it. A Component Pattern Mapping $cpm_i \in CPM_{pdrm}$ is defined as follows:

  $$cpm_i = (mn_{es}, mn_{ps})$$

  Herein, $mn_{es} \in MN_{es_{pdrm}}$ is a Model Node in the Executable Structure es_{pdrm} needed to realize the pattern specified by a Model Node $mn_{ps} \in MN_{ps_{pdrm}}$ in the Pattern Structure ps_{pdrm}.

3.3 Detection of Behavior Patterns and Component Patterns

This section presents the first phase of the pattern detection process which detects Behavior Patterns as well as Component Patterns. Section 3.3.1 provides an overview of the phase and describes how Behavior Pattern Mappings and Property Mappings are used. Section 3.3.2 formally defines the conditions which must be fulfilled by a PDRM to match a deployment model. Section 3.3.3 introduces an algorithm which utilizes the detected patterns to build a PbDCM.

3.3.1 Overview

The first phase of the pattern detection process is executed in a semi-automatic and iterative fashion with each iteration consisting of the three steps described in Section 3.1. During each iteration of the first phase, Component Patterns as well as Behavior Patterns can be detected by comparing the Executable Structures of the available PDRMs to the subgraphs of a given deployment model. A PDRM is applicable to a given deployment model if its Executable Structure matches a subgraph of the deployment model in terms of structure, types of the contained elements, and Properties of the contained elements. If the Executable Structure of a PDRM matches a subgraph of a deployment model, then the Pattern Structure of the PDRM represents the detected patterns. The PDRM can therefore be used to replace the matching subgraph with the Pattern Structure and, thus, create a PbDCM. For example, the Executable Structure of the PDRM depicted in Figure 3.2 matches the subgraph of the *Input Queue* which was discussed in Section 1.1. The Executable Structure consists of an SQS Queue Component hosted on an AWS SQS Component which is in turn hosted on an AWS Component. The *type* Property of the SQS Queue Component is set to *FIFO* and the *sse* Property of the SQS Queue Component is set to *true*, i.e., Server-Side Encryption is enabled.

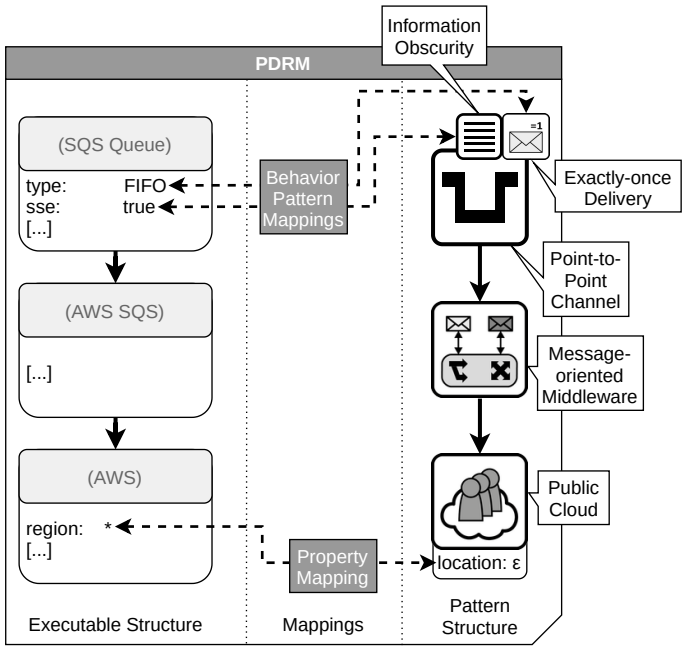

Figure 3.2: A PDRM with two Behavior Pattern Mappings and a Property Mapping.

The *masterKey* Property of the SQS Queue Component and the *account* Property of the AWS SQS Component are not set, i.e., any values are allowed for these Properties. The *region* Property of the AWS Component is set to the wildcard value "*", i.e., any non-empty value is allowed for this Property. The PDRM is therefore applicable to the subgraph of the *Input Queue* depicted in Figure 1.1 as there exists a structural isomorphism between the Executable Structure of the PDRM and the subgraph, the types of the elements of the Executable Structure match the types of the elements of the subgraph, and the Properties of the elements of the Executable Structure match the Properties of the elements of the subgraph.

As the PDRM depicted in Figure 3.2 matches the subgraph of the Input Queue, the subgraph can be replaced by the Pattern Structure of the PDRM to create a PbDCM. The AWS Component of the Executable Structure realizes the *Public Cloud* [FLR+14] Component Pattern and is therefore represented by this element in the Pattern Structure of the PDRM. The AWS SQS Component of the Executable Structure realizes the *Message-oriented Middleware* [FLR+14] Component Pattern. The SQS Queue Component of the Executable Structure realizes the *Point-to-Point Channel* [HW04] Component Pattern and is annotated with the *Information Obscurity* [SFH+06] Behavior Pattern and the *Exactly-once Delivery* [FLR+14] Behavior Pattern. As AWS SQS queues of type FIFO guarantee that each message is delivered once without duplicates [Ama21], the *Exactly-once Delivery* Behavior Pattern is realized by the SQS Queue Component of the Executable Structure which is configured accordingly. The *Information Obscurity* Behavior Pattern is realized by the SQS Queue Component as its *sse* Property is set to *true*.

However, while the PDRM depicted in Figure 3.2 can already be used to detect the patterns realized by the subgraph of the *Input Queue* of the running example and create a corresponding PbDCM, the PDRM is not applicable to the subgraph of the *Output Queue* which was also discussed in Section 1.1. The only difference between the subgraph of the *Input Queue* and the subgraph of the

Output Queue is the *type* Property of the respective SQS Queue Component. In the *Input Queue* Component, the *type* Property is set to *FIFO*. In the *Output Queue* Component, the *type* Property is not set. To enable less strict compatibility conditions and, thus, enable the usage of PDRMs which do not match a considered subgraph exactly, the Behavior Pattern Mappings defined for PDRMs are utilized. As described in Section 3.2, a Behavior Pattern Mapping defines the correspondence between a Behavior Pattern of the Pattern Structure of a PDRM and a Property specified in the Executable Structure of a PDRM. For example, the PDRM depicted in Figure 3.2 defines two Behavior Pattern Mappings. The *Exactly-once Delivery* Behavior Pattern is connected to the *type* Property of the SQS Queue Component which is set to *FIFO*. The *Information Obscurity* Behavior Pattern is connected to the *sse* Property of the SQS Queue Component which is set to *true*. This way, Behavior Pattern Mappings define the configuration required to realize a Behavior Pattern and, thus, enable the conditional detection of Behavior Patterns. If the configuration, i.e., the Properties required to realize a Behavior Pattern are not all set in a considered subgraph as specified by the Behavior Pattern Mappings defined for a PDRM, the PDRM is still applicable as its Pattern Structure can be adapted to not contain the respective Behavior Pattern. For example, by utilizing the defined Behavior Pattern Mappings, the PDRM depicted in Figure 3.2 is also applicable to the subgraph of the *Output Queue* in the running example as the Pattern Structure of the PDRM can be adapted to not contain the *Exactly-once Delivery* Behavior Pattern and the *type* Property required to realize the Behavior Pattern can therefore be ignored.

Behavior Pattern Mappings can also be used to define the correspondence between a Behavior Pattern and multiple Properties, possibly specified in different elements of the Executable Structure of a PDRM. In this case, multiple Behavior Pattern Mappings for the same Behavior Pattern need to be created. If multiple Behavior Pattern Mappings exist for a Behavior Pattern, all specified Properties must be set in a considered subgraph for the Behavior Pattern to be realized. In other words, multiple Behavior Pattern Mappings for the same Behavior Pattern are evaluated using AND semantics. Furthermore, there are also Behavior Patterns for which no Behavior Pattern Mappings can be defined, i.e., Behavior Patterns which do not depend on specific Properties to be realized. Such Behavior Patterns are therefore considered to be realized by the Executable Structure of a PDRM as a whole.

Finally, in this phase of the pattern detection process, Property Mappings [WBH+20] are utilized during the application of a PDRM to transfer the value of a Property from the subgraph that is to be replaced to the Pattern Structure that replaces it. As a result, Property values can be transferred to the final PbDCM created by the pattern detection process which makes them available for future development and for the pattern refinement process. For example, in Figure 3.2, a Property Mapping connects the *region* Property of the AWS Component with the *location* Property of the *Public Cloud* Component Pattern. As a result, the value of the *region* Property set in the AWS Component of a matching subgraph is transferred to the *location* Property of the *Public Cloud* Component Pattern when the PDRM is applied. By applying the PDRM to the running example depicted in Figure 1.1, the *region* Property of the inserted *Public Cloud* Component Pattern would therefore be set to *eu*. For the sake of completeness, it should be noted that the Relation Mappings required to apply the PDRM to the running example were omitted in Figure 3.2.

3.3.2 Pattern Detection

The Executable Structure of a PDRM matches a subgraph of a deployment model if there exists a structural isomorphism between the two and all Structure Elements are compatible. If the Executable Structure of a PDRM matches a subgraph of a deployment model, then the Pattern Structure of the PDRM represents the detected patterns. The PDRM can therefore be used to replace the matching subgraph with the Pattern Structure and, thus, create a PbDCM. In the following, the conditions which must be fulfilled for the Executable Structure of a PDRM to match a subgraph of a deployment model are formally defined.

To determine if there exists a structural isomorphism between the Executable Structure of a PDRM and the subgraph of a deployment model where all Structure Elements are compatible, *Subgraph Mappings* [HBF+20; HBM+18] are utilized. Subgraph Mappings define the correspondence between the elements of two subgraphs and will also be used in Section 3.3.3 to create a PbDCM for a given deployment model. Subgraph Mappings can be obtained, e.g., by adapting one of the VF* algorithms [CFSV04; CFSV17; CFV15]. Let $SM_{es_{pdrm},t}$ be the set of all Subgraph Mappings between the Executable Structure es_{pdrm} of a PDRM and a deployment model t. A Subgraph Mapping $sm_i \in SM_{es_{pdrm},t}$ consists of a number of *Element Mappings* where one Element Mapping $em_j \in sm_i$ is defined as follows [HBF+20; HBM+18]:

$$em_j = (se_{es}, se_t)$$

An Element Mapping em_j is a pair of Structure Elements where $se_{es} \in SE_{es_{pdrm}}$ is a Structure Element in the Executable Structure es_{pdrm} of a PDRM and $se_t \in SE_t$ is a Structure Element in the subgraph of the given deployment model t. The Structure Elements of an Element Mapping need to be compatible according to the *Compatibility Operator* "$\overrightarrow{\approx}_{bcp}$", i.e., $se_{es} \overrightarrow{\approx}_{bcp} se_t$ must hold.

Equation (3.1) defines the Compatibility Operator "$\overrightarrow{\approx}_{bcp}$" where the acronym *bcp* indicates that Behavior Patterns as well as Component Patterns are considered. A Structure Element $se_{es} \in SE_{es_{pdrm}}$ is compatible with a Structure Element $se_t \in SE_t$ according to the Compatibility Operator "$\overrightarrow{\approx}_{bcp}$" if (i) the type of se_t or one of its supertypes is equal to the type of se_{es}, (ii) all annotations defined at se_t are also annotated at se_{es} and vice versa, and (iii) all Properties which are not part of a Behavior Pattern Mapping and are set in se_{es} are equally set in se_t with the exception of Properties set to the Empty Word ε, i.e., any value is allowed for the corresponding Property, and Properties set to the wildcard value "*", i.e., any non-empty value is allowed for the corresponding Property. The Compatibility Operator "$\overrightarrow{\approx}_{bcp}$" is defined as follows:

$$\begin{aligned} se_{es} \overrightarrow{\approx}_{bcp} se_t \Leftrightarrow & \; type_{es_{pdrm}}(se_{es}) \in supertypes_t(se_t) \\ & \wedge \; se_{es} \overrightarrow{\approx}_{bp} se_t \\ & \wedge \; se_{es} \overrightarrow{\approx}_{pr} se_t \end{aligned} \tag{3.1}$$

The Compatibility Operator "$\overrightarrow{\approx}_{bp}$" used in Equation (3.1) considers the compatibility of Behavior Patterns. The compatibility of Behavior Patterns must be considered as the matching subgraph in a deployment model may already be annotated with Behavior Patterns. These annotated Behavior Patterns may have been modeled by a user or may be the result of previous iterations of this phase of the pattern detection process. To prevent the existing Behavior Patterns of se_t from being overwritten, all Behavior Patterns annotated at se_t must also be annotated at se_{es}. To ensure that

se_{es} matches se_t, all Behavior Patterns annotated at se_{es} must also be annotated at se_t. In other words, se_{es} and se_t are compatible if they are annotated with exactly the same Behavior Patterns. The Compatibility Operator "$\vec{\approx}_{bp}$" is defined as follows:

$$\begin{aligned} se_{es} \vec{\approx}_{bp} se_t \Leftrightarrow \big(&\forall bp_i \in annotations_t(se_t) \\ &\exists bp_x \in annotations_{es_{pdrm}}(se_{es})(type_t(bp_i) = type_{es_{pdrm}}(bp_x))\big) \\ \wedge \big(&\forall bp_y \in annotations_{es_{pdrm}}(se_{es}) \\ &\exists bp_j \in annotations_t(se_t)(type_{es_{pdrm}}(bp_y) = type_t(bp_j))\big) \end{aligned} \tag{3.2}$$

The Compatibility Operator "$\vec{\approx}_{pr}$" used in Equation (3.1) considers the compatibility of Properties. As the Pattern Structure of a PDRM can be adapted to not contain a specific Behavior Pattern if its corresponding Behavior Pattern Mappings do not hold, Properties which are part of a Behavior Pattern Mapping do not need to match the corresponding Properties of se_t. All other Properties set in se_{es} must match the Properties set in se_t. If a Property of se_{es} is set to the Empty Word ε, the corresponding Property of se_t can have any value. If a Property of se_{es} is set to the wildcard value "*", the corresponding Property of se_t can have any non-empty value. The Compatibility Operator "$\vec{\approx}_{pr}$" is defined as follows:

$$\begin{aligned} se_{es} \vec{\approx}_{pr} se_t \Leftrightarrow \forall pr_x \in properties_{es_{pdrm}}(se_{es}) \\ \Big(\big(\exists (se_{es}, pr_x, se_{ps}, bp_y) \in BPM_{pdrm}\big) \\ \vee \Big(\pi_2(pr_x) = \varepsilon \vee \exists pr_i \in properties_t(se_t)\big(\pi_1(pr_x) = \pi_1(pr_i) \\ \wedge (\pi_2(pr_x) = \pi_2(pr_i) \vee (\pi_2(pr_x) = \text{“}*\text{”} \wedge \pi_2(pr_i) \neq \varepsilon))\big)\Big)\Big) \end{aligned} \tag{3.3}$$

In summary, the Executable Structure es_{pdrm} of a PDRM matches a subgraph of a given deployment model t if the set of Subgraph Mappings $SM_{es_{pdrm},t}$ is not empty, i.e., there exists a Subgraph Mapping $sm_i \in SM_{es_{pdrm},t}$ where the Structure Elements of all Element Mappings $em_j \in sm_i$ are compatible according to the Compatibility Operator "$\vec{\approx}_{bcp}$".

