Institute of Architecture of Application Systems

An approach to automatically detect problems in restructured deployment models based on formalizing architecture and design patterns

Karoline Saatkamp, Uwe Breitenbücher,
Oliver Kopp, Frank Leymann

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

BibTeX:

```
@article {Saatkamp2019_ProblemDetection,
    Author    = {Karoline Wild and Uwe Breitenb{\"u}cher and Oliver Kopp
                 and Frank Leymann},
    Title     = {{An approach to automatically detect problems in
                 restructured deployment models based on formalizing
                 architecture and design patterns}},
    Journal   = {SICS Software-Intensive Cyber-Physical Systems},
    Publisher = {Springer Berlin Heidelberg},
    Pages     = {1--13},
    Month     = feb,
    Year      = 2019,
    doi       = {10.1007/s00450-019-00397-7}
}
```


Universität Stuttgart
Germany

Noname manuscript No.
(will be inserted by the editor)

An Approach to Automatically Detect Problems in Restructured Deployment Models based on Formalizing Architecture and Design Patterns

Karoline Saatkamp · Uwe Breitenbücher · Oliver Kopp · Frank Leymann

Received: date / Accepted: date

Abstract For the automated deployment of applications, technologies exist which can process topology-based deployment models that describes the application's structure with its components and their relations. The topology-based deployment model of an application can be adapted for the deployment in different environments. However, the structural changes can lead to problems, which had not existed before and prevent a functional deployment. This includes security issues, communication restrictions, or incompatibilities. For example, a formerly over the internal network established insecure connection leads to security problems when using the public network after the adaptation. In order to solve problems in adapted deployment models, first the problems have to be detected. Unfortunately, detecting such problems is a highly non-trivial challenge that requires deep expertise about the involved technologies and the environment. In this paper, we present (i) an approach for detecting problems in deployment models using architecture and design patterns and (ii) the automation of the detection process by formalizing the problem a pattern solves in a certain context. We validate the practical feasibility of our approach by a prototypical implementation for the automated problem detection in TOSCA topologies.

Keywords Topology-based Deployment Model · Patterns · Problem Detection · TOSCA · Logic Programming · Prolog

K. Saatkamp / U. Breitebücher / O. Kopp / F. Leymann
Universitätsstraße 38 70569 Stuttgart Germany
Tel.: +49-711-68588 – 470
Fax: +49-711-68588 – 472
E-mail: [lastname]@informatik.uni-stuttgart.de

1 Introduction

The increasing number of applications in today's enterprises leads to a great management effort, since different technologies often have to be combined for their deployment (Breitenbücher et al., 2013b). Therefore, several systems for automating the deployment have been developed. Beside deployment systems such as Ansible[1] or Cloud Foundry[2], standards such as TOSCA (OASIS, 2013) are published to describe the deployment and management of applications in a portable manner.

Diverse deployment technologies are based on processing topology-based deployment models that describe the application's structure with its components and their relations. These deployment models often have to be adapted for several reasons: For example, an application is deployed several times in different environments or parts of the IT are outsourced. Such a *restructuring* can result in complex problems that have not existed before. This includes security issues, communication restrictions, or incompatibilities. Security issues can occur, for example, when components intended to communicate over the internal network are now distributed and have to communicate over the public network without using security mechanisms such as encryption. Thus, the exchanged data are potentially available for eavesdroppers. Another problem can occur when one component of two communicating components that were previously hosted on the same virtual machine is moved to an environment protected by a firewall and this component is no longer accessible from outside. Such problems prevent a functional deployment after the adaptation of the deployment model.

[1] `https://www.ansible.com/`
[2] `https://www.cloudfoundry.org/`

However, detecting such problems is a highly non-trivial knowledge-intensive challenge that typically also requires knowledge about the involved environments. An enormous number of components, their capabilities, and their interconnections have to be known as many problemsarise from communication dependencies that are violated when, for example, an application previously hosted on one virtual machine gets distributed into different environments. Furthermore, the available services in different cloud environments must be known and how their have to be configured, e.g. firewalls in order to allow incoming connections. A human operator would require expertise in all these areas (i) to detect that a problem arose and moreover (ii) to solve it. The variety and complexity of possible problems makes an manual approach unpractical: Existing problems are overlooked or non-existing problems are detected due to lake of knowledge. Thus, problem detection in restructured deployment models cannot be done manually, but is required in several scenarios today.