3.3.3 Generation of Pattern-based Deployment and Configuration Models

In this subsection, the patterns detected by comparing the Executable Structures of PDRMs to the subgraphs of a given deployment model according to the conditions defined in Section 3.3.2 are utilized to generate a PbDCM. As described in Section 3.1, a PbDCM for a deployment model is built in a semi-automatic and iterative fashion with each iteration consisting of three steps. During the third step of each iteration, the matching subgraph of the deployment model is replaced with the Pattern Structure of the PDRM selected by a user. To be precise, a user does not only select a PDRM but also a corresponding Subgraph Mapping as the same PDRM may be applicable to multiple subgraphs. To replace a subgraph with the Pattern Structure of a PDRM, the algorithms for pattern refinement [HBF+20; HBM+18] are utilized. As previously described, the algorithms for pattern refinement can be used to build a PbDCM instead of refining it by inverting the process. In this inverted process, the matching subgraph is replaced with the Pattern Structure of a PDRM

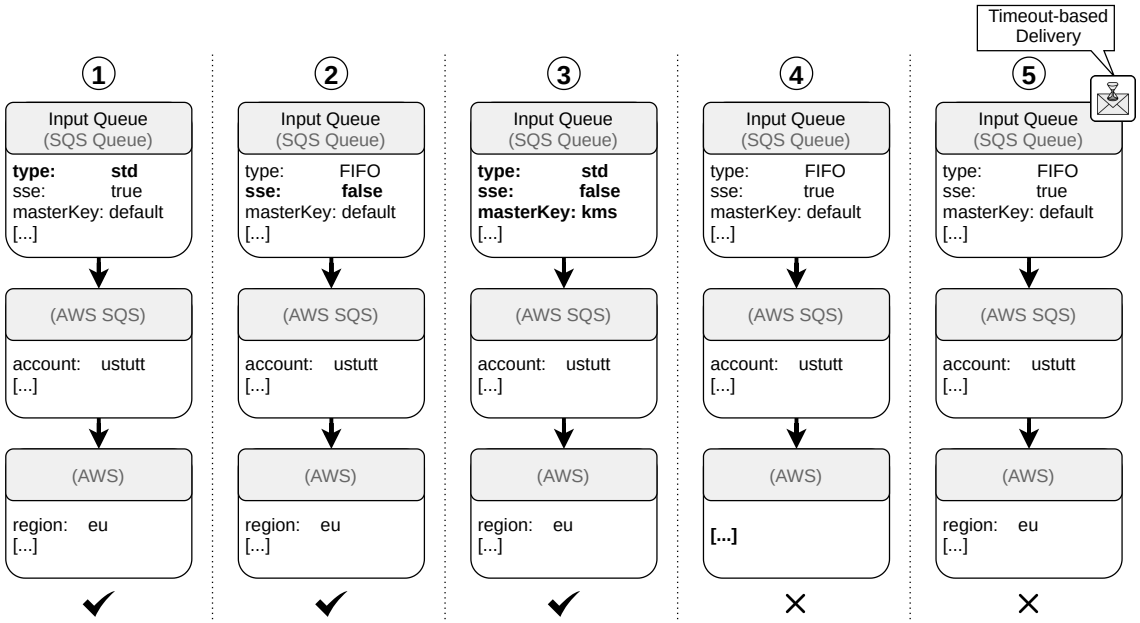

Figure 3.3: Exemplary subgraphs for the detection of Behavior Patterns and Component Patterns.

instead of the Executable Structure, external Relations are redirected to the inserted elements of the Pattern Structure instead of to the inserted elements of the Executable Structure, and Stay Mappings are considered in the inverse direction.

To conditionally detect Behavior Patterns and, thus, enable the usage of PDRMs which do not match a subgraph of a given deployment model exactly, Behavior Pattern Mappings were introduced in Section 3.2. As the Pattern Structure of a PDRM can be adapted to not contain a specific Behavior Pattern if its corresponding Behavior Pattern Mappings do not hold, Properties which are part of a Behavior Pattern Mapping were ignored by the compatibility conditions introduced in Section 3.3.2. For example, Figure 3.3 depicts five variations of the subgraph of the *Input Queue* described in the running example. If a Property was changed in contrast to the subgraph of the running example, the respective Property is highlighted using bold letters. Below each variation, a check mark or a cross indicates whether the PDRM shown in Figure 3.2 is applicable to the depicted subgraph.

In the first variation depicted in Figure 3.3, the *type* Property of the SQS Queue Component has been changed to *std*, i.e., an SQS queue of type standard. As the PDRM depicted in Figure 3.2 specifies a Behavior Pattern Mapping for the *type* Property of the SQS Queue Component, the *type* Property can be ignored and the PDRM is therefore still applicable to the subgraph. However, to use the PDRM for the creation of a PbDCM, the Pattern Structure of the PDRM must be adapted to not contain the *Exactly-once Delivery* Behavior Pattern. In the second variation, the *sse* Property of the SQS Queue Component has been changed to *false*, i.e., Server-Side Encryption is disabled. As the PDRM depicted in Figure 3.2 specifies a Behavior Pattern Mapping for the *sse* Property of the SQS Queue Component, the *sse* Property can be ignored and the PDRM is therefore still applicable to the subgraph. Again, to use the PDRM for the creation of a PbDCM, the Pattern Structure of the PDRM must be adapted to not contain the *Information Obscurity* Behavior Pattern. In the third variation, the *type* Property of the SQS queue Component has been changed to *std*, the *sse* Property has been changed to *false*, and the *masterKey* Property has been changed to the *kms*, i.e., a master

key provided by a Key Management System (KMS). The PDRM depicted in Figure 3.2 is still applicable to the subgraph because of the specified Behavior Pattern Mappings and because the *masterKey* Property of the SQS Queue Component is not set in the PDRM which is equivalent to it being set to the Empty Word ε, i.e., any value is allowed for this Property. To use the PDRM for the creation of a PbDCM, the Pattern Structure of the PDRM must be adapted to not contain the *Exactly-once Delivery* Behavior Pattern and the *Information Obscurity* Behavior Pattern.

In the fourth variation, the *region* Property of the AWS Component is not set. As the *region* Property of the AWS Component is set to the wildcard value “*" in the PDRM depicted in Figure 3.2, i.e., any non-empty value is allowed for this Property, the PDRM is not applicable to the subgraph. Finally, in the fifth variation, the SQS Queue Component has been annotated with the *Timeout-based Delivery* [FLR+14] Behavior Pattern. This Behavior Pattern may, for example, have been added during a previous iteration of the pattern detection process. As the subgraph now is not annotated with the same Behavior Patterns as the Executable Structure of the PDRM depicted in Figure 3.2, the PDRM is not applicable to the subgraph.

Algorithm 3.1 describes the steps necessary to adapt the Pattern Structure of a PDRM according to the defined Behavior Pattern Mappings and the Properties specified in the matching subgraph of a deployment model. The algorithm is executed after the matching subgraph of a deployment model was replaced with the Pattern Structure of a PDRM by utilizing the algorithms defined for the pattern refinement process. Therefore, the given deployment model now already contains the Pattern Structure and all external Relations were redirected according to the defined Relation Mappings. However, as the algorithms defined for the pattern refinement process do not consider Behavior Pattern Mappings, the inserted Pattern Structure needs to be adapted to provide a comprehensive view of the detected patterns by (i) adding Behavior Patterns to staying elements if all Behavior Pattern Mappings hold and (ii) removing Behavior Patterns from inserted elements if a Behavior Pattern Mapping does not hold. The algorithm receives the selected *pdrm*, the given deployment model t, and the selected Subgraph Mapping *sm* as input. In Line 1, the algorithm iterates over all Element Mappings of *sm*. The algorithm then considers the two described cases: (i) the Structure Element $se_t \in SE_t$ is a staying element (Lines 2 to 12) or (ii) the Structure Element se_t was replaced by a Structure Element of the Pattern Structure ps_{pdrm} (Lines 13 to 22).

A Structure Element se_t is a staying element if it was not replaced by a Structure Element of the Pattern Structure ps_{pdrm}. A Structure Element se_t is determined to be a staying element in Line 2 if there exists a Stay Mapping in S_{pdrm} for the Structure Element $se_{es} \in SE_{es_{pdrm}}$ which matches se_t. If se_t is a staying element and, therefore, was not replaced, the Behavior Patterns of the inserted Pattern Structure ps_{pdrm} need to be added to it. First, all Behavior Patterns of the placeholder Model Node $mn_{ps} \in MN_{ps_{pdrm}}$ which is the element corresponding to the Structure Element se_{es} in the considered Stay Mapping are added, regardless of whether all Behavior Pattern Mappings are fulfilled or not (Lines 4 and 5). This ensures that also Behavior Patterns without Behavior Pattern Mappings are added to the staying element se_t. Next, all Behavior Pattern Mappings for the Structure Element se_{es} and the Model Node mn_{ps} are determined (Line 6). In Line 7 and 8, it is checked if any of the Behavior Pattern Mappings are not fulfilled for the staying element se_t according to the same conditions introduced in Section 3.3.2 for the compatibility of Properties. A Behavior Pattern Mapping is not fulfilled if the value of its specified Property pr_x is not the Empty Word ε and there exists no Property $pr_i \in properties_t(se_t)$ with (i) the same key and value as pr_x or

Algorithm 3.1 `checkBehaviorPatternMappings`($pdrm \in PDRM, t \in \mathcal{T}, sm \in SM_{es_{pdrm},t}$)

```
1:  for all (se_es, se_t) ∈ sm do
2:    if ∃(se_es, mn_ps) ∈ S_pdrm then
3:      // Add Behavior Patterns to staying element if all Behavior Pattern Mappings hold
4:      BP_t := BP_t ∪ annotations_{ps_pdrm}(mn_ps)
5:      addAnnotations(se_t, annotations_{ps_pdrm}(mn_ps))
6:      for all (se_es, pr_x, mn_ps, bp_y) ∈ BPM_pdrm do
7:        if π2(pr_x) ≠ ε ∧ ∄pr_i ∈ properties_t(se_t) : (π1(pr_x) = π1(pr_i)
8:          ∧ (π2(pr_x) = π2(pr_i) ∨ (π2(pr_x) = " * " ∧ π2(pr_i) ≠ ε))) then
9:          BP_t := BP_t \ {bp_y}
10:         removeAnnotation(se_t, bp_y)
11:       end if
12:     end for
13:   else
14:     // Remove Behavior Patterns of added element if not all Behavior Pattern Mappings hold
15:     for all (se_es, pr_x, se_ps, bp_y) ∈ BPM_pdrm do
16:       if π2(pr_x) ≠ ε ∧ ∄pr_i ∈ properties_t(se_t) : (π1(pr_x) = π1(pr_i)
17:         ∧ (π2(pr_x) = π2(pr_i) ∨ (π2(pr_x) = " * " ∧ π2(pr_i) ≠ ε))) then
18:         BP_t := BP_t \ {bp_y}
19:         removeAnnotation(se_ps, bp_y)
20:       end if
21:     end for
22:   end if
23: end for
```

(ii) with the same key as pr_x while the value of pr_x is set to the wildcard value "*" and the value of pr_i is not set to the Empty Word ε (Lines 7 and 8). If any Behavior Pattern Mapping is not fulfilled for the staying element se_t, the corresponding Behavior Pattern is removed again (Lines 9 and 10).

A Structure Element se_t was not a staying element if (i) it was replaced by a Structure Element of the Pattern Structure ps_{pdrm} or (ii) se_t was not a Model Node as Stay Mappings only apply to Model Nodes. In this case, the Behavior Patterns not realized by the replaced subgraph need to be removed from the inserted Pattern Structure ps_{pdrm} (Lines 13 to 22). The removal of Behavior Patterns from the inserted Pattern Structure ps_{pdrm} is performed analogously to the removal of Behavior Patterns from a staying element. First, all Behavior Pattern Mappings for the Structure Element se_{es} are determined (Line 15). Next, it is checked if any of the Behavior Pattern Mappings were not fulfilled by the replaced Structure Element se_t (Lines 16 and 17). If any Behavior Pattern Mapping was not fulfilled by the replaced Structure Element se_t, the corresponding Behavior Pattern is removed from the Structure Element $se_{ps} \in SE_{ps_{pdrm}}$ which replaced se_t (Lines 18 and 19).

Finally, two adaptations need to be made to the algorithms defined for the pattern refinement process to enable their reuse for the pattern detection process. As described, the algorithms defined for the pattern refinement process are executed before Algorithm 3.1 to replace the matching subgraph of a deployment model with the Pattern Structure of a PDRM. The replacement procedure of the pattern refinement process first adds all Structure Elements of the structure to be inserted to t, i.e., in case of the pattern detection process $SE_t := SE_t \cup SE_{ps_{pdrm}}$ [HBF+20]. As this does not include the Behavior Patterns required by the pattern detection process, the replacement procedure is adapted to add all

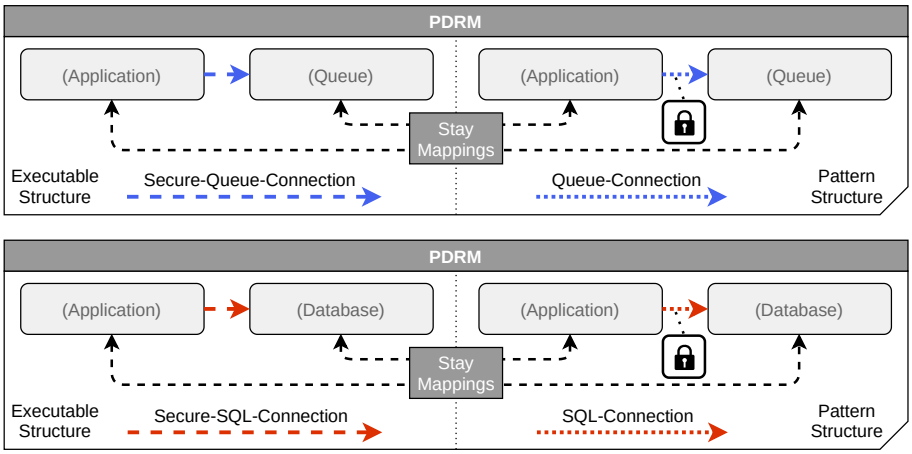

Figure 3.4: PDRMs for detecting the *Secure Channel* [SFH+06] Behavior Pattern.

Model Elements of the Pattern Structure of a PDRM to t. Furthermore, as the algorithms defined for the pattern refinement process are intended to refine patterns and not detect them, Behavior Patterns are removed after each iteration of the replacement procedure [HBF+20]. This interferes with the steps described in Algorithm 3.1 as the inserted Pattern Structure of a PDRM must be adapted according to the defined Behavior Pattern Mappings and not according to the requirements of the pattern refinement process. The replacement procedure is therefore adapted for the pattern detection process to not remove Behavior Patterns after an iteration.

It should be noted that the conditions presented in Section 3.3.2 and the algorithm described in this subsection can not only be used to detect Behavior Patterns in Components but also Relations. For example, Figure 3.4 depicts two PDRMs which can be used to detect the *Secure Channel* [SFH+06] Behavior Pattern realized by the *Secure-Queue-Connection* Relations and the *Secure-SQL-Connection* Relation of the running example. At the top of Figure 3.4, a PDRM which can be used to detect the *Secure Channel* Behavior Pattern on a *Secure-Queue-Connection* Relation is depicted. At the bottom of Figure 3.4, a PDRM which can be used to detect the *Secure Channel* Behavior Pattern on a *Secure-SQL-Connection* Relation is depicted. Both PDRMs define Stay Mappings for all contained Model Nodes. Therefore, when the PDRMs are applied to a deployment model, only the respective Relation will be replaced as it is the only Structure Element not associated with a Stay Mapping. Furthermore, the Model Nodes are not specific Components or Component Patterns but abstract supertypes. As a result, the PDRM depicted at the top of Figure 3.4 is applicable to all *Secure-Queue-Connection* Relations originating from a Model Node derived from the *Application* type and targeted at a Model Node derived from the *Queue* type. The PDRM depicted at the bottom of Figure 3.4 is applicable to all *Secure-SQL-Connection* Relations originating from a Model Node derived from the *Application* type and targeted at a Model Node derived from the *Database* type.

3.4 Detection of Additional Component Patterns

This section presents the second phase of the pattern detection process which detects additional Component Patterns. Section 3.4.1 provides an overview of the phase and describes how Component Pattern Mappings are used to support the detection of patterns. Section 3.4.2 formally defines the conditions which must be fulfilled by a PDRM to match an interim PbDCM created by the first phase. Section 3.4.3 introduces an algorithm which utilizes the detected patterns to further adapt a created interim PbDCM.

3.4.1 Overview

Analogously to the first phase of the pattern detection process, the second phase is also executed in a semi-automatic and iterative fashion with each iteration consisting of the three steps described in Section 3.1. During each iteration of the second phase, additional Component Patterns can be detected by comparing the Executable Structures of the available PDRMs to the subgraphs of the interim PbDCM created by the first phase. If the Executable Structure of a PDRM matches a subgraph of the interim PbDCM created by the first phase, then the Pattern Structure of the PDRM represents the detected patterns. The PDRM can therefore be used to replace the matching subgraph with the Pattern Structure and, thus, create a new interim PbDCM which contains additional Component Patterns.

To detect additional Component Patterns, the conditions that must be met for a PDRM to match an interim PbDCM are relaxed. Specifically, the conditions are relaxed by the following adaptations: (i) all Properties are ignored and (ii) two elements are compatible if they share a common supertype or (iii) two elements are compatible if they realize the same Component Pattern. Two elements can be identified to realize the same Component Pattern by the Component Pattern Mappings they are associated with. In contrast, the conditions introduced in Section 3.3.2 for the first phase of the pattern detection process (i) require all Properties not associated with a Behavior Pattern Mapping to match, (ii) only consider two elements to be compatible if the type of one is equal to or derived from the other, and (iii) do not actively consider Component Patterns during the compatibility check. While the relaxed conditions can lead to the detection of additional Component Patterns, the detection of Behavior Patterns cannot be considered during this phase and Property Mappings cannot be applied to transfer the values of Properties to the respective pattern elements.

Figure 3.5 visualizes the relaxed conditions using two PDRMs and a subgraph of the running example described in Section 1.1. The PDRM depicted on the left side of Figure 3.5 is based on the CBPRM described in Section 2.3.2 and will be referred to as *PDRM 1*. The Executable Structure of PDRM 1 consists of a Java 11 App Component which is executed by a Tomcat 10 Component running on an Ubuntu 20.04 Component which is configured with fixed values for its RAM and storage is and hosted on an OpenStack Victoria Component. The Pattern Structure of PDRM 1 consists of a Java 11 App Component hosted on an *Execution Environment* [FLR+14] Component Pattern which is in turn hosted on a *Private Cloud* [FLR+14] Component Pattern. The Behavior Patterns which may be realized by the Executable Structure of PDRM 1 are omitted for brevity. As the Tomcat 10 Component realizes the *Execution Environment* Component Pattern, the two Model Nodes are connected by a Component Pattern Mapping in PDRM 1. As the OpenStack Victoria Component realizes the *Private Cloud* Component Pattern, the two Model Nodes are also connected by a Component Pattern Mapping in PDRM 1. The PDRM depicted on the right side of

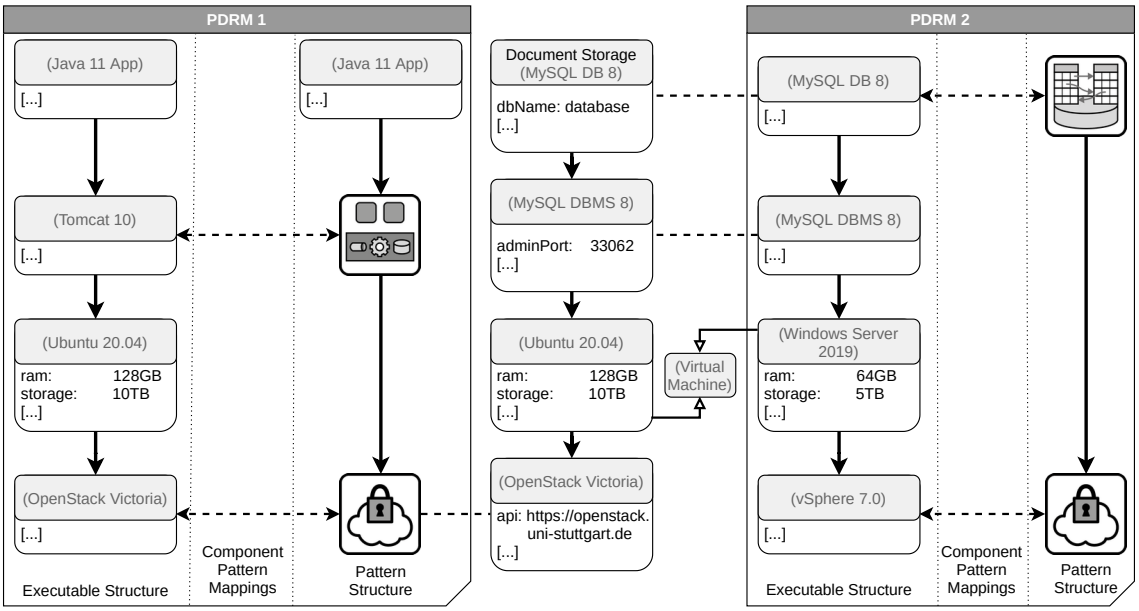

Figure 3.5: Matching procedure using Component Pattern Mappings.

Figure 3.5 contains the elements and patterns already discussed in Section 2.3 and will be referred to as *PDRM 2*. The Executable Structure of PDRM 2 consists of a MySQL DB 8 Component which is hosted on a MySQL DBMS 8 Component running on a Windows Server[1] 2019 Component which is in turn hosted on a vSphere[2] 7.0 Component. The Pattern Structure of PDRM 2 consists of a *Relational Database* [FLR+14] Component Pattern hosted on a *Private Cloud* Component Pattern. As the MySQL DB 8 Component realizes the *Relational Database* Component Pattern, the two Model Nodes are connected by a Component Pattern Mapping in PDRM 2. As the vSphere 7.0 Component realizes the *Private Cloud* Component Pattern, the two Model Nodes are also connected by a Component Pattern Mapping in PDRM 2.

Although the subgraph shown at the center of Figure 3.5 and the Executable Structure of PDRM 2 realize the same Component Patterns, they are not compatible according to the conditions introduced in Section 3.3.2 as the Ubuntu 20.04 Component and the Windows Server 2019 Component as well as the OpenStack Victoria Component and the vSphere 7.0 Component are not of the same type nor derived from each other. Furthermore, the Properties of the Ubuntu 20.04 Component and the Windows Server 2019 Component do not match and are not associated with Behavior Pattern Mappings in this example, i.e., cannot be ignored by the conditions introduced in Section 3.3.2. However, the Ubuntu 20.04 Component and the Windows Server 2019 Component are both derived from the *Virtual Machine* type as visualized in Figure 3.5. Additionally, the OpenStack Victoria Component in PDRM 1 is associated with the *Private Cloud* Component Pattern by the described Component Pattern Mapping which is the same Component Pattern the vSphere 7.0 Component is associated with in PDRM 2. Therefore, the Pattern Structure of PDRM 2 should be considered representative of the patterns realized by the subgraph depicted at the center of Figure 3.5.

[1] https://www.microsoft.com/windows-server/
[2] https://www.vmware.com/products/vsphere.html/

3.4.2 Pattern Detection

In this subsection, the relaxed conditions for the detection of additional Component Patterns in an interim PbDCM created by the first phase of the pattern detection process are formally defined. Analogously to the conditions defined in Section 3.3.2, the Executable Structure of a PDRM matches a subgraph of an interim PbDCM if there exists a structural isomorphism between the two and all Structure Elements are compatible. While the Subgraph Mappings and Element Mappings used in Section 3.3.2 to describe structural isomorphisms are also used in this phase, a different Compatibility Operator is employed to determine if the Structure Elements of an Element Mapping are compatible. This Compatibility Operator is defined in Equation (3.5).

To facilitate the relaxed conditions defined by the Compatibility Operator of the second phase, the Compatibility Operator "$\overrightarrow{\approx}_{bcp}$" defined in Section 3.3.2 for the first phase is adapted to ignore all Properties and is introduced as the Compatibility Operator "$\overrightarrow{\approx}_{bcp\backslash pr}$". While the detection of additional Behavior Patterns is not considered during this phase of the pattern detection process, the Behavior Patterns detected in the first phase need to be maintained. Therefore, the Compatibility Operator "$\overrightarrow{\approx}_{bcp\backslash pr}$" still requires two Structure Elements to be compatible according to the Compatibility Operator "$\overrightarrow{\approx}_{bp}$" to prevent the existing Behavior Patterns from being overwritten. The Compatibility Operator "$\overrightarrow{\approx}_{bcp\backslash pr}$" is defined as follows:

$$\begin{aligned} se_{es} \overrightarrow{\approx}_{bcp\backslash pr} se_t \Leftrightarrow type_{es_{pdrm}}(se_{es}) \in supertypes_t(se_t) \\ \wedge\ se_{es} \overrightarrow{\approx}_{bp} se_t \end{aligned} \tag{3.4}$$

Equation (3.5) defines the Compatibility Operator "$\overrightarrow{\approx}_{cp}$" used in the second phase of the pattern detection process where the acronym *cp* indicates that only the detection of Component Patterns is considered. A Structure Element $se_{es} \in SE_{es_{pdrm}}$ is compatible with a Structure Element $se_t \in SE_t$ according to the Compatibility Operator "$\overrightarrow{\approx}_{cp}$" if (i) se_{es} is compatible with se_t according to the Compatibility Operator "$\overrightarrow{\approx}_{bcp\backslash pr}$", or (ii) se_{es} is not associated with a Stay Mapping, is not associated with a Component Pattern Mapping, shares a common supertype with se_t, and is compatible with se_t according to the Compatibility Operator "$\overrightarrow{\approx}_{bp}$", or (iii) se_{es} is not associated with a Stay Mapping and another PDRM $pdrm_x$ exists which contains a Model Node $mn_{es_x} \in MN_{es_{pdrm_x}}$ that is compatible with se_{es} according to the Compatibility Operator "$\overrightarrow{\approx}_{bcp\backslash pr}$" and realizes the same Component Pattern. The Compatibility Operator "$\overrightarrow{\approx}_{cp}$" is defined as follows:

$$\begin{aligned} se_{es} \overrightarrow{\approx}_{cp} se_t \Leftrightarrow se_{es} \overrightarrow{\approx}_{bcp\backslash pr} se_t \\ \vee\ \big(\nexists(se_{es}, mn_{ps}) \in S_{pdrm} \\ \wedge\ (se_{es} \overrightarrow{\approx}_{\cap} se_t \vee se_{es} \overrightarrow{\approx}_{cpm} se_t)\big) \end{aligned} \tag{3.5}$$

The Compatibility Operator "$\overrightarrow{\approx}_{\cap}$" used in Equation (3.5) considers the compatibility of two Structure Elements based on their supertypes. Two Structure Elements are considered to be compatible if they share a common supertype. To prevent the detection of Pattern Structures that are not representative of a given subgraph, Structure Elements which are associated with a Stay Mapping or a Component Pattern Mapping are not considered by the Compatibility Operator "$\overrightarrow{\approx}_{\cap}$". Furthermore, the Behavior Patterns annotated at se_{es} and se_t must be compatible according to the Compatibility Operator "$\overrightarrow{\approx}_{bp}$"

to prevent the Behavior Patterns detected during the first phase from being overwritten. The Compatibility Operator "$\overrightarrow{\approx}_{\cap}$" is defined as follows:

$$\begin{aligned} se_{es} \overrightarrow{\approx}_{\cap} se_t \Leftrightarrow \nexists(se_{es}, mn_{ps}) \in CPM_{pdrm} \\ \wedge\ supertypes_{es_{pdrm}}(se_{es}) \cap supertypes_t(se_t) \neq \emptyset \\ \wedge\ se_{es} \overrightarrow{\approx}_{bp} se_t \end{aligned} \tag{3.6}$$

The Compatibility Operator "$\overrightarrow{\approx}_{cpm}$" used in Equation (3.5) considers the compatibility of two Structure Elements based on the Component Patterns they realize. Two Structure Elements are considered to be compatible if they realize the same Component Pattern. To ensure that two Structure Elements realize the same Component Pattern, the set of one-to-one Component Pattern Mappings $CPM^{1:1}_{pdrm} \subseteq CPM_{pdrm}$ is introduced. The set of one-to-one Component Pattern Mappings $CPM^{1:1}_{pdrm}$ only contains Component Pattern Mappings which define a one-to-one correspondence between two Model Nodes, i.e., for a Model Node mn_{es} and a Model Node mn_{ps} of a one-to-one Component Pattern Mapping $(mn_{es}, mn_{ps}) \in CPM^{1:1}_{pdrm}$ there exists no other Component Pattern Mapping. The Compatibility Operator "$\overrightarrow{\approx}_{cpm}$" utilizes these one-to-one Component Pattern Mappings to determine if two Structure Elements realize the same Component Pattern. As Component Pattern Mappings are only defined for PDRMs, the Component Patterns realized by the Components of a given subgraph cannot be determined directly. Therefore, a PDRM $pdrm_x$ needs to be found which contains a Model Node mn_{es_x} that is compatible with the Structure Element se_t according to the Compatibility Operator "$\overrightarrow{\approx}_{bcp\backslash pr}$" and is associated with a one-to-one Component Pattern Mapping. This one-to-one Component Pattern Mapping can then be used to check if the Structure Element se_t of the given subgraph realizes the same Component Pattern as the Structure Element se_{es}. To prevent the detection of Pattern Structures that are not representative of a given subgraph, Structure Elements which are associated with a Stay Mapping are not considered. The Compatibility Operator "$\overrightarrow{\approx}_{cpm}$" is defined as follows:

$$\begin{aligned} se_{es} \overrightarrow{\approx}_{cpm} se_t \Leftrightarrow \exists pdrm_x \in PDRM \Big(\exists (mn_{es_x}, mn_{ps_x}) \in CPM^{1:1}_{pdrm_x} \\ \big(\exists (se_{es}, mn_{ps}) \in CPM^{1:1}_{pdrm} (mn_{es_x} \overrightarrow{\approx}_{bcp\backslash pr} se_t \\ \wedge\ type_{ps_{pdrm_x}}(mn_{ps_x}) = type_{ps_{pdrm}}(mn_{ps})) \big) \Big) \end{aligned} \tag{3.7}$$

To provide an example, Figure 3.6 depicts three variations of the matching procedure visualized in Figure 3.5. If a Property or type was changed in contrast to the subgraph depicted at the center of Figure 3.5, the respective Property or type is highlighted using bold letters. Below each variation, a check mark or a cross indicates whether the matching procedure visualized in Figure 3.5 is applicable. In the first variation depicted in Figure 3.6, all Properties of the subgraph depicted at the center of Figure 3.5 have been changed. However, as the Compatibility Operator "$\overrightarrow{\approx}_{cp}$" does not consider any Properties, the matching procedure visualized in Figure 3.5 is still applicable. In the second variation, the MySQL DB 8 Component of the subgraph depicted at the center of Figure 3.5 has been changed to a MariaDB 10.5 Component and the MySQL DBMS 8 Component to a MariaDB DBMS 10.5 Component. The MySQL DBMS 8 Component and the MariaDB DBMS 10.5 Component share a common supertype and are therefore compatible according to the Compatibility Operator "$\overrightarrow{\approx}_{\cap}$". However, there is no PDRM available which can be used to identify the Component Pattern realized by the MariaDB 10.5 Component. It can therefore

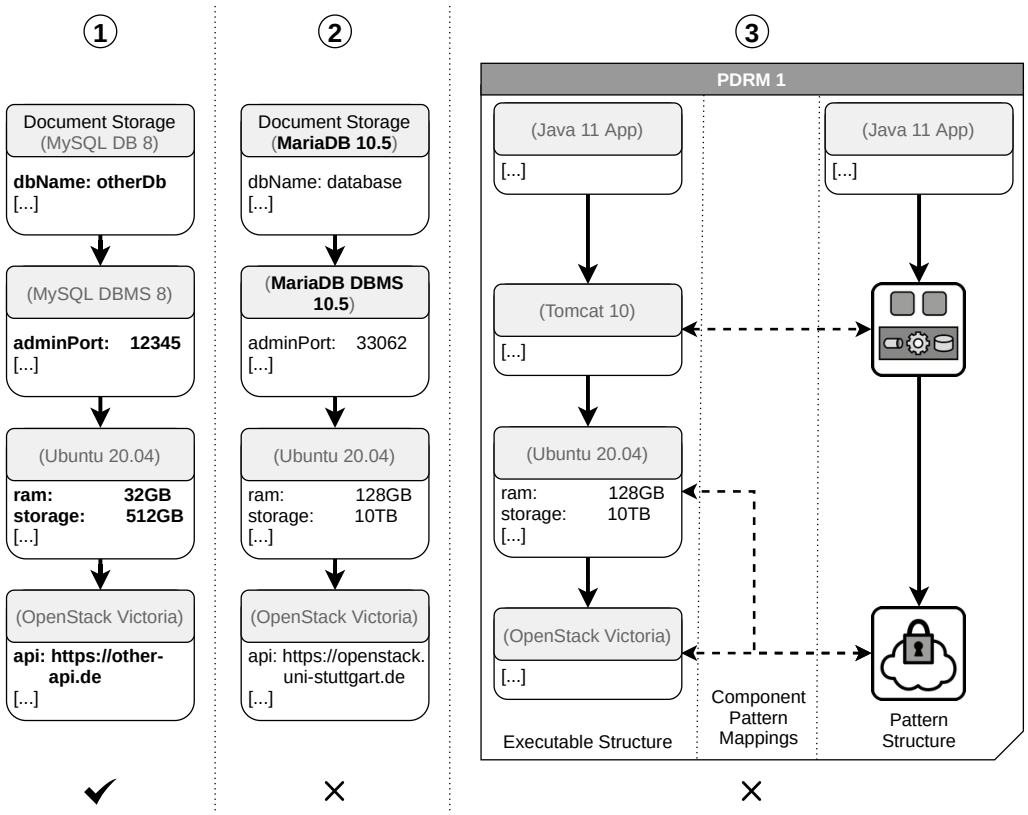

Figure 3.6: Variations of the matching procedure visualized in Figure 3.5.

not be determined if the MySQL DB 8 Component and the MariaDB 10.5 Component realize the same Component Pattern according to the Compatibility Operator "$\vec{\approx}_{cpm}$". As a result, the matching procedure visualized in Figure 3.5 is not applicable. By creating a PDRM that associates a MariaDB 10.5 Component with a *Relational Database* Component Pattern using a one-to-one Component Pattern Mapping, the matching procedure visualized in Figure 3.5 would be applicable. In the third variation, the Component Pattern Mappings specified in the PDRM 1 depicted on the left side of Figure 3.5 have been adapted. The PDRM now specifies that in order to realize the *Private Cloud* Component Pattern, the Ubuntu 20.04 Component as well as the OpenStack Victoria Component need to be present. The *Private Cloud* Component Pattern is therefore not associated with a one-to-one Component Pattern Mapping anymore. As the conditions defined by the Compatibility Operator "$\vec{\approx}_{cpm}$" only consider one-to-one Component Pattern Mappings, the matching procedure visualized in Figure 3.5 is not applicable. By considering more general Component Pattern Mappings, the matching procedure visualized in Figure 3.5 could be determined to be applicable as the Components associated with the *Private Cloud* Component Pattern are both present in the subgraph depicted at the center of Figure 3.5.