The contributions of this paper to tackle these issues are (i) how problems can be detected in restructured deployment models using architecture and design patterns and (ii) how the problem detection can be automated based on formalizing these patterns. Architecture and design patterns describe proven solutions for recurring problems in a structured way (Meszaros and Doble, 1997; Wellhausen and Fiesser, 2012). Patterns for different domains have been captured, for example, for messaging systems (Hohpe and Woolf, 2004), cloud computing (Fehling et al., 2014), and security architectures (Schumacher et al., 2006). We apply this knowledge in form of patterns to restructured topology-based deployment models for problem detection. However, since a manual approach is not possible due to the discussion above, we developed a concept for the automated problem detection based on formalized problem and context descriptions of patterns. For the automated detection, we use the logic inference capability of logic programming languages and chose Prolog as basis for the formalization. To validate the feasibility of our approach, we selected the TOSCA standard as modeling language and developed a prototype for the automated detection of problems in restructured TOSCA topologies.

The paper is structured as follows: Section 2 introduces fundamentals and motivates our concepts. In Section 3 the pattern-based problem detection approach is presented. In Section 4 the pattern formalization used for an automated problem detection are introduced and in Section 5 a case study with selected patterns are presented. The system architecture and validation are described in Section 6. Finally Section 7 discusses related work and Section 8 concludes the paper.

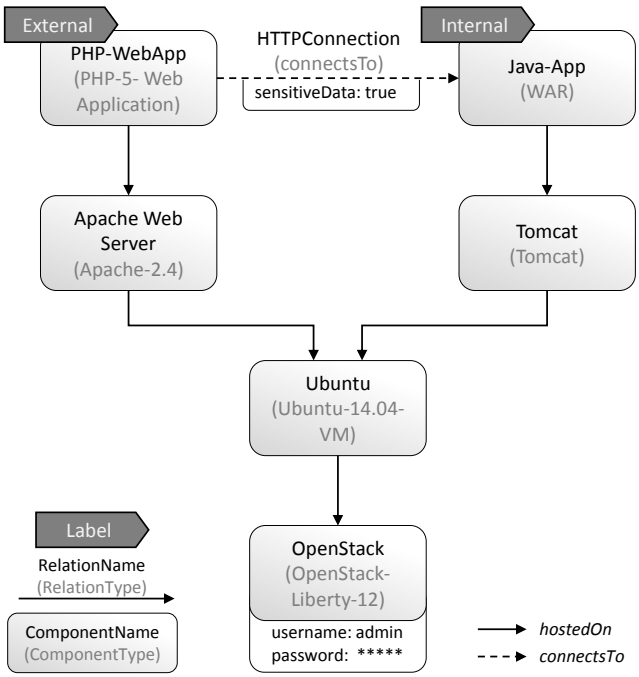

Fig. 1 Exemplary Topology-based Deployment Model

2 Motivation and Fundamentals

There are several deployment technologies that can process topology-based deployment models. This kind of models are declarative deployment models (Endres et al., 2017). A declarative deployment model describes the structure of an application that shall be deployed. This typically comprises the application's components, their relations, and specific configuration properties. This deployment model can be abstracted as *topology model*. Therefore, we first introduce basics about topology-based deployment models and then present a motivating scenario to better understand the effects of restructuring such topology-based deployment models.

2.1 Topology-Based Deployment Model

As explained above, a declarative deployment model can be abstracted as topology model. A topology-based deployment model is a directed graph and describes the *components* of an application with their *relations*. Each component and relation is of a certain *type* and can have specific configuration properties, such as login information. This is the canonical metamodel on the basis of which we explain our concepts.

Figure 1 depicts a topology-based deployment model. The *PHP-WebApp* establishes a *HTTPConnection* to the *Java-App*. As additional information, the characteristic of the exchange data, namely that it is sensitive data, is added as a property. The PHP-WebApp requires an

Apache Web Server while the Java-App is hosted on a *Tomcat*. They in turn are hosted on a single virtual machine *Ubuntu*, which is hosted on an *OpenStack* platform. In this example two relation types are distinguished: a *hostedOn* indicating that the target component serves as host and a *connectsTo* to represent a connection.

The OASIS standard TOSCA (OASIS, 2013) used for validating our concepts can be used to describe such topology-based deployment models. It provides a vendor- and technology-independent metamodel for declarative deployment models. It was therefore selected for the validation[3]. In addition to the modeling language, basis types are defined for components and relations, among other things, in order to assure a precise semantics for them (OASIS, 2016). This facilities a common understanding of components and relations and their processing as indented by the modeler.

2.2 Redistribution of application components

The deployment of an application, as shown in Figure 1 can change for various reasons, for example, if the application has to be deployed multiple times in different environments. This leads to a redistribution of the application's components. A redistribution can reflect a migration pattern as presented by Jamshidi et al. (2014). In a previous work, we presented an approach for the redistribution of application's components based on target labels (Saatkamp et al., 2017). For this, components can be annotated with *target labels*. These labels indicate the target environment of a component, for example, that a component shall be hosted on-premise or by a public cloud provider such as Amazon AWS.

In the example depicted in Figure 1, target labels are attached to the two application-specific components: PHP-WebApp and Java-App. The label *internal* attached to the Java-App indicates that it shall be hosted at the on-premise infrastructure of an enterprise, while the label *external* specifies that the PHP-WebApp shall be hosted by an external cloud provider, for example Amazon Web Services (AWS). Thus, this labeling describes a redistribution of the components in the deployment model. With the approach presented by Saatkamp et al. (2017) the deployment model is split according to the assigned target labels and appropriate hosting components are selected such as an EC2 component for AWS. However, problems can occur caused by the redistribution. The next section describes possibly occurring problems when restructuring the deployment model.

[3] A comparison of different cloud modeling languages can be found in Bergmayr et al. (2018)

2.3 Motivating Scenario

In Figure 1 an exemplary topology-based deployment model of an application with two communicating components hosted on one single virtual machine is shown. However, the distribution of the components can vary: For example, an application shall be deployed for several customers. Some of the customers may use external cloud providers, others rely on their own infrastructure. Another reason can be that an enterprise extends their IT infrastructure with offers from a public cloud provider or outsources parts of the IT and the deployments must be adjusted accordingly. These reasons can cause a restructuring of the application's components in the topology-based deployment model.

This kind of model adaptation can be carried out using the splitting approach described in the previous section (Saatkamp et al., 2017). However, the interconnections between components hosted in different environments can cause problems. In our example on the previous page, the distribution affects the communication between the PHP-WebApp and the Java-App. In the original topology-based deployment model the communication took place in the internal network. After the restructuring they use the public network. Thus, security issues, such as the encryption of data, authorization, or authentication can come up caused by the restructuring of the deployment, that were not relevant for a communication in the internal network. Furthermore, restrictions of the environments, for example, a disabled inbound communication by a firewall, can lead new problems. Therefore, the main contributions of our work are (i) to enable the problem detection in restructured topology-based deployment models and (ii) to automated this detection process to support human operators in detecting and solving deployment problems.

3 Pattern-based Approach to Detect Problems in Restructured Deployment Models

The restructuring of topology-based deployment models can causes new problems, which have to be detected by the operator to facilitate a valid and functional deployment of the application. In this section, we present an approach on how existing architecture knowledge can be applied to restructured topology-based deployment models to detect newly emerged problems caused by the restructuring. Before we detail our approach, first some details about architecture and design patterns capturing existing architecture knowledge are explained as they provide the basis for our problem detection approach.