3.4.3 Generation of Pattern-based Deployment and Configuration Models

In this subsection, the additional Component Patterns detected based on the relaxed conditions described in Section 3.4.2 are utilized to further adapt an interim PbDCM generated by the first phase of the pattern detection process. Analogously to the first phase of the pattern detection process, the second phase is also executed in a semi-automatic and iterative fashion with each iteration consisting of the three steps described in Section 3.1. During the third step of each iteration, the matching subgraph in the interim PbDCM is replaced with the Pattern Structure of the PDRM selected by a user. To replace a subgraph with the Pattern Structure of a PDRM and, thus, create a new interim PbDCM which contains additional Component Patterns, the algorithms for pattern refinement [HBF+20; HBM+18] are utilized.

However, due to the relaxed conditions of the second phase of the pattern detection process, three adaptations need to be made to a PDRM which was determined to be applicable to a subgraph of an interim PbDCM before it is applied. As the Compatibility Operator "$\overrightarrow{\approx}_{cpm}$" defined in Section 3.4.2 considers Model Nodes from different PDRMs which may contain Relation Mappings specific to their context, the Relation Mappings associated with the considered Model Nodes need to be added to the PDRM which is to be applied to enable the redirection of all external Relations. As Property Mappings are defined for specific Model Nodes and their configuration which may not be equally present in the matching subgraph because of the relaxed conditions, the Property Mappings need to be removed from the PDRM which is to be applied. Finally, as the Behavior Patterns defined in the Pattern Structure of a PDRM are only guaranteed to be realized by the corresponding Executable Structure and Behavior Pattern Mappings are not considered in this phase, the Behavior Patterns defined in the Pattern Structure of the PDRM which is to be applied need to be removed to prevent the detection of patterns that are not realized in the matching subgraph.

To be able to add the Relation Mappings of the PDRMs considered during the compatibility check described in Section 3.4.2, the elements which were determined to be compatible by the Compatibility Operator "$\overrightarrow{\approx}_{cpm}$" must be known. The combinations of such compatible elements determined by the Compatibility Operator "$\overrightarrow{\approx}_{cpm}$" is defined by the set of Component Pattern Candidates $CPC_{pdrm,t}$ for a $pdrm \in PDRM$ based on a given interim PbDCM $t \in \mathcal{T}$. A Component Pattern Candidate $cpc_i \in CPC_{pdrm,t}$ is defined as follows:

$$cpc_i = (mn_{es}, mn_{ps}, prdrm_x, mn_{es_x})$$

Herein, $mn_{es} \in MN_{es_{pdrm}}$ is a Model Node that is not compatible with a Structure Element $se_t \in SE_t$ according to the Compatibility Operator "$\overrightarrow{\approx}_{bcp\backslash pr}$" or the Compatibility Operator "$\overrightarrow{\approx}_{\cap}$" and $mn_{ps} \in MN_{ps_{pdrm}}$ is a Model Node which is associated with mn_{es} by a one-to-one Component Pattern Mapping in $pdrm$. $prdrm_x \in PDRM$ is another PDRM which contains a Model Node $mn_{es_x} \in MN_{es_{pdrm_x}}$ that is compatible with se_t according to the Compatibility Operator "$\overrightarrow{\approx}_{bcp\backslash pr}$" and is associated with a one-to-one Component Pattern Mapping in $pdrm_x$ which specifies a Component Pattern that is of the same type as mn_{ps}.

Algorithm 3.2 formalizes the steps necessary to adapt a PDRM before it is applied. This adapted PDRM is then used as input for the inverted pattern refinement algorithms which replace the matching subgraph with the Pattern Structure of the adapted PDRM. In contrast to the algorithm described in Section 3.3.3 which is executed after the inverted pattern refinement algorithms, Algorithm 3.2 is executed before the inverted pattern refinement algorithms. Again, the inverted

Algorithm 3.2 `adaptPdrm`($pdrm \in PDRM, CPC_{pdrm,t}$)

1: *// Redirect and add Relation Mappings of* $pdrm_x$
2: **for all** $(mn_{es}, mn_{ps}, prdrm_x, mn_{es_x}) \in CPC_{pdrm,t}$ **do**
3: **for all** $rm_y \in RM_{pdrm_x} : \big(\pi_1(rm_y) = mn_{es_x}$
4: $\wedge\ type_{pdrm_x}(\pi_2(rm_y)) = type_{pdrm}(mn_{ps})\big)$ **do**
5: $\pi_1(rm_y) := mn_{es}$
6: $\pi_2(rm_y) := mn_{ps}$
7: $RM_{pdrm} := RM_{pdrm} \cup \{rm_y\}$
8: **end for**
9: **end for**
10: *// Remove Property Mappings of pdrm*
11: $PM_{pdrm} := \emptyset$
12: *// Remove all Behavior Patterns not in the Executable Structure*
13: **for all** $bp_i \in BP_{ps_{pdrm}} : (bp_i \notin BP_{es_{pdrm}})$ **do**
14: $BP_{ps_{pdrm}} := BP_{ps_{pdrm}} \setminus \{bp_i\}$
15: `removeAnnotation`(ps_{pdrm}, bp_i)
16: **end for**

pattern refinement algorithms are adapted to also add the Behavior Patterns of the respective Pattern Structure at the beginning of an iteration and to not remove Behavior Patterns after an iteration as described in Section 3.3.3. Algorithm 3.2 receives a $pdrm \in PDRM$ which is applicable to a subgraph of an interim PbDCM $t \in \mathcal{T}$ and the set of Component Pattern Candidates $CPC_{pdrm,t}$ which was determined during the compatibility check of the Compatibility Operator "$\overrightarrow{\approx}_{cpm}$" as input. First, the Relation Mappings of the PDRMs considered during the compatibility check of the Compatibility Operator "$\overrightarrow{\approx}_{cpm}$" need to be added to *pdrm* to enable the redirection of all external Relations (Lines 1 to 9). This is necessary as a considered $pdrm_x \in PDRM$ may have been developed with different external Relations in mind than *pdrm*. To also redirect such external Relations during the application of *pdrm* and, thus, create a new and valid interim PbDCM, Algorithm 3.2 iterates over all Component Pattern Candidates defined for *pdrm* and t (Line 2). In Line 3 and 4, all Relation Mappings of a $pdrm_x$ which are associated with a Structure Element $mn_{es_x} \in MN_{es_{pdrm_x}}$ of the currently considered Component Pattern Candidate and are associated with a Structure Element which is of the same type as the Model Node $mn_{ps} \in MN_{ps_{pdrm}}$ are determined. Each Relation Mapping of $pdrm_x$ associated with the described Model Nodes is adapted to be associated with the Model Nodes $mn_{es} \in MN_{es_{pdrm}}$ and $mn_{ps} \in MN_{ps_{pdrm}}$ and is then added to the Relation Mappings of *pdrm* (Lines 5 to 7). Additionally, in Line 11, all Property Mappings of *pdrm* are removed as they cannot be guaranteed to be applicable due to the relaxed conditions of this phase. Finally, to maintain the existing Behavior Patterns but avoid adding additional Behavior Patterns which may not be realized by the matching subgraph due to the relaxed conditions of this phase, all Behavior Patterns not present in the Executable Structure of *pdrm* are removed from the Pattern Structure of *pdrm* (Lines 13 to 16).

It should be noted that an adapted PDRM is not guaranteed to be applicable to a subgraph which is to be replaced. For example, if a $pdrm_x$ contains a Relation Mapping which redirects external Relations to a Model Node $mn_{ps_x} \in MN_{ps_{pdrm_x}}$ which is not of the same type as mn_{ps}, the Relation Mapping will not be added to *pdrm* as defined by Line 4 of Algorithm 3.2. Such a missing Relation Mapping may result in the failure to redirect all external Relations when replacing the matching subgraph

with the Pattern Structure of *pdrm*. A similar issue may occur during the pattern refinement process if a CBPRM does not specify all Relation Mappings necessary for the redirection of the present external Relations [HBF+20; HBM+18]. Therefore, if not all external Relations can be redirected, the CBPRM is determined to not be applicable [HBF+20; HBM+18]. As the approach presented in this thesis utilizes the algorithms defined for the refinement of patterns, the same applies during the first and the second phase of the pattern detection process. To be precise, if a PDRM is determined to be compatible with a subgraph according to the conditions described in Section 3.4.2 but the PDRM adapted according to Algorithm 3.2 cannot be used to redirect all external Relations due to missing Relation Mappings, it is ultimately determined to not be applicable.

4 Prototypical Implementation

This chapter presents the prototypical implementation of the approach described in this thesis. Section 4.1 introduces the TOSCA standard [OAS13; OAS20] which can be mapped to the described PbDCM metamodel and was used as a foundation for the implementation of the pattern detection process. Section 4.2 introduces Eclipse Winery [KBBL13] which is a modeling tool for deployment models and was extended to support the approach presented in this thesis.

4.1 Topology and Orchestration Specification for Cloud Applications

The Topology and Orchestration Specification for Cloud Applications (TOSCA) is a standard by the Organization for the Advancement of Structured Information Standards (OASIS) which specifies a metamodel for describing the structure and the management of applications in a provider and technology agnostic way [OAS13; OAS20]. The structure of an application can be described in TOSCA by a *Topology Template* which is part of a *Service Template* [OAS13; OAS20]. As the metamodel of Topology Templates can be mapped to the metamodel of PbDCMs [HBF+20; HBM+18], TOSCA was used as a foundation for the prototypical implementation of the approach described in this thesis.

A Topology Template is a directed graph containing *Node Templates* which represent the Components of an application and *Relationship Templates* which represent the Relations between the Components of the application. Similarly to the metamodel of PbDCMs introduced in Section 2.3.1, Node Templates and Relationship Templates can specify Properties which describe their state or configuration. Node Templates and Relationship Templates are instances of *Node Types* and *Relationship Types*, respectively, which specify the semantics of their instances in a reusable format. Both Node Types and Relationship Types can specify the Properties a respective Template provides. Node Types therefore correspond to Component Types and Relationship Types correspond to Relation Types [HBF+20; HBM+18; OAS13; OAS20].

Component Pattern Types are not represented as separate elements in TOSCA but can be realized by Node Types annotated with a pattern flag. By definition, this also identifies the Node Templates of a Node Type as pattern elements which, therefore, realizes Component Patterns. Component Behavior Patterns as well as Relation Behavior Patterns can be represented by *Policies* in TOSCA. A Policy defines non-functional behavior and is an instance of a *Policy Type* which defines the semantics of a Policy in a similar fashion as Node Types and Relationship Types for Node Templates and Relationship Templates. Again, Policies are not specifically defined as pattern elements but identified as such by annotating the respective Policy Types with a pattern flag. Policy Types therefore realize Component Behavior Pattern Types and Relation Behavior Pattern Types. However, while TOSCA allows the annotation of Node Templates with Policies, this is not the case for Relationship Templates. Consequently, the standard was extended to support the annotation of Relationship Templates with Policies [HBF+20; HBM+18; OAS13; OAS20].

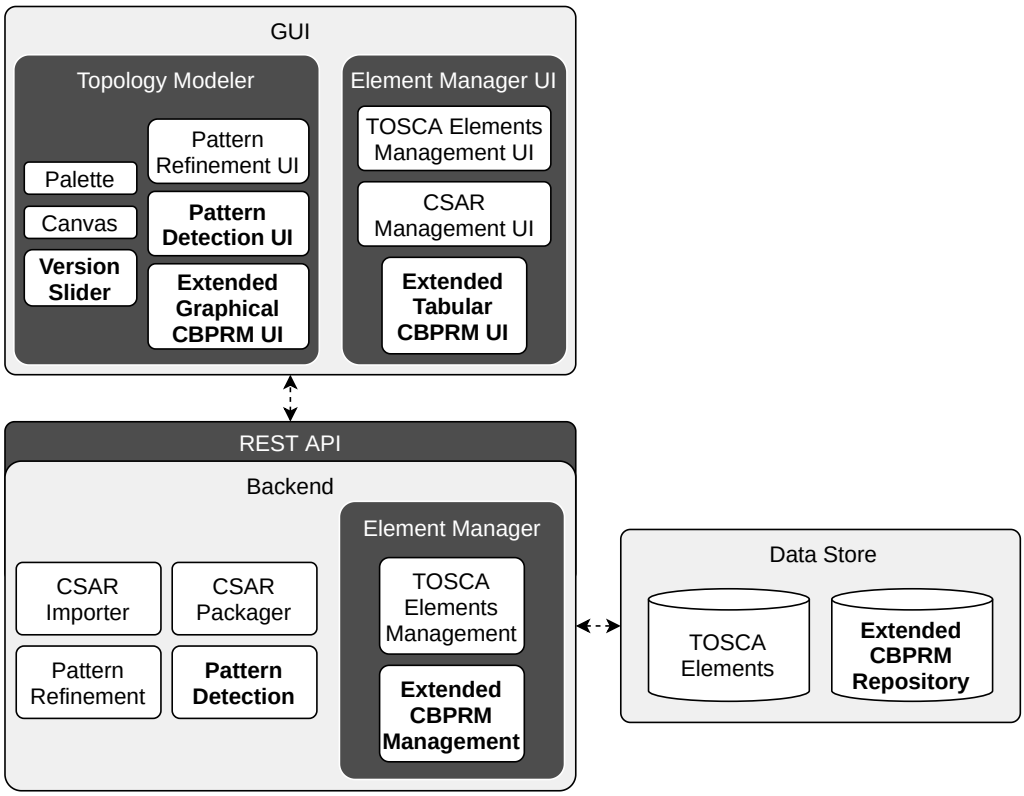

Figure 4.1: Extended components of Eclipse Winery based on [KBBL13].

4.2 Eclipse Winery

Eclipse Winery is a modeling tool for deployment models which was extended to support the proposed pattern detection process. Figure 4.1 depicts the architecture of Eclipse Winery where adapted or newly implemented components are highlighted using bold letters. The architecture of Eclipse Winery consists of a GUI for modeling deployment models and managing the corresponding modeling elements, a backend which contains the necessary business logic, and a data store for persisting the managed elements. The GUI communicates with the backend using a Representational State Transfer (REST) Application Programming Interface (API) [KBBL13].

In the data store of Eclipse Winery, TOSCA elements such as Service Templates and their Topology Templates, Node Types, Relationship Types, and Policy Types are persisted. As the pattern refinement process was implemented in Eclipse Winery, the data store is also used to persist CBPRMs. The persisted TOSCA elements and CBPRMs are accessed by the *Element Manager* which communicates with the *Element Manager User Interface (UI)* via the REST API. In the Element Manager UI of Eclipse Winery, TOSCA elements and CBPRMs can be created, adapted, and deleted [HBF+20; HBM+18; KBBL13].

To model the Topology Template of a Service Template, the *Topology Modeler* of Eclipse Winery can be used. The Topology Modeler provides a *Palette* which lists all available Node Types and a *Canvas* to display the Topology Template of a Service Template. To add a Node Template to the Canvas and, therefore, to a Topology Template, a respective Node Type must be selected from

the Palette. The Node Templates displayed on the Canvas can then be connected via Relationship Templates which is also reflected in the respective Topology Template. Furthermore, the Properties of the modeled Node Templates and Relationship Templates can be edited and Policies of the available Policy Types can be added [KBBL13].