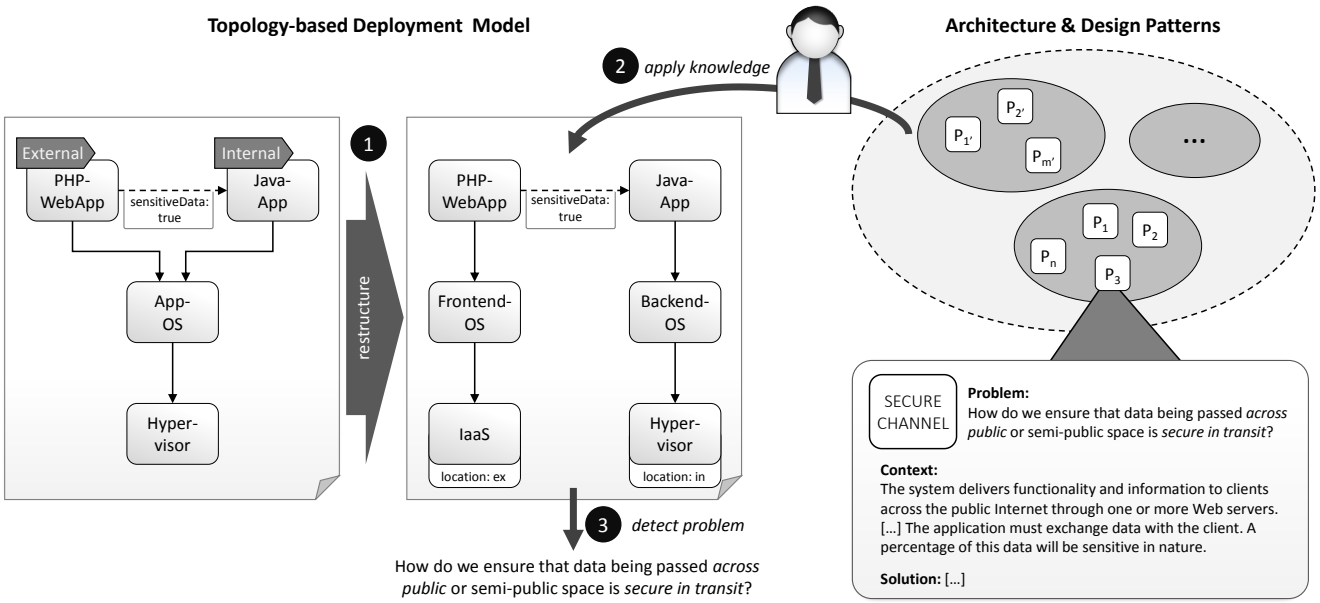

Fig. 2 Overview for the pattern-based problem detection approach in restructured topology-based deployment models

3.1 Architecture and Design Patterns

The concept of gathering architectural knowledge and best practices for recurring problems as structured patterns has been introduced by Alexander et al. (1977). They presented patterns for real architecture, i.e., towns, buildings, or one room of a building. Even though a pattern focuses on one specific problem, patterns are not independent from each other: a pattern can be, for example, a refinement (Falkenthal et al., 2015) or an alternative of another one (Buschmann et al., 1996). Thus, patterns can form a *pattern language*, which can be used to combine patterns to solve complex problems. This pattern approach is also adopted by several IT domains. Based on the knowledge of software architects and known solutions, best practices were collected by different communities in the form of patterns.

Architecture and design patterns are presented in several areas, for example, for general software architectures (Buschmann et al., 1996), messaging systems (Hohpe and Woolf, 2004), security architectures (Schumacher et al., 2006), and cloud computing (Fehling et al., 2014). These architecture and design patterns describe proven solutions for recurring problems independent of a specific technology or programming language. They provide generic solutions that can be applied to a variety of use cases in a different way.

In Figure 2 on the right an excerpt of the description of the Secure Channel pattern is shown (Schumacher et al., 2006). The pattern addresses the problem that sensitive sent over a public network, e.g. the Internet, can be eavesdropped. To ensure that such data remain confidential, an encrypted *secure channel* should be created. The pattern description is generic and applicable to several scenarios. Data confidentiality in transit, for example, can be enabled by different TLS or VPN protocols. The suitable technology for a specific scenario can vary, but the underlying concept is the same. Thus, by providing technology independent solutions such pattern can support human operators to solve problems in different scenarios.

The description of the domain knowledge is based on a pattern format that provides the structure for the pattern description. Even if these pattern formats vary slightly from pattern language to pattern language, the essential parts of a pattern format are similar: Each of the mentioned software pattern languages use a pattern format that describes inter ilia the *problem*, the *context*, and the *solution*. These parts are also contained in the guidelines for writing patterns provided by Meszaros and Doble (1997) and Wellhausen and Fiesser (2012).

3.2 Apply Architectural Knowledge to Restructured Topology-based Deployment Models

Existing architecture and design knowledge are provided as patterns in several domains. Patterns of one domain capture different viewpoints, i.e., the purpose and stakeholders are different. Even within one pattern language viewpoints often vary. This makes it difficult to identify relevant patterns for a specific purpose. For example, the security pattern language includes patterns such as the Secure Channel pattern for designing secure Internet applications as well as patterns at the strate-

gic level for selecting and integrating security services within an organization (Schumacher et al., 2006). The same applies to other pattern languages as well. The cloud computing patterns cover the characteristics of application workloads as well as integration concepts for distributed applications (Fehling et al., 2014). The different viewpoints covered by pattern languages open up a wide range of possible applications of patterns. Because there are so many languages and even more patterns a single operator (i) cannot be aware of all relevant patterns in his domain and therefore (ii) cannot be aware of all possible problems that may occur in his restructured deployment models.

The term architecture and design pattern makes it clear that the intention of these patterns is to support the creation of new software systems. However, the problem that sensitive data being sent over an insecure channel over a public network or the required inbound communication being restricted by a firewall addressed by the *Application Component Proxy* pattern can occur not only when creating a new system but also when restructuring a deployment model. Therefore, we apply architecture and design patterns to a new domain: restructured deployment models. This eases (i) the recognition of required adaptation caused by a redistribution as well as (ii) the solution of problems discovered in this way. In the next section, our concept for applying architecture and design patterns to restructured topology-based deployment models to detect problems and as a consequence to get solutions for these problems is presented.

3.3 Overview of the Pattern-based Problem Detection Approach

In this section, the pattern-based approach for detecting problems in restructured topology-based deployment models is described in detail. Based on this approach, our concept for automating the problem detection is presented in the following sections. Figure 2 shows our pattern-based problem detection approach. Caused by a restructuring, problems can occur in the deployment model that did not exist before (step 1). Knowledge about such recurring problems and best practice for these problems are captured in patterns in different pattern languages. This encompasses, for example, the security patterns or the cloud computing patterns. This knowledge in form of patterns can be applied to a restructured deployment model (step 2) to detect problems caused by the restructuring (step 3). Even though these architecture and design patterns are intended to be used during the creation of new software systems they also support solving problems in restructured deployment models as discussed in Section 3.2.

The example in Figure 2 on the left depicts a topology-based deployment model representing an application consisting of two components, a *PHP-WebApp* and a *Java-App* hosted on a single machine. The shown topology-based deployment model corresponds to the deployment model in Figure 1. The application shall be now deployed for a customer in a different environmental setup as it was used before. According to the customer requirements, the operator annotates the components with target labels. The application in this example shall be distributed across two environments: internal and external. Thus, by using the splitting method of Saatkamp et al. (2017) the deployment model is restructured according to the assigned target labels (step 1). After the restructuring the PHP-WebApp component is hosted by a suitable Infrastructure-as-a-Service (IaaS) and the Java-App on a private cloud platform. Due to the changes, the communication between the two different locations now takes place over a public network. This means that the communication network is not controlled by a single environment and the sensitive data can be eavesdropped if security mechanisms such as encryption technologies are not used. Thus, the restructuring added a problem to the deployment model in the form that sensitive data are exchanged using an unprotected communication channel. Fortunately, there is already a pattern that solves this problem (step 2): the Secure Channel pattern (Schumacher et al., 2006). It indicates that a encrypted secure channel should be created to solve this problem. Thus, by using known patterns for detecting problems in topology-based deployment models, the operator can use the generic knowledge from the pattern to solve the specific problem (step 3). Applying the existing knowledge to restructured topology-based deployment models eases the adaptation to enable a valid deployment. However, for this the knowledge must be accessible and usable.

Even if architecture and design pattern may help us understanding, detecting, and solving problems, a human operator can hardly be aware of all the patterns and pattern languages available and how they are interrelated with each other. Thus, although patterns describe the problems that should be solved, it is impossible to execute this approach manually as no operator can be aware of all patterns that describe possibly occurring problems. Therefore, the main challenge to be tackled is to *automatically* detect the problems that might occur in a restructured topology-based deployment model. An approach to formalize architecture and design patterns for automating the problem detection is presented in the following sections.