Additionally, Eclipse Winery provides the option to export and import TOSCA elements as Cloud Service Archives (CSARs) which is visualized in Figure 4.1 by the *CSAR Packager*, the *CSAR Importer*, and the *CSAR Management UI*. A CSAR is a self-contained archive which can include the deployment model of an application in the form of a Service Template and all elements associated with it. In this case, the exported archive can be processed by a compliant TOSCA runtime to execute the deployment. A CSAR can also contain TOSCA elements without including a Service Template. In this case, the archive can be imported, e.g., to make the contained elements available in the current environment [KBBL13; OAS13; OAS20].

To enable the pattern detection process, multiple components of Eclipse Winery were extended and new components were implemented. First, the CBPRMs defined for the pattern refinement process were extended to reflect the PDRMs defined in this thesis. The data store depicted at the bottom right side of Figure 4.1 therefore contains the *Extended CBPRM Repository* reflecting the described PDRM Repository. Additionally, the Element Manager of the backend was adapted to support the extended CBPRMs. Next, the pattern detection process presented in this thesis was implemented in the backend of Eclipse Winery which also contains the implementation of the pattern refinement process. This enabled the reuse of the pattern refinement implementation for the pattern detection process. Furthermore, Eclipse Winery provides two means for modeling CBPRMs: (i) the *Tabular CBPRM UI* and (ii) the *Graphical CBPRM UI*. Both the Tabular CBPRM UI and the Graphical CBPRM UI were adapted to support the extended CBPRMs required for the pattern detection process. Analogously to the *Pattern Refinement UI*, the *Pattern Detection UI* was implemented in the Topology Modeler of Eclipse Winery. Finally, a version slider providing a coherent view of the versions of a deployment model was implemented in the Topology Modeler as well. The details of the implementation of the extended CBPRMs, the pattern detection process, and the version slider will be described in the remainder of this section.

4.2.1 Implementation of Pattern Detection and Refinement Models

As previously described, the CBPRMs implemented in Eclipse Winery for the pattern refinement process were extended instead of implementing the introduced PDRMs. By extending the implemented CBPRMs, the existing code was reused and reimplementation was avoided. As a result, in the implementation, the Executable Structure of a PDRM is referred to as Refinement Structure and the Pattern Structure of a PDRM is referred to as Detector. The semantics of these elements, however, remain the same in the context of the pattern detection process, i.e., the Refinement Structure is used to determine matching subgraphs in an executable deployment model and the Detector represents the detected patterns and is used to replace the matching subgraph. Furthermore, the Property Mappings [WBH+20] of PDRMs are referred to as *Attribute Mappings*. The introduced Behavior Pattern Mappings and Component Pattern Mappings were implemented as an extension to the CBPRMs implemented in Eclipse Winery. In the remainder of this chapter, PDRMs will be referred to as *extended CBPRMs*, the Executable Structure of a PDRM as Refinement Structure, the Pattern Structure of a PDRM as Detector, and the Property Mappings of a PDRM as Attribute Mappings to enable a description consistent with the implementation.

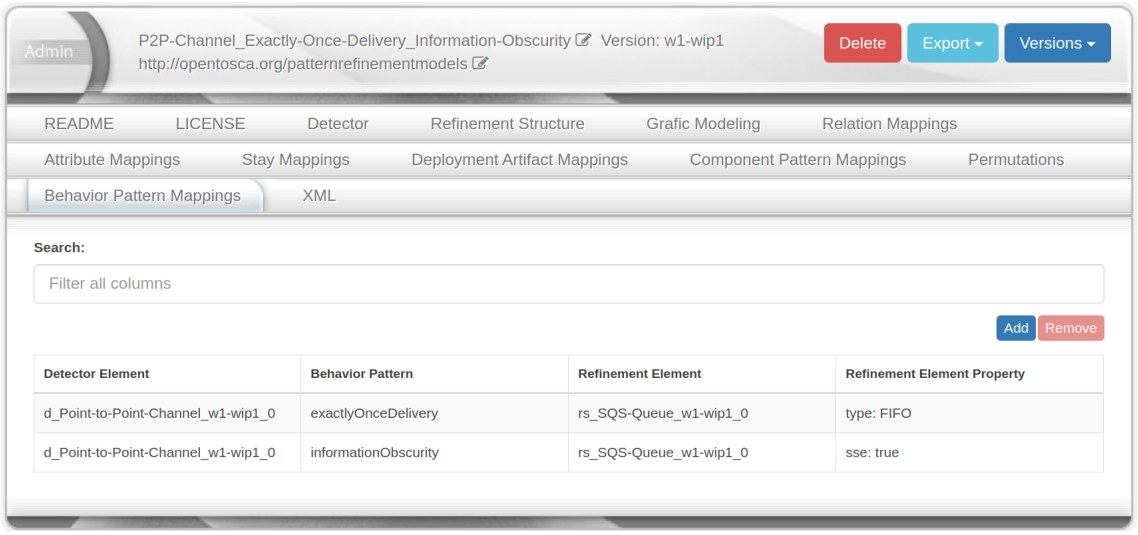

Figure 4.2: Extended tabular CBPRM view of the PDRM depicted in Figure 3.2.

As described in Section 4.2, Eclipse Winery provides two means for modeling CBPRMs. Figure 4.2 depicts the extended tabular CBPRM view showing the PDRM described in Section 3.3.1. The three horizontal bars at the top of Figure 4.2 display the options for and elements of the extended CBPRM. The Detector and the Refinement Structure of the extended CBPRM can be edited individually in the Topology Modeler of Eclipse Winery. Furthermore, all mappings of the extended CBPRM can be displayed in a tabular form. Figure 4.2 currently displays the two described Behavior Pattern Mappings of the extended CBPRM.

In addition to the tabular CBPRM view, the graphical view for modeling CBPRMs was extended in the course of this thesis as well. Figure 4.3 depicts the graphical view of the extended CBPRM shown in Figure 4.2. The Detector of the extended CBPRM is depicted on the left side of Figure 4.3. The Refinement Structure is shown on the right side of Figure 4.3. Between the two, the Behavior Pattern Mappings and the Attribute Mapping which were discussed in Section 3.3.1 are visualized. Furthermore, three Component Pattern Mappings between the Detector and the Refinement Structure of the extended CBPRM are visualized.

4.2.2 Implementation of the Pattern Detection Process

The pattern detection process can be used by opening the Topology Template of a Service Template in the Topology Modeler of Eclipse Winery. As previously described, the Topology Modeler displays all elements of a Topology Template including its Node Templates, Relationship Templates, the defined Properties, and the annotated Policies. To show the patterns detected in a Topology Template, a sidebar was implemented which lists all extended CBPRMs applicable to a subgraph of the Topology Template. Figure 4.4 depicts this sidebar for an exemplary Topology Template.

In Figure 4.4, each extended CBPRM which is applicable to a subgraph of the Topology Template is identified by a name. The Detector of a listed extended CBPRMs can be viewed in the Topology Modeler of Eclipse Winery which, therefore, enables viewing the respective detected patterns. The sidebar is also used to select an extended CBPRM for application which replaces the matching

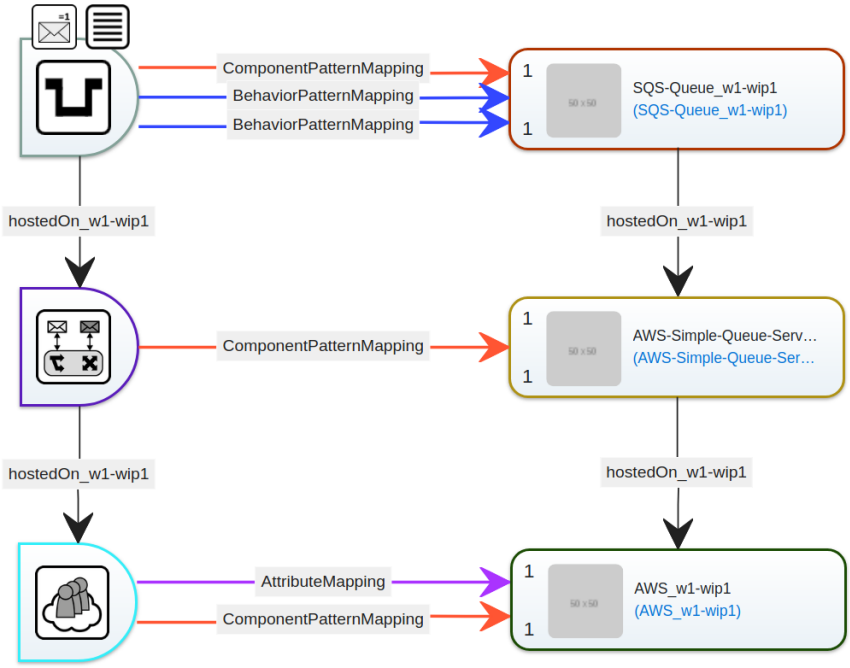

Figure 4.3: Extended graphical CBPRM view of the PDRM depicted in Figure 3.2.

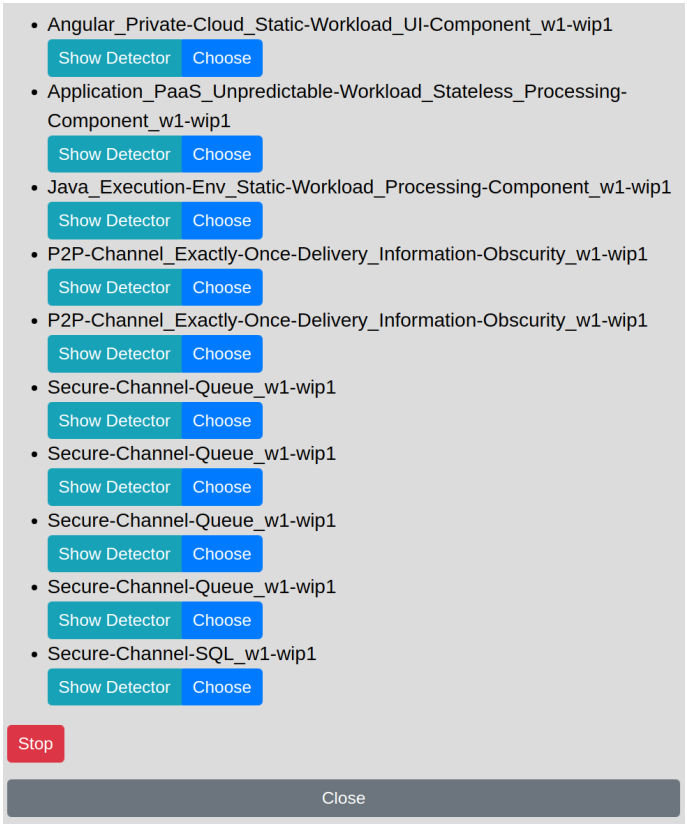

Figure 4.4: Pattern detection sidebar listing matching extended CBPRMs.

Figure 4.5: Version slider showing the versions of a Service Template.

subgraph of the Topology Template with the respective Detector and creates an interim PbDCM. After the application of an extended CBPRM, the sidebar is refreshed for the next iteration to list all extended CBPRMs which apply to the adapted Topology Template or, in other words, interim PbDCM. The pattern detection process stops if a user does not choose to continue with the next iteration or no more matching extended CBPRMs can be determined. When the pattern detection process stops, the adapted Topology Template representing the final PbDCM is opened in a new window of the Topology Modeler.

4.2.3 Version Slider

Harzenetter [Har18] introduces a versioning approach for TOSCA elements which proposes appending a string representing the version of an element to the Identifier (ID) of the element. The approach was implemented in Eclipse Winery and applies to, for example, Service Templates, Node Types, Relationship Types, and Policy Types. As the approach applies to Service Templates, it also indirectly applies to the contained Topology Templates which are used both by the implementation of the pattern detection process and the implementation of the pattern refinement process to represent PbDCMs and executable deployment models. The combination of the pattern detection process and the pattern refinement process could enable the following scenario: (i) an executable deployment model is created for an application, (ii) a PbDCM for the created deployment model is generated using the approach described in this thesis, (iii) the generated PbDCM is adapted to reflect new design decisions which have emerged for the application, and finally (iv) the adapted PbDCM is refined to an executable deployment model which realizes the new design decisions by using the pattern refinement process. During each of these four steps at least one new version is created. While every single version is a standalone Service Template, the different versions are still part of the same versioning history. This versioning history can quickly grow to unmanageable proportions which complicates the comparison of individual versions and complicates comprehending the overall development history. To mitigate these issues, a version slider was developed in the course of this thesis which is depicted in Figure 4.5.

The version slider depicted in Figure 4.5 was implemented in the Topology Modeler of Eclipse Winery which displays the Topology Template of a Service Template corresponding to the currently selected version. Figure 4.5 shows a similar scenario as the scenario which was previously described. From left to right, the version slider visualizes the individual versions of the development history. The first version which is identified by the string `w1-wip1` represents the initial executable deployment model. The second version represents the PbDCM generated for the deployment model. The version of a Service Template generated by the pattern detection process consists of the version of the given Service Template and the substring `detected`. Therefore, the second version on the version slider starts with the substring `w1-wip1-detecte`. The third version which is also the version currently selected in Figure 4.5 represents the adapted PbDCM. The fourth and final version represents the executable deployment model which was generated by the pattern refinement process based on the

adapted PbDCM. The version of a Service Template generated by the pattern refinement process consists of the version of the given Service Template and the substring `refined`. Therefore, the fourth version is identified by the string `w1-wip1-detected-w1-wip2-refined-w1-wip1`. The four described versions can be viewed in place, that is without opening a new window or reloading the current one, by using the version slider. After selecting a version, it can be opened in a new window for further actions. The implemented version slider therefore eases the process of comparing different versions and provides a coherent view on the development history.

5 Validation

In this chapter, the feasibility of the presented approach is validated. Section 5.1 presents a case study based on the described running example. Section 5.2 discusses the limitations of the approach presented in this thesis.

5.1 Case Study

In this section, the presented approach is validated by conducting a case study. The case study is based on the running example introduced in Section 1.1. To generate a PbDCM for the deployment model described by the running example, the PDRMs discussed throughout this thesis are utilized. The remaining PDRMs required for the pattern detection process will be presented in the following. For the sake of brevity, most of the mappings not essential to the understanding of the case study are omitted. The patterns used in the PDRMs and the generated PbDCM are taken from the Cloud Computing Patterns [FLR+14], the Enterprise Integration Patterns [HW04], and the Security Patterns [SFH+06].

Figure 5.1 depicts two additional PDRMs required to conduct the case study. The left side of Figure 5.1 depicts a PDRM applicable to the subgraph of the *Document Frontend* in the running example. The Angular 11 Web App Component is associated with a Stay Mapping and is present in the Executable Structure as well as the Pattern Structure. As the Tomcat 10 Component of the Executable Structure is used to execute the Angular 11 Web App Component, the Tomcat 10 Component realizes the *Execution Environment* [FLR+14] Component Pattern and is therefore represented by this element in the Pattern Structure of the PDRM. The OpenStack Victoria Component of the Executable Structure realizes the *Private Cloud* [FLR+14] Component Pattern. The *Static Workload* [FLR+14] Behavior Pattern annotated at the Angular 11 Web App Component of the Pattern Structure is realized if the RAM and storage of the Ubuntu 20.04 Component are set as specified by the Behavior Pattern Mappings. The *User Interface Component* [FLR+14] Behavior Pattern annotated at the Angular 11 Web App Component of the Pattern Structure is not associated with any Behavior Pattern Mappings and is therefore not dependent on any Properties to be realized. Instead, the *User Interface Component* Behavior Pattern is dependent on the complete Executable Structure of the PDRM. The informal rationale behind the decision to annotate the Pattern Structure with the *User Interface Component* Behavior Pattern is that Angular applications are often used to implement UIs. As the pattern detection process is executed in a semi-automatic fashion, a user has the option to discard PDRMs which do not apply to his or her particular use case.