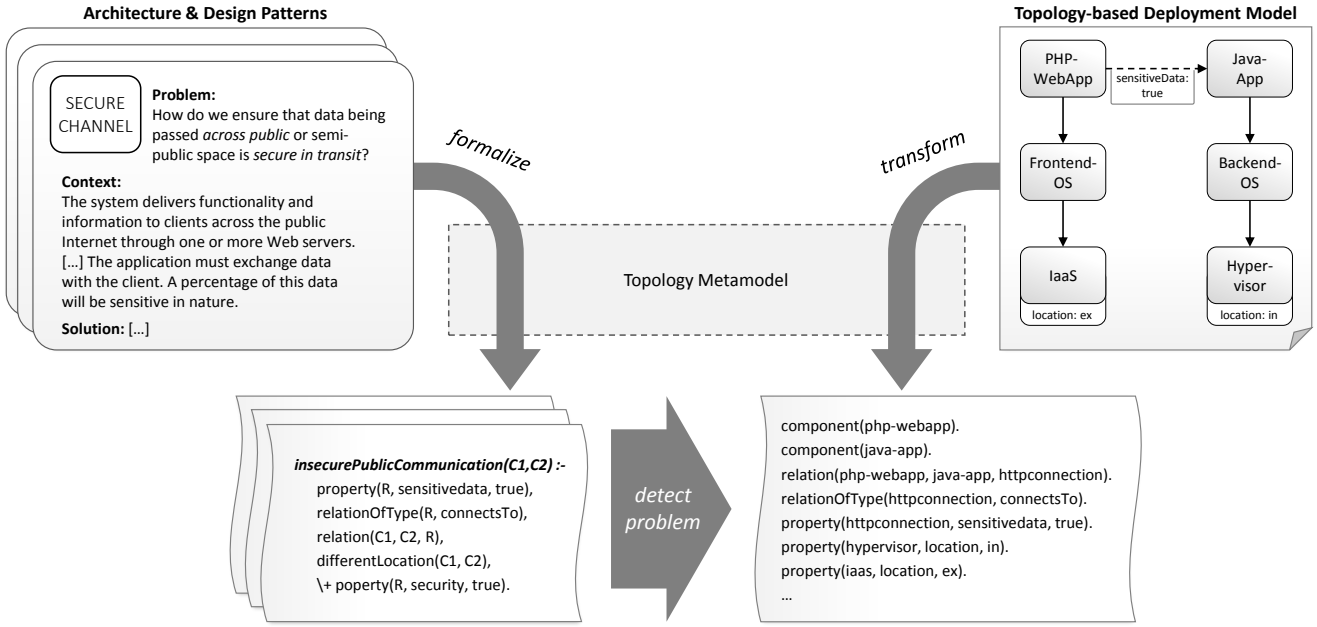

Fig. 3 Overview of the Patterm Formalization Approach based on Logic Programming

4 Automated Problem Detection by Problem- and Context-Formalized Patterns

The second contribution of this paper is the automation of the problem detection approach in restructured topology-based deployment models. For this, we present a concept based on *Problem- and Context-Formalized Patterns*. We provide an overview of our approach (Section 4.1) and introduce fundamentals about logic programming (Section 4.2). Finally, we present the topology-based deployment model metamodel required for the formalization and the transformation itself (Section 4.3).

4.1 Overview of the Pattern Formalization Approach

The manual detection of problems in restructured topology-based deployment models is not feasible due to the variety of possibly occurring problems. Therefore, the idea is to automatically recognize these problems based on formalized patterns in deployment models. To achieve automation, (1) the problem and context descriptions of patterns and (2) the topology-based deployment models are expressed as logical formulas. Consequently, it can automatically be detected whether the deployment model complies to the logic formula of a pattern. If this is the case, a problem is detected. This can be realized by logic programming. We use this in this paper to automatically detect problems in restructured deployment models. In a logic program, domain knowledge can be expressed as facts and rules and it can be checked whether a fact base satisfies the conditions of a rule.

Figure 3 describes the overall approach based on the example of the introduced Secure Channel pattern and the restructured topology-based deployment model depicted in Figure 2. In a first step, the problem and context description of each pattern are formalized as *rule* with conditions that have to be satisfied for the rule to be fulfilled. In this example, the rule *insecurePublicCommunication* formalizes the problem and context of the Secure Channel pattern. This pattern addresses the problem that sensitive data are exchanged over a public network using an unprotected channel. A detailed description of the Secure Channel pattern rule is shown in Section 5. All formalized patterns are stored in a repository to be used for detecting problems in topology-based deployment models. For this, in a second step, the elements of a deployment model to be investigated are transformed into *facts*. The resulting facts comprises all elements, i.e., components, relations and properties. In Figure 3 an excerpt of the fact base of the graphically represented deployment model is shown. Based on this fact base, all available pattern rules can be queries to detect problems in the deployment model.

Prolog is a widely used logic programming language and in previous works used for detection pattern solutions in UML diagrams (Bergenti and Poggi, 2002; Lim and Lu, 2006). We provide an overview on logic programming and Prolog in Section 4.2 and use it for expressing rules and facts in this work. In addition, we present the topology-based deployment model metamodel and the corresponding facts in Section 4.3.

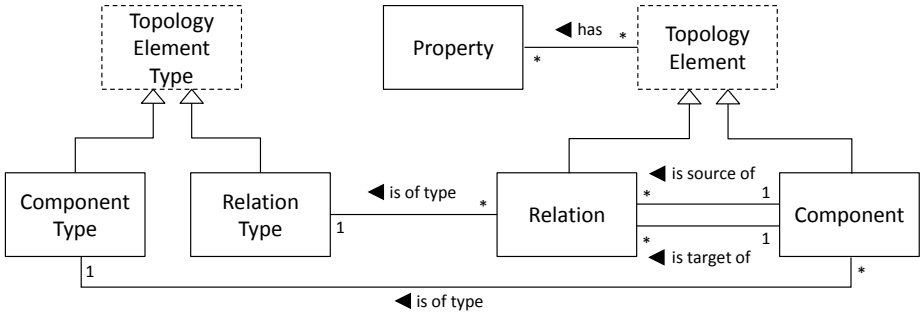

Fig. 4 Metamodel of Topology-based Deployment Models (Extended Metamodel Presented by Saatkamp et al. (2017))

4.2 Logic Programming for Problem Detection

Logic programming languages are part of the declarative programming languages that can be used to express what should be achieved rather than how it is achieved. Prolog as a widely used logic programming language is used to express the *Problem- and Context-Formalized* Patterns and the topology-based deployment models. It is based on horn clauses of the first-order logic and enables the representation of available knowledge as *facts* and *rules* (Clocksin and Mellish, 2003).

Facts describe specific circumstances, i.e., objects and their relationships. Based on facts, we can use queries either to prove that a fact is true or to infer new knowledge based on existing facts. Rules simplifies complex queries. Rules are extended facts with conditions that have to be satisfied for the rule to be fulfilled. The conditions of a rule can be linked with an AND or OR operator. The AND operator is expressed as "," and the OR operator as ";". As an example: The rule `son(X,Y):-father(Y,X),male(Y).` is fulfilled for the query *"Is John the son of Ben?"* if there exist the facts `father(ben,john)` and `male(ben)`. Rules can also be used to infer new knowledge based on existing facts. For example, the query *"Do objects exist for which `son(X,Y)` is fulfilled?"* provides the answer: `X = john` and `Y = ben`. Thus, facts and rules form a logic program and can be analyzed by an interpreter for a given query.

The rule `insecurePublicCommunication(C1,C2)` in Figure 3 is defined for the Secure Channel pattern. Results matching this rule are components that do not establish a secure channel in public communication. The query is interpreted on facts representing the deployment model to be investigated. For example, the fact `component(java-webapp)` denotes that `java-webbapp` is a component of the deployment model. The fact `relation(php-webapp,java-app,httpconnection)` denotes that it exists a relation `httpconnection` between the components `php-webapp` (source) and `java-app` (target). The facts as well as the rules are generated based on the metamodel presented in the next section.