The right side of Figure 5.1 depicts a PDRM applicable to the subgraph of the *Document Processor* in the running example. The Java 11 App Component is associated with a Stay Mapping and is present in the Executable Structure as well as the Pattern Structure. As AWS Elastic Beanstalk is a managed service, the Elastic Beanstalk Worker Environment Component and the AWS Elastic

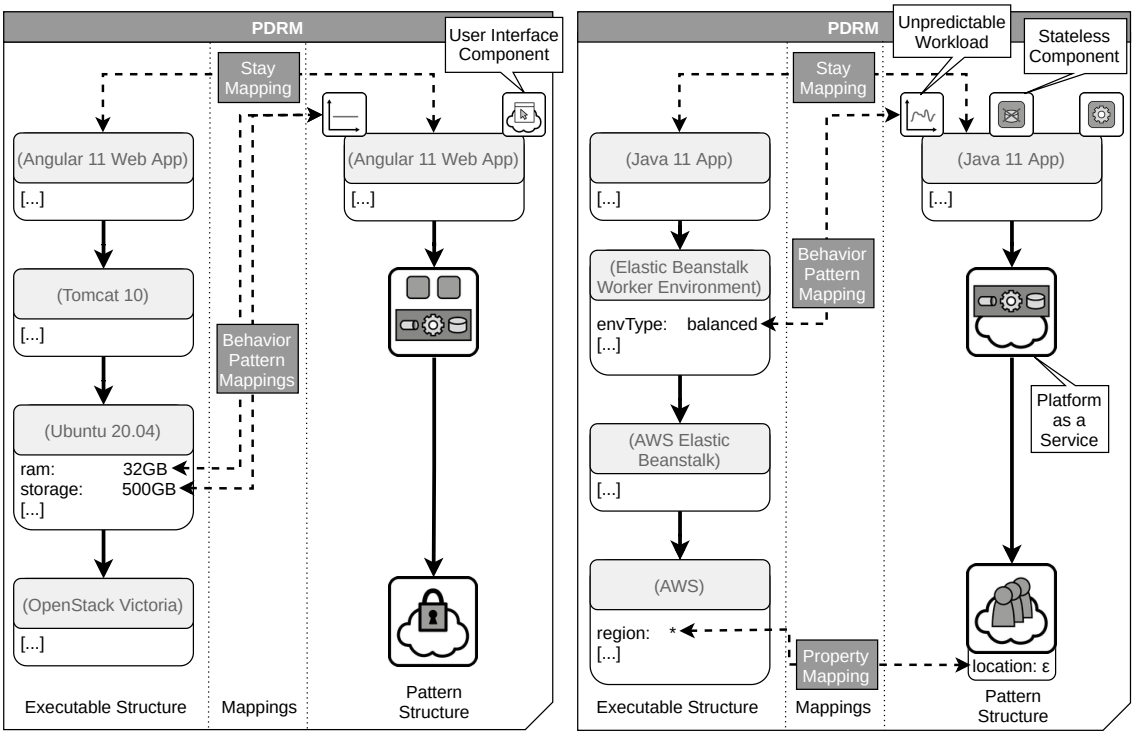

Figure 5.1: Additional PDRMs required for the case study.

Beanstalk Component of the Executable Structure realize the *Platform as a Service* [FLR+14] Component Pattern and are therefore represented by this element in the Pattern Structure of the PDRM. The AWS Component of the Executable Structure realizes the *Public Cloud* [FLR+14] Component Pattern. The *Unpredictable Workload* [FLR+14] Behavior Pattern annotated at the Java 11 App Component of the Pattern Structure is realized if the *envType* Property of the Elastic Beanstalk Worker Environment Component is set as specified by the Behavior Pattern Mapping. The *Stateless Component* [FLR+14] Behavior Pattern and the *Processing Component* [FLR+14] Behavior Pattern annotated at the Java 11 App Component of the Pattern Structure are not associated with any Behavior Pattern Mappings and are therefore not dependent on any Properties to be realized. The informal rationale behind the decision to annotate the Pattern Structure with the *Stateless Component* Behavior Pattern is that stateless applications can be scaled in and out more easily [FLR+14]. The informal rationale for the *Processing Component* Behavior Pattern is that the Executable Structure contains the Elastic Beanstalk Worker Environment Component. Finally, the *region* Property of the AWS Component contained in the Executable Structure is connected to the *location* Property of the *Public Cloud* Component Pattern by a Property Mapping.

The PDRMs discussed throughout this thesis and in this section are now used to build a PbDCM for the running example depicted in Figure 1.1. This provides a comprehensive view of the patterns realized by the running example and enables the application of the pattern refinement process. The final PbDCM generated by the pattern detection process is depicted in Figure 5.2. All four *Secure-Queue-Connection* Relations were replaced with *Queue-Connection* Relations annotated with the *Secure Channel* [SFH+06] Behavior Pattern by applying the PDRM depicted at the top of Figure 3.4. The *Secure-SQL-Connection* Relation was replaced with a *SQL-Connection* Relation annotated with the *Secure Channel* Behavior Pattern by applying the PDRM depicted at the bottom

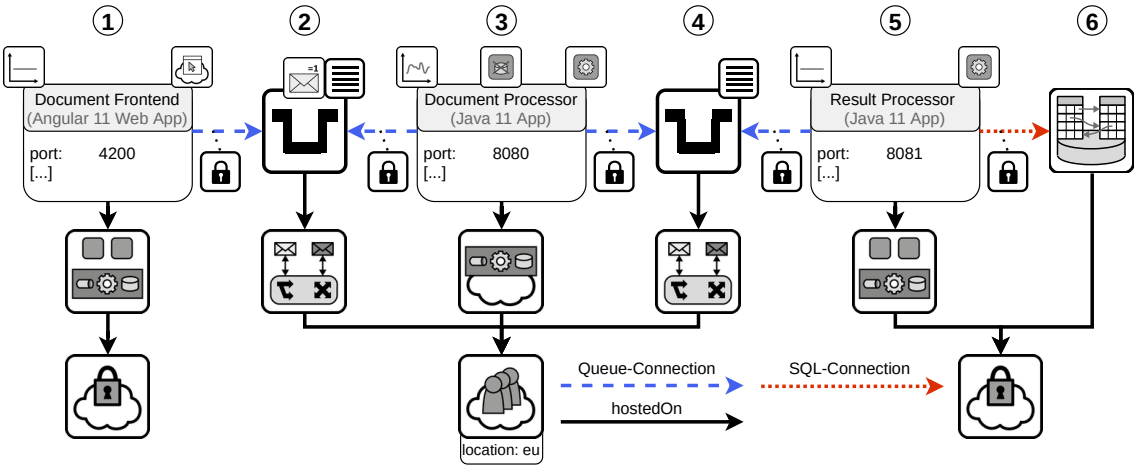

Figure 5.2: PbDCM for the running example depicted in Figure 1.1.

of Figure 3.4. The subgraph labeled with the number one in Figure 5.2 is the result of applying the PDRM depicted on the left side of Figure 5.1 to the running example. The subgraph labeled with the number two is the result of applying the PDRM depicted in Figure 3.2. The subgraph labeled with the number three is the result of applying the PDRM depicted on the right side of Figure 5.1. The subgraph labeled with the number four is the result of applying the PDRM depicted in Figure 3.2. As the respective SQS Queue Component in the running example does not specify a type in its Properties, the *Exactly-once Delivery* [FLR+14] Behavior Pattern was removed. Furthermore, as the PDRM depicted in Figure 3.2 and the PDRM depicted in Figure 5.1 specify Property Mappings, the value of the *region* Property of the AWS Component was transferred to the *location* Property of the *Public Cloud* Component Pattern. The subgraph labeled with the number five is the result of applying a PDRM based on the CBPRM depicted in Figure 2.4. The subgraph labeled with the number six is the result of applying the PDRM depicted on the right side of Figure 3.5 based on the procedure described in Section 3.4.

The document processing application discussed in Section 1.1 stores the digitalized documents in a MySQL database. While a MySQL database can be used to store the digitalized documents, it is not the ideal storage solution for this use case. In this example, the documents are stored in the database as Binary Large Objects (BLOBs) identified by an ID [FLR+14]. There are no dependencies between individual documents and the complex query capabilities offered by the MySQL database are not required by the document processing application [FLR+14]. Therefore, it was decided to replace the MySQL database with a more suitable storage solution. BLOB storage solutions are specifically optimized for storing BLOBs with unique IDs in a hierarchy similar to a file system [FLR+14]. Consequently, it was decided to replace the MySQL database with a technology offering such BLOB storage capabilities. However, instead of manually researching existing BLOB storage technologies, determining and selecting one which is applicable to the described use case, and manually adapting the executable deployment model, the generated PbDCM is adapted to reflect the new design decision instead.

Figure 5.3 shows the adapted PbDCM. The *Relational Database* [FLR+14] Component Pattern was replaced with the *Blob Storage* [FLR+14] Component Pattern to reflect the new design decision. To protect the stored documents, the *Blob Storage* Component Pattern is annotated with the *Information Obscurity* [FLR+14] Behavior Pattern. Additionally, the delivery semantics of the queue which

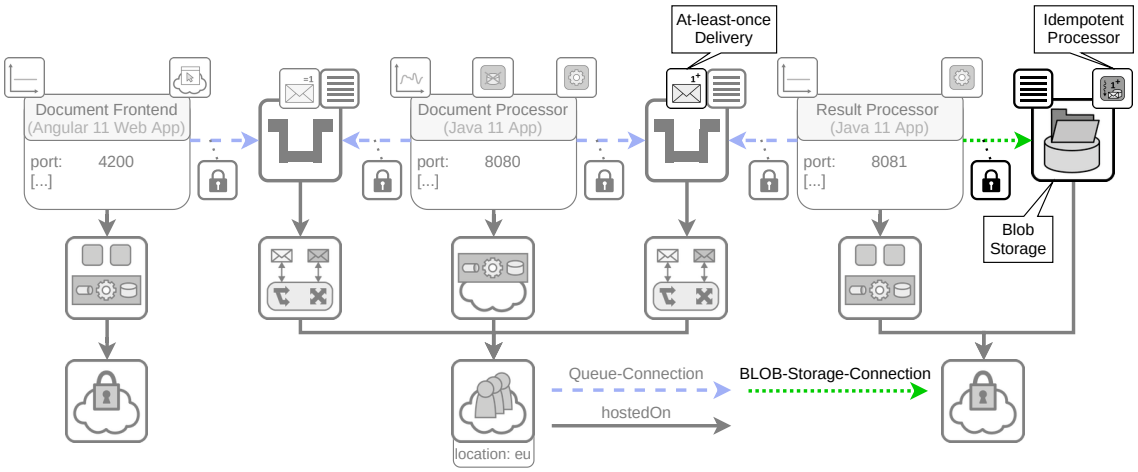

Figure 5.3: Adapted version of the PbDCM depicted in Figure 5.2.

receives the digitalized documents is now specified using the *At-least-once Delivery* [FLR+14] Behavior Pattern. To cope with potential duplicate messages, the *Blob Storage* Component Pattern is annotated with the *Idempotent Processor* [FLR+14] Behavior Pattern. Finally, the *SQL-Connection* Relation was replaced with a *BLOB-Storage-Connection* Relation.

To obtain an executable deployment model from the adapted PbDCM, the pattern refinement process is utilized. Figure 5.4 depicts the refined deployment model which realizes the new design decisions. The *Output Queue* Component now specifies that the used SQS Queue is of type *std*, i.e., an SQS Queue of type standard which guarantees that each message is delivered at least once [Ama21]. Instead of the MySQL DB 8 Component, the deployment model now contains a MinIO[1] Component hosted on a Kubernetes[2] 1.20 Component. The MinIO Component offers the described BLOB storage capabilities and is configured to support Server-Side Encryption (SSE) with a master key provided by a Key Management System (KMS). This configuration realizes the *Information Obscurity* Behavior Pattern. Furthermore, the *Idempotent Processor* Behavior Pattern is realized as the MinIO Component overwrites the respective existing digitalized document if a duplicate message occurs. Finally, the *BLOB-Storage-Connection* Relation annotated with the *Secure Channel* Behavior Pattern is realized by a *Secure-BLOB-Storage-Connection* Relation.

In summary, the pattern detection approach presented in this thesis eased the process of identifying the underlying architectural concepts and semantics in the declarative deployment model of the running example. As the approach utilizes the detected patterns to build PbDCMs, a comprehensive view of the patterns realized by the running example was provided. Finally, by adapting the generated PbDCM and utilizing the pattern refinement process, new design decisions were realized on an abstract level. The combination of the pattern detection process and the pattern refinement process enables more reproducible results, reduces the required technical knowledge, reduces the risk for errors, and reduces the time required to perform the described activities.

[1] https://min.io/
[2] https://kubernetes.io/

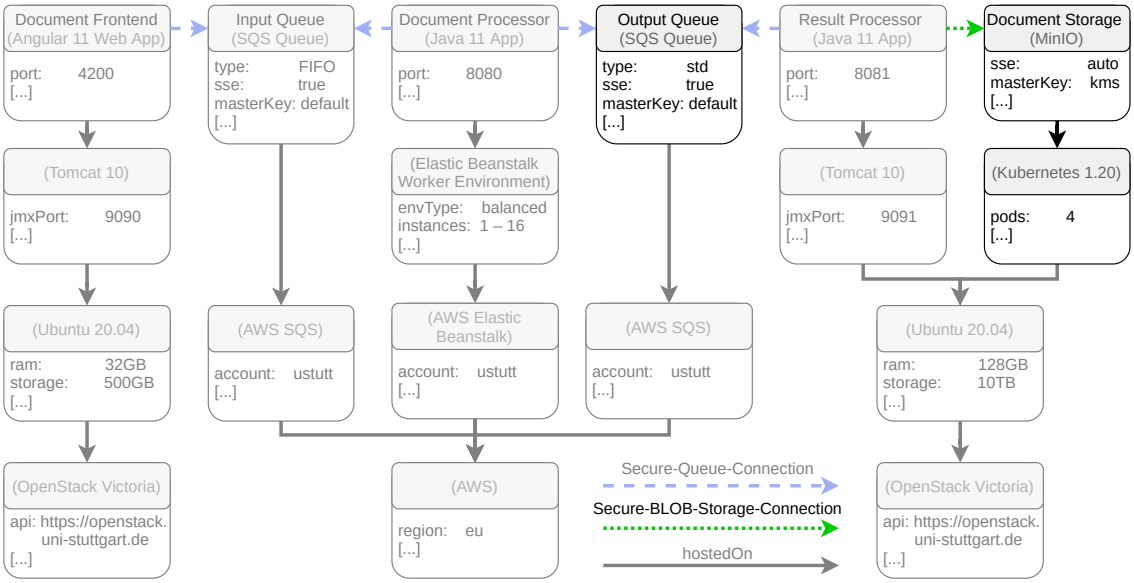

Figure 5.4: Executable deployment model obtained by refining the PbDCM depicted in Figure 5.3.

5.2 Discussion

In this section, the limitations of the presented approach are discussed. (i) First, it should be noted that in order to successfully implement the case study conducted in Section 5.1, multiple versions of the described PDRMs are required due to a limitation that also applies to the pattern refinement process [HBF+20; HBM+18]. For example, the PDRM depicted on the right side of Figure 5.1 replaces the AWS Component contained in the running example with the *Public Cloud* Component Pattern. This also affects the other Components originally hosted on the AWS Component. To be precise, the Relations between both AWS SQS Components and the AWS Component are redirected to the *Public Cloud* Component Pattern during the replacement procedure by means of Relation Mappings. As a result, the PDRM depicted in Figure 3.2 is not applicable anymore as its Executable Structure contains the AWS Component and not the *Public Cloud* Component Pattern. Therefore, another version of the PDRM is required which is applicable to the adapted subgraphs, i.e., contains the *Public Cloud* Component Pattern instead of the AWS Component in the Executable Structure. For brevity, the different versions of the described PDRMs were omitted in the case study.

(ii) Another limitation of the presented approach are the introduced Component Pattern Mappings. For the detection of additional Component Patterns, relaxed compatibility conditions were introduced. The relaxed compatibility conditions consider two elements to be compatible if they realize the same Component Pattern. To guarantee that the realization of a Component Pattern does not depend on other elements, one-to-one Component Pattern Mappings were defined. However, there also exist Component Patterns which require not one Component but a combination of multiple Components to be realized. Such Component Patterns can be detected by the presented approach if the Executable Structure of a PDRM matches a subgraph of a given deployment model exactly but cannot be detected by the relaxed conditions supported by the Component Pattern Mappings.

(iii) Furthermore, there are patterns which cannot be detected based on subgraph isomorphisms and the presented compatibility conditions. For example, the detection of antipatterns and the detection of the problems described by patterns can require the absence of certain elements or semantics [SBKL19]. As the proposed approach is based on the detection of predefined structures in the architecture of an application, the absence of elements or semantics cannot be detected.

(iv) Finally, it should be noted that there are PDRMs which can be used for the refinement of patterns but cannot be used unambiguously for the detection of patterns. For example, both the *Unpredictable Workload* pattern and the *Periodic Workload* [FLR+14] pattern reference elastic scaling as a possible solution to the respective described problems [FLR+14]. Therefore, during pattern refinement, both patterns can be realized by a service which offers elastic scaling capabilities such as the AWS Elastic Beanstalk service configured with a balanced environment. However, while applications using the AWS Elastic Beanstalk service configured with a balanced environment may experience the utilization described by the *Unpredictable Workload* pattern, it cannot be ruled out conclusively that a different kind of utilization occurs. For the sake of simplicity and to avoid the introduction of new patterns, the examples used in this thesis are assumed to be unambiguous with respect to the detected patterns. Furthermore, as the presented pattern detection process is performed in a semi-automatic fashion, a user has the option to discard PDRMs which contain patterns that do not apply unambiguously to his or her particular use case.

6 Related Work

This chapter discusses other approaches which are related to the concept presented in this thesis. Section 6.1 discusses approaches that address the detection of patterns in deployment models. Section 6.2 discusses related work which is concerned with detecting patterns in UML models of applications. Section 6.3 discusses approaches which address the detection of patterns in application code. Section 6.4 discusses related work focused on a specific domain such as the detection of management patterns, antipatterns, problems, and smells. Section 6.5 discusses other approaches relevant in the context of this thesis.

6.1 Pattern Detection in Deployment Models

Wohlfarth [Woh17] describes an approach for the detection of patterns in TOSCA topologies using predefined pattern graphs, pattern taxonomies, and keywords. Pattern graphs are directed graphs consisting of nodes containing the abstract types relevant for a pattern. Additional models defining possible edges between the nodes of a pattern graph are used to impose constraints on the pattern detection procedure. Furthermore, pattern taxonomies are specified which represent directed graphs that define the connections between the patterns themselves. The approach also introduces keywords that are used to determine the abstract type of a Node Template. The proposed algorithm for pattern detection first assigns abstract types to the Node Templates of a given topology based on the keywords found in the names of the respective Node Templates. In the next step, possible patterns are annotated with according probabilities in the pattern taxonomies based on the detected abstract types. The pattern taxonomies can also be used to derive additional patterns from the patterns that have already been detected. In the last step of the algorithm, the given topology is mapped to a graph based on the detected abstract types and its subgraphs are compared to the existing pattern graphs to increase the confidence of the current predictions. The result of the algorithm is a list of patterns and their confidence levels. While the concept of pattern graphs and pattern taxonomies can be compared to the concept of PDRMs, the generation of PbDCMs or similar pattern models and the detection of patterns based on other characteristics such as the properties of the given elements is not considered.

Röhl [Röh17] proposes a framework to detect patterns in TOSCA topologies, analyze the impact of each detected pattern on the non-functional properties of the application, and apply additional patterns to the architecture of the application to adapt its non-functional properties. The approach requires Softgoal Interdependency Graphs (SIGs) for each pattern. An automated approach to generate the SIG for a pattern is not described. The work itself does not include a concept for pattern detection as the author argues that any algorithm capable of pattern detection can be used by the proposed framework. The implementation and validation of the proposed framework uses an extended version of the approach introduced by Wohlfarth [Woh17] for pattern detection. Following the argumentation of the author, the approach described in this thesis is also compatible with the

proposed framework. After the impact of the patterns present in a given topology has been analyzed, additional patterns can be applied. The work does not specify a general-purpose algorithm to apply patterns to a given topology but argues that each pattern has unique characteristics. Therefore, individual algorithms adjusting a given topology need to be developed for each pattern. The generation of PbDCMs or similar pattern models is not considered.

Kumara et al. [KVM+20] propose an approach to detect smells in deployment models such as TOSCA Topology Templates by utilizing a deployment model ontology and SPARQL Protocol and RDF Query Language (SPARQL) rules. A deployment model smell is defined as a violation against best practices or the implementation of bad practices. Smells are detected by converting a given deployment model to the deployment model ontology and executing predefined SPARQL rules on the result. The detection of more general patterns or the generation of PbDCMs or similar pattern models is not considered.

6.2 Pattern Detection in UML Models

Multiple approaches address the detection of patterns in UML models of applications by utilizing Prolog rules and statements. Bergenti and Poggi [BP00; BP02] evaluate the class diagrams and collaboration diagrams of applications using Prolog rules and suggest possible improvements based on the result. Di Martino and Esposito [DE16] and Kim and Lu [KL06] introduce procedures to transform UML models to Prolog statements which are evaluated by Prolog rules representing patterns. The generation of PbDCMs or similar pattern models is not considered.

Zdun and Avgeriou [ZA05] propose a set of architectural primitives described as UML models which are common to the implementations of multiple design patterns. Haitzer and Zdun [HZ15] propose a semi-automatic approach to detect patterns in applications based on architectural primitives. The approach requires the architecture of an application to be modeled in a Domain-specific Language (DSL) based on UML which describes the contained architectural primitives. The detection of patterns is performed by comparing the architectural primitives of predefined patterns to the architectural primitives present in the created model of the application. Pattern detection in deployment models and the generation of models that contain complete patterns instead of architectural primitives is not considered.

Kim and El Khawand [KE07] propose an UML-based metamodel to specify the problem domain of design patterns. A pattern described using the proposed metamodel can be detected in the architectural description of an application using the same metamodel based on a divide-and-conquer process comparing the individual elements and the complete models. The detection of patterns in deployment models and the generation of PbDCMs or similar pattern models is not considered.

6.3 Pattern Detection in Application Code

A number of approaches have been proposed with the detection of the patterns introduced by Gamma et al. [GHJV94] in mind. Blewitt et al. [BBS05] enable the detection of patterns described as declarative statements by evaluating the statements on Java source code.

Kramer and Prechelt [KP96] enable the detection of patterns described as Object-Modeling Technique (OMT) diagrams in the source code of an application. The source code is transformed to OMT diagrams which are then transformed to Prolog facts that can be evaluated by Prolog rules generated from the pattern OMT diagrams. Albin-Amiot and Guéhéneuc [AG01] enable the detection of patterns described as UML models by comparing them to UML models generated from the source code of an application. Philippow et al. [PSRN05] introduce positive and negative search criteria to reduce the number of false positive results during the detection of patterns in application code. Heuzeroth et al. [HHHL03] describe a two-phased pattern detection process consisting of a static analysis and a dynamic analysis of the source code of an application. Both of the analysis phases rely on the definition of respective algorithms for each pattern to perform the actual detection. Shi and Olsson [SO06] reclassify the patterns introduced by Gamma et al. [GHJV94] and enable their detection in Java source code by utilizing static analyses. Kampffmeyer and Zschaler [KZ07] propose an ontology language to formalize the problem space of patterns. The ontology language can be used to create a knowledge base which enables the identification of patterns by formulating the design of an application as a query to the knowledge base. None of the approaches considers the detection of patterns on a more abstract architectural level or the detection of patterns in deployment models.

Guéhéneuc and Antoniol [GA08] describe an approach to detect design patterns defined using an UML-like metamodel in object-oriented application code. The detection of patterns is enabled by generating an UML-like model from the code of an application and enriching it with additional characteristics and relationships. Patterns can be detected by transforming the model of a pattern to constraints and solving the created constraint satisfaction problem. Tsantalis et al. [TCSH06] propose an approach to detect patterns and their variations in the source code of an application. Patterns are represented by directed graphs describing characteristics such as inheritance hierarchies. The described characteristics are extracted from the source code of an application and used to build the required graph structures which are then compared to the pattern graphs by means of a similarity algorithm instead of a graph matching algorithm. Moreno and Marcus [MM12] propose a tool to detect method and class stereotypes in Java source code. The detected stereotypes are used to enrich Javadoc comments and generate reports. Arcelli Fontana and Zanoni [AZ11] propose methods for design pattern detection and software architecture reconstruction based on the Abstract Syntax Tree (AST) of the source code of an application. For design pattern detection, fundamental pattern elements are extracted from the AST and used to create a graph representing the application. Predefined subgraphs are then used to detect pattern instances in the graph which are finally analyzed by a classification process to reduce the number of false positives. None of the approaches considers the detection of patterns on a more abstract architectural level, the detection of patterns in deployment models, or the generation of PbDCMs or similar pattern models.

6.4 Detection of Patterns in a Specific Domain

Across several works Breitenbücher [Bre16] and Breitenbücher et al. [BBK+14b; BBKL13; BBKL14] describe pattern-based processes for application management including the automated realization of management patterns, the automated refinement of management patterns, and context-aware application management. Similarly to the approach described in this thesis, the processes are based on the detection of subgraphs in models of an application. The subgraphs used for comparison are linked to management operations, transformations, workflows, and other elements supporting

the described concepts. The approach presented in this thesis is closely related to and in some parts inspired by the described works. However, the described works are concerned with management patterns and not with architectural patterns to describe the semantics of an application.

Multiple approaches address the detection of antipatterns. Rekik et al. [RBB15] describe an ontology to describe cloud services. In [RBGB17], an approach is introduced to detect antipatterns in cloud service models described using the aforementioned ontology. Based on the detected antipatterns, possible corrections are suggested. Cortellessa et al. [CDT14] introduce an approach to detect performance antipatterns in applications. The detection procedure is based on a pattern catalogue containing patterns described as logical predicates and operates on an Extensible Markup Language (XML) representation of the architecture of an application. Nahar and Sakib [NS15] propose an approach to detect antipatterns in UML class diagrams and UML sequence diagrams. Patterns are detected by converting the respective diagrams to matrices and comparing them to matrices representing the antipatterns. Based on the detected antipatterns, corresponding design patterns are suggested. Brabra et al. [BMS+16] describe several patterns and antipatterns to reflect the guidelines of Open Cloud Computing Interface (OCCI) based on several ontologies. The described patterns and antipatterns are used to create a knowledge base of Semantic Web Rule Language (SWRL) rules which can be queried to verify the presence of patterns or antipatterns in applications modeled using the introduced ontologies. In [BMP+19], this approach is extended to support REST patterns and antipatterns. Settas et al. [SMSB11] propose an ontology to describe antipatterns and their relationships to each other. The described antipatterns can be identified based on a set of symptoms present in an application and selected by a user. Palma et al. [PMG18] introduce a unified metamodel based on REST, Service Component Architecture (SCA), and SOAP. Antipatterns described using this unified metamodel can be detected by specifying a set of rules for each antipattern which is used for the automatic generation of respective detection algorithms. While antipatterns can be detected in a similar fashion as patterns, they are by definition not suitable for the generation of abstract solution models such as the PbDCMs used in this thesis. The approaches which recommend patterns based on the detected antipatterns do not use them for the generation of PbDCMs or similar pattern models.

Different works address the detection of problems. Saatkamp et al. [SBF+19; SBKL18; SBKL19] introduce an approach to automatically detect and solve problems in restructured deployment models. Problems are defined as logical rules and linked to solution strategies. The detection of problems is enabled by transforming deployment models to logical facts. Uchiumi et al. [UKM12] propose an approach to detect faulty configuration values in cloud datacenters based on decision tree analysis. Mekruksavanich [Mek17] describes an approach to detect design defects in object-oriented source code using artificial neural networks. None of the approaches considers the generation of PbDCMs or similar pattern models.

Multiple approaches address the detection of smells in different artifacts. Moha et al. [MGDL10] propose a method to specify smells and automatically detect them in a given application. Smells are specified by a set of rules which can be automatically transformed to domain-specific detection algorithms. The detection algorithms are executed on a model of the application which is suggested to be obtained through reverse-engineering techniques. Rahman et al. [RPW19] identify seven common security smells in Infrastructure as Code (IaC) scripts and introduce a static analysis tool for detecting said smells. The static analysis tool employs a parser which tokenizes IaC scripts and a rules engine which evaluates predefined rules on the tokenized representation returned by the parser. Polášek et al. [PSK12] propose a rule-based approach to detect smells in UML models.

Smells are defined via the Jess rules engine and evaluated based on automatically generated Jess facts corresponding to the UML models. None of the approaches considers the detection of more general patterns or the generation of PbDCMs or similar pattern models.

6.5 Other Related Work

Hasheminejad and Jalili [HJ12] propose an approach to suggest applicable design patterns during the design phase of an application. Design patterns are suggested based on a text classification procedure using the problem descriptions of the design patterns as training data and analyzing a textual description of the design problem to be solved. The generation of PbDCMs or similar pattern models is not considered.

Krieger et al. [KBKL18] describe an approach to automatically check the compliance of topologies by utilizing predefined compliance rules. Similarly to the approach described in this thesis, a compliance rule consists of two topologies and verifying a rule for a given topology is based on subgraph isomorphism. However, both topologies of a compliance rule are used for verification in a given topology and the detected subgraph is not replaced. Furthermore, the detection of architectural patterns is not considered.

Wild et al. [WBH+20] introduce two modeling styles based on TOSCA to support the deployment and orchestration of quantum applications. The first modeling style is agnostic to any quantum computing Software Development Kit (SDK). The second modeling style is specific to quantum computing SDKs. Models created using the first modeling style can be refined to models of the second modeling style by using Topology Fragment Refinement Models (TFRMs) which are based on CBPRMs. The approach introduced the Property Mappings which were adopted in this thesis. The detection of architectural patterns is not considered.

7 Conclusion and Future Work

Deployment models of applications increase the reusability of deployments and ease the process of automation but vendor and product-specific details obfuscate their underlying architectural concepts and semantics. The essential architectural decisions realized in a deployment model can be stated more clearly by abstract design patterns, however, manually extracting such patterns from deployment models is a non-trivial task as it requires technical knowledge about the patterns a certain component or parts of the application realize. In this thesis, a semi-automatic and iterative process for the detection of patterns in declarative deployment models was presented. In contrast to the manual detection of patterns, the presented approach enables more reproducible results, reduces errors, reduces the needed effort to be made, and reduces the required technical knowledge. The pattern detection process was facilitated by the introduction of Pattern Detection and Refinement Models (PDRMs) which are suitable to both the detection of patterns and the refinement of patterns. The detected patterns were used to generate Pattern-based Deployment and Configuration Models (PbDCMs) which provide a comprehensive view of the architectural concepts and semantics of a deployment model. The generated PbDCMs can be further adapted to model new design decisions on an abstract level which can then be realized in an executable deployment model by using the pattern refinement process.

As described in Section 5.2, the presented approach is limited by the need for multiple versions of the same PDRMs, the one-to-one Component Pattern Mappings, and the potential ambiguity of PDRMs. To avoid the manual creation of multiple versions of the same PDRM which may be required to create a PbDCM for an executable deployment model (cf. limitation (i) in Section 5.2), permutations of PDRMs could be generated in future work, e.g., based on the defined mappings. Additionally, to utilize the Component Pattern Mappings introduced in this thesis for the detection of Component Patterns which require a combination of multiple Components to be realized (cf. limitation (ii) in Section 5.2), the relaxed conditions described in Section 3.4 could be generalized to allow many-to-many correspondences between Model Nodes instead of only one-to-one correspondences. This generalization would entail more complex compatibility conditions and additional subgraph comparisons which were not considered in this thesis due to time constraints. Furthermore, to avoid potential ambiguity during the pattern detection process (cf. limitation (iv) in Section 5.2), new unambiguous patterns could be defined and unambiguous PDRMs could be differentiated from ambiguous PDRMs. A new unambiguous pattern could, for example, describe that a component must be scaled in or out elastically depending on its utilization instead of specifying a specific kind of workload. PDRMs could be used in combination with CBPRMs to differentiate models which can be used for the detection of patterns as well as the refinement of patterns and models which can only be used for the refinement of patterns. Finally, the presented approach could be adapted to be fully automated. Like the pattern refinement process, the pattern detection process determines matching PDRMs for a model but still requires a user to select the PDRM to apply during each iteration and decide on the overall order of the process. Therefore, criteria could be defined to automatically select the most suitable PDRM if multiple options are available.

Bibliography

[AG01] H. Albin-Amiot, Y.-G. Guéhéneuc. "Meta-modeling design patterns: Application to pattern detection and code synthesis". In: *Proceedings of ECOOP Workshop on Automating Object-Oriented Software Development Methods*. 2001 (cit. on p. 71).

[AIS77] C. Alexander, S. Ishikawa, M. Silverstein. *A pattern language: towns, buildings, construction*. Oxford University Press, 1977 (cit. on pp. 22, 23).

[Ale79] C. Alexander. *The timeless way of building*. Oxford University Press, 1979 (cit. on p. 22).

[Ama21] Amazon Web Services. *Amazon SQS features*. 2021. URL: https://aws.amazon.com/sqs/features/ (cit. on pp. 16, 39, 66).

[AZ11] F. Arcelli Fontana, M. Zanoni. "A tool for design pattern detection and software architecture reconstruction". In: *Information Sciences*. Vol. 181. 7. Elsevier, 2011, pp. 1306–1324 (cit. on p. 71).

[BBK+12] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, D. Schumm. "Vino4TOSCA: A Visual Notation for Application Topologies based on TOSCA". In: *On the Move to Meaningful Internet Systems: OTM 2012 (CoopIS 2012)*. Springer, Sept. 2012, pp. 416–424 (cit. on p. 17).

[BBK+14a] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, J. Wettinger. "Combining Declarative and Imperative Cloud Application Provisioning Based on TOSCA". In: *2014 IEEE International Conference on Cloud Engineering*. IEEE. 2014, pp. 87–96 (cit. on pp. 15, 21).