4.3 Topology-based Deployment Model Metamodel And Corresponding Facts

The formalization of patterns to rules and the transformation of topologies to facts are based on a topology-based deployment model metamodel. Therefore, the formalization of patterns is specific to topology-based deployment models. This can be seen as the essential deployment model to which most deployment technologies can be mapped and thus forms a practical base for the formalization concept. In Figure 4 the underlying metamodel for the formalization and transformation is depicted. It is an extended version of the metamodel presented in our previous work (Saatkamp et al., 2017). A topology-based deployment model contains elements that are either *components* or *relations*. Each relation and component has a unique *id*. Relations additionally have a source and a target, which indicate the components that serve as start and endpoint of a relation, respectively. Relations and components are of a specific type. For this, *component types* and *relation types* are defined with a unique *id*. Types determine the semantic of the topology elements. Additional information about a topology element can be specified by *properties*. A property is related to one topology element and has a *key* representing the name of the property and a *value*.

Based on this metamodel corresponding facts can be expressed as follows: A component in a deployment model is specified as `component(component-id)`. For relations between component, the source and target component are important: `relation(source,target, relation-id)`. The topology element types must also be considered: `relationOfType(relation-id, type-id)` and `componentOfType(component-id,type-id)`. Last, the properties added to topology elements are expressed as `property(element-id,key,value)`.

Facts in this form can be queried based on the formalized patterns to detect problems in a deployment model. In the following the problem and context statements of two patterns are formalized and the applicability to detect a problem in a given deployment model is shown.

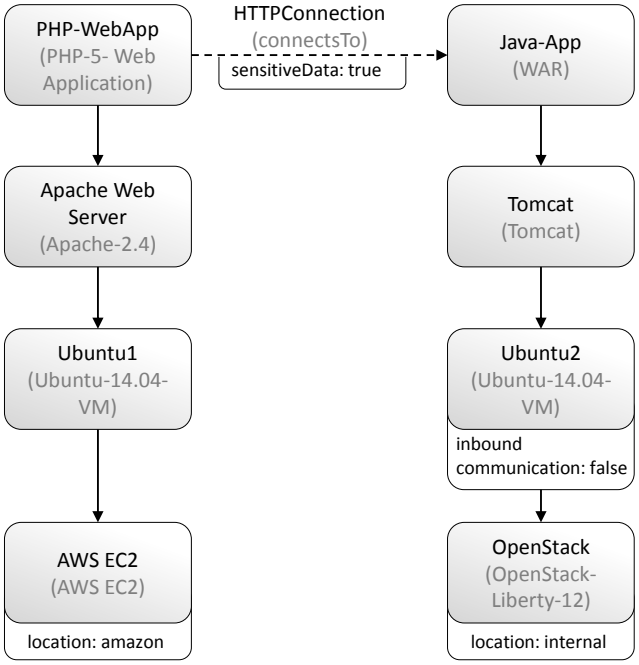

Fig. 5 Deployment Model with Problems

5 Case Study

In this section two *Problem- and Context-formalized Patterns* for topology-based deployment models are presented. By applying the presented approach, Prolog rules for each pattern are created based on their problem and context description. We applied the approach to the presented Secure Channel pattern as part of the security pattern language by Schumacher et al. (2006) and the Application Component Proxy pattern from the cloud computing pattern language by Fehling et al. (2014).

Figure 5 presents a minimal working deployment model we use as running example: There exists an insecure communication between the PHP-WebApp and the Java-App (which should be a Secure Channel) and the direct communication is restricted (Application Component Proxy). Based on this, in the following the patterns and their formalizations are described in detail.

5.1 Secure Channel

The Secure Channel pattern is part of the security pattern language by Schumacher et al. (2006). The problem and context are described as follows:

Problem:
> How do we ensure that data being passed across public or semi-public space is secure in transit?

Context:
> The system delivers functionality and information to clients across the public Internet through one or more web servers. [...] The application must exchange data with the client. A percentage of this data will be sensitive in nature.

Solution:
> Create secure channels for sensitive data that obscure the data in transit. Exchange information between client and server to allow them to set up encrypted communication between themselves. [...]

All essential statements of the problem and context description of the pattern must be formalized based on the metamodel introduced in Figure 4. These facts are the *conditions* for the pattern rule. For each statement described in natural language in the pattern description, corresponding *conditions* that have to be fulfilled for this rule must be extracted. In the following, the pattern description is decomposed and for each statement the corresponding condition is defined.

The context description states that sensitive data are exchanged. In a deployment model, this can be expressed as property. Thus, following fact serves as condition for the pattern rule: `property