[BBK+14b] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, M. Wieland. "Context-Aware Cloud Application Management". In: *Proceedings of the 4th International Conference on Cloud Computing and Services Science (CLOSER 2014)*. SciTePress, Apr. 2014, pp. 499–509 (cit. on p. 71).

[BBKL13] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. "Pattern-based Runtime Management of Composite Cloud Applications". In: *Proceedings of the 3rd International Conference on Cloud Computing and Services Science (CLOSER 2013)*. SciTePress, May 2013, pp. 475–482 (cit. on p. 71).

[BBKL14] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. "Automating Cloud Application Management Using Management Idioms". In: *Proceedings of the Sixth International Conferences on Pervasive Patterns and Applications (PATTERNS 2014)*. Xpert Publishing Services, May 2014, pp. 60–69 (cit. on p. 71).

[BBS05] A. Blewitt, A. Bundy, I. Stark. "Automatic Verification of Design Patterns in Java". In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ASE '05. Long Beach, CA, USA: Association for Computing Machinery, 2005, pp. 224–232 (cit. on p. 70).

[BMP+19] H. Brabra, A. Mtibaa, F. Petrillo, P. Merle, L. Sliman, N. Moha, W. Gaaloul, Y.-G. Guéhéneuc, B. Benatallah, F. Gargouri. “On semantic detection of cloud API (anti)patterns”. In: *Information and Software Technology*. Vol. 107. Elsevier, 2019, pp. 65–82 (cit. on p. 72).

[BMS+16] H. Brabra, A. Mtibaa, L. Sliman, W. Gaaloul, B. Benatallah, F. Gargouri. “Detecting Cloud (Anti)Patterns: OCCI Perspective”. In: *Service-Oriented Computing*. Cham: Springer International Publishing, 2016, pp. 202–218 (cit. on p. 72).

[BP00] F. Bergenti, A. Poggi. “IDEA: A design assistant based on automatic design pattern detection”. In: *Proceedings of the 12th international conference on Software Engineering and Knowledge Engineering*. Springer-Verlag. 2000, pp. 336–343 (cit. on p. 70).

[BP02] F. Bergenti, A. Poggi. “Improving UML designs using automatic design pattern detection”. In: *Handbook of Software Engineering and Knowledge Engineering: Volume II: Emerging Technologies*. World Scientific, 2002, pp. 771–784 (cit. on p. 70).

[Bre16] U. Breitenbücher. “Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements”. Dissertation. University of Stuttgart, 2016 (cit. on p. 71).

[CDT14] V. Cortellessa, A. Di Marco, C. Trubiani. “An Approach for Modeling and Detecting Software Performance Antipatterns Based on First-Order Logics”. In: *Software and Systems Modeling (SoSyM)*. Vol. 13. 1. Berlin, Heidelberg: Springer, Feb. 2014, pp. 391–432 (cit. on p. 72).

[CFSV04] L. P. Cordella, P. Foggia, C. Sansone, M. Vento. “A (sub)graph isomorphism algorithm for matching large graphs”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 26. 10. IEEE, 2004, pp. 1367–1372 (cit. on p. 41).

[CFSV17] V. Carletti, P. Foggia, A. Saggese, M. Vento. “Introducing VF3: A New Algorithm for Subgraph Isomorphism”. In: *Graph-Based Representations in Pattern Recognition*. Cham: Springer International Publishing, 2017, pp. 128–139 (cit. on p. 41).

[CFV15] V. Carletti, P. Foggia, M. Vento. “VF2 Plus: An Improved version of VF2 for Biological Graphs”. In: *Graph-Based Representations in Pattern Recognition*. Cham: Springer International Publishing, 2015, pp. 168–177 (cit. on p. 41).

[DE16] B. Di Martino, A. Esposito. “A Rule-Based Procedure for Automatic Recognition of Design Patterns in UML Diagrams”. In: *Software Practice and Experience*. Vol. 46. 7. USA: John Wiley & Sons, Inc., July 2016, pp. 983–1007 (cit. on p. 70).

[EBF+17] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, J. Wettinger. “Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications”. In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services (XPS), 2017, pp. 22–27 (cit. on pp. 15, 21).

[Feh15] C. Fehling. “Cloud computing patterns: identification, design, and application”. Dissertation. University of Stuttgart, 2015 (cit. on p. 15).

[FL17] M. Falkenthal, F. Leymann. “Easing Pattern Application by Means of Solution Languages”. In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services (XPS), 2017, pp. 58–64 (cit. on p. 23).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Springer, 2014 (cit. on pp. 15–18, 22–25, 36, 39, 44, 47, 48, 63–66, 68).

[GA08] Y. Guéhéneuc, G. Antoniol. “DeMIMA: A Multilayered Approach for Design Pattern Identification”. In: *IEEE Transactions on Software Engineering*. Vol. 34. 5. IEEE, 2008, pp. 667–684 (cit. on p. 71).

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Oct. 1994 (cit. on pp. 70, 71).

[Har18] L. Harzenetter. “Versioning of applications modeled in TOSCA”. Master’s Thesis. University of Stuttgart, 2018 (cit. on p. 60).

[HAW11] H. Herry, P. Anderson, G. Wickler. “Automated Planning for Configuration Changes”. In: *Proceedings of the 25th International Conference on Large Installation System Administration*. LISA’11. Boston, MA: USENIX Association, 2011, p. 5 (cit. on pp. 15, 21).

[HBF+20] L. Harzenetter, U. Breitenbücher, M. Falkenthal, J. Guth, F. Leymann. “Pattern-based Deployment Models Revisited: Automated Pattern-driven Deployment Configuration”. In: *Proceedings of the Twelfth International Conference on Pervasive Patterns and Applications (PATTERNS 2020)*. Xpert Publishing Services, Oct. 2020, pp. 40–49 (cit. on pp. 15, 16, 21, 24, 26–31, 33, 34, 37, 41, 42, 45, 46, 52, 54–56, 67).

[HBM+18] L. Harzenetter, U. Breitenbücher, F. Michael, J. Guth, C. Krieger, F. Leymann. “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, pp. 41–52 (cit. on pp. 15, 16, 21, 24, 26–31, 33, 34, 37, 41, 42, 52, 54–56, 67).

[HHHL03] D. Heuzeroth, T. Holl, G. Hogstrom, W. Lowe. “Automatic design pattern detection”. In: *11th IEEE International Workshop on Program Comprehension, 2003*. IEEE. 2003, pp. 94–103 (cit. on p. 71).

[HJ12] S. M. H. Hasheminejad, S. Jalili. “Design patterns selection: An automatic two-phase method”. In: *Journal of Systems and Software*. Vol. 85. 2. Special issue with selected papers from the 23rd Brazilian Symposium on Software Engineering. Elsevier, 2012, pp. 408–424 (cit. on p. 73).

[HW04] G. Hohpe, B. Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004 (cit. on pp. 15, 22, 24, 39, 63).

[HZ15] T. Haitzer, U. Zdun. “Semi-Automatic Architectural Pattern Identification and Documentation Using Architectural Primitives”. In: *Journal of Systems and Software*. Vol. 102. C. USA: Elsevier, Apr. 2015, pp. 35–57 (cit. on p. 70).

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. “Winery - A Modeling Tool for TOSCA-based Cloud Applications”. In: *Proceedings of 11th International Conference on Service-Oriented Computing (ICSOC’13)*. Vol. 8274. LNCS. Springer Berlin Heidelberg, Dec. 2013, pp. 700–704 (cit. on pp. 17, 55–57).

[KBKL18] C. Krieger, U. Breitenbücher, K. Képes, F. Leymann. “An Approach to Automatically Check the Compliance of Declarative Deployment Models”. In: *Papers from the 12th Advanced Summer School on Service-Oriented Computing (SummerSoC 2018)*. IBM Research Division, Oct. 2018, pp. 76–89 (cit. on p. 73).

[KE07] D.-K. Kim, C. El Khawand. “An Approach to Precisely Specifying the Problem Domain of Design Patterns”. In: *Journal of Visual Languages & Computing*. Vol. 18. 6. USA: Academic Press, Inc., Dec. 2007, pp. 560–591 (cit. on p. 70).

[KL06] D.-K. Kim, L. Lu. “Inference of design pattern instances in UML models via logic programming”. In: *11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS’06)*. IEEE. 2006 (cit. on p. 70).

[KP96] C. Kramer, L. Prechelt. “Design recovery by automated search for structural design patterns in object-oriented software”. In: *Proceedings of WCRE ’96: 3rd Working Conference on Reverse Engineering*. IEEE. 1996, pp. 208–215 (cit. on p. 71).

[KVM+20] I. Kumara, Z. Vasileiou, G. Meditskos, D. A. Tamburri, W.-J. Van Den Heuvel, A. Karakostas, S. Vrochidis, I. Kompatsiaris. “Towards Semantic Detection of Smells in Cloud Infrastructure Code”. In: *Proceedings of the 10th International Conference on Web Intelligence, Mining and Semantics*. ACM, 2020, pp. 63–67 (cit. on p. 70).

[KZ07] H. Kampffmeyer, S. Zschaler. “Finding the Pattern You Need: The Design Pattern Intent Ontology”. In: *Model Driven Engineering Languages and Systems*. Springer Berlin Heidelberg, 2007, pp. 211–225 (cit. on p. 71).

[Mek17] S. Mekruksavanich. “An adaptive approach for automatic design defects detection in object-oriented systems”. In: *2017 International Conference on Digital Arts, Media and Technology (ICDAMT)*. IEEE. 2017, pp. 342–346 (cit. on p. 72).

[MGDL10] N. Moha, Y.-G. Guéhéneuc, L. Duchien, A.-F. Le Meur. “DECOR: A Method for the Specification and Detection of Code and Design Smells”. In: *IEEE Transactions on Software Engineering*. Vol. 36. 1. IEEE, 2010, pp. 20–36 (cit. on p. 72).

[MM12] L. Moreno, A. Marcus. “JStereoCode: automatically identifying method and class stereotypes in Java code”. In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2012, pp. 358–361 (cit. on p. 71).

[NS15] N. Nahar, K. Sakib. “Automatic Recommendation of Software Design Patterns Using Anti-patterns in the Design Phase: A Case Study on Abstract Factory”. In: *3rd International Workshop on Quantitative Approaches to Software Quality (QuASoQ)*. Dec. 2015 (cit. on p. 72).

[OAS13] OASIS. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. 2013 (cit. on pp. 15, 21, 55, 57).

[OAS20] OASIS. *TOSCA Simple Profile in YAML Version 1.3*. 2020 (cit. on pp. 15, 21, 55, 57).

[OGP03] D. Oppenheimer, A. Ganapathi, D. A. Patterson. “Why Do Internet Services Fail, and What Can Be Done about It?” In: *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*. USITS’03. Seattle, WA: USENIX Association, 2003, p. 1 (cit. on p. 15).

[PMG18] F. Palma, N. Moha, Y.-G. Guéhéneuc. “UniDoSA: The Unified Specification and Detection of Service Antipatterns”. In: *IEEE Transactions on Software Engineering*. Vol. PP. IEEE, Mar. 2018, pp. 1–1 (cit. on p. 72).

[PSK12] I. Polášek, S. Snopko, I. Kapustík. “Automatic identification of the anti-patterns using the rule-based approach”. In: *2012 IEEE 10th Jubilee International Symposium on Intelligent Systems and Informatics*. IEEE. 2012, pp. 283–286 (cit. on p. 72).

[PSRN05] I. Philippow, D. Streitferdt, M. Riebisch, S. Naumann. “An approach for reverse engineering of design patterns”. In: *Software and System Modeling*. Vol. 4. Springer, Feb. 2005, pp. 55–70 (cit. on p. 71).

[RBB15] M. Rekik, K. Boukadi, H. Ben-Abdallah. “Cloud description ontology for service discovery and selection”. In: *2015 10th International Joint Conference on Software Technologies (ICSOFT)*. Vol. 1. IEEE. 2015, pp. 1–11 (cit. on p. 72).

[RBGB17] M. Rekik, K. Boukadi, W. Gaaloul, H. Ben-Abdallah. “Anti-Pattern Specification and Correction Recommendations for Semantic Cloud Services”. In: *50th Hawaii International Conference on System Sciences*. Jan. 2017 (cit. on p. 72).

[Rei14] R. Reiners. “An evolving pattern library for collaborative project documentation”. Dissertation. RWTH Aachen University, 2014 (cit. on p. 23).

[Röh17] A. Röhl. “Design and implementation of a framework to automate the inclusion of patterns in existing architectures”. Bachelor’s Thesis. University of Stuttgart, 2017 (cit. on p. 69).

[RPW19] A. Rahman, C. Parnin, L. Williams. “The Seven Sins: Security Smells in Infrastructure as Code Scripts”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 164–175 (cit. on p. 72).

[SBF+19] K. Saatkamp, U. Breitenbücher, M. Falkenthal, L. Harzenetter, F. Leymann. “An Approach to Determine & Apply Solutions to Solve Detected Problems in Restructured Deployment Models using First-order Logic”. In: *Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER 2019)*. SciTePress, May 2019, pp. 495–506 (cit. on p. 72).

[SBKL18] K. Saatkamp, U. Breitenbücher, O. Kopp, 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 (SummerSOC’18)*. IBM Research Division, 2018, pp. 43–53 (cit. on p. 72).

[SBKL19] K. Saatkamp, U. Breitenbücher, O. Kopp, F. Leymann. “An approach to automatically detect problems in restructured deployment models based on formalizing architecture and design patterns”. In: *SICS Software-Intensive Cyber-Physical Systems*. Springer Berlin Heidelberg, Feb. 2019, pp. 1–13 (cit. on pp. 68, 72).

[SBLE12] D. Schumm, J. Barzen, F. Leymann, L. Ellrich. “A Pattern Language for Costumes in Films”. In: *Proceedings of the 17th European Conference on Pattern Languages of Program (EuroPLoP 2012)*. ACM, 2012 (cit. on p. 22).

[SFH+06] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, P. Sommerlad. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons, Inc., Jan. 2006 (cit. on pp. 15, 16, 22, 24, 25, 39, 46, 63, 64).

[SMSB11] D. L. Settas, G. Meditskos, I. G. Stamelos, N. Bassiliades. "SPARSE: A symptom-based antipattern retrieval knowledge-based system using Semantic Web technologies". In: *Expert Systems with Applications*. Vol. 38. 6. Elsevier, 2011, pp. 7633–7646 (cit. on p. 72).

[SO06] N. Shi, R. A. Olsson. "Reverse Engineering of Design Patterns from Java Source Code". In: *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE. 2006, pp. 123–134 (cit. on p. 71).

[TCSH06] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S. T. Halkidis. "Design Pattern Detection Using Similarity Scoring". In: *IEEE Transactions on Software Engineering*. Vol. 32. 11. IEEE, 2006, pp. 896–909 (cit. on p. 71).

[UKM12] T. Uchiumi, S. Kikuchi, Y. Matsumoto. "Misconfiguration detection for cloud datacenters using decision tree analysis". In: *2012 14th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE. 2012, pp. 1–4 (cit. on p. 72).

[WBF+19] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp, J. Soldani. "The Essential Deployment Metamodel: A Systematic Review of Deployment Automation Technologies". In: *SICS Software-Intensive Cyber-Physical Systems*. Vol. 35. Springer, Aug. 2019, pp. 63–75 (cit. on pp. 15, 21, 22, 24, 25).

[WBH+20] K. Wild, U. Breitenbücher, L. Harzenetter, F. Leymann, D. Vietz, M. Zimmermann. "TOSCA4QC: Two Modeling Styles for TOSCA to Automate the Deployment and Orchestration of Quantum Applications". In: *Proceedings of the 24th International Enterprise Distributed Object Computing Conference (EDOC 2020)*. IEEE, Oct. 2020, pp. 125–134 (cit. on pp. 34, 36, 37, 40, 57, 73).

[Woh17] M. Wohlfarth. "Design pattern detection framework for TOSCA-topologies". Bachelor's Thesis. University of Stuttgart, 2017 (cit. on p. 69).

[ZA05] U. Zdun, P. Avgeriou. "Modeling Architectural Patterns Using Architectural Primitives". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. San Diego, CA, USA: Association for Computing Machinery, 2005, pp. 133–146 (cit. on p. 70).

[Zdu07] U. Zdun. "Systematic Pattern Selection Using Pattern Language Grammars and Design Space Analysis". In: *Software: Practice and Experience*. Vol. 37. 9. USA: John Wiley & Sons, Inc., July 2007, pp. 983–1016 (cit. on p. 23).

All links were last followed on April 27, 2021.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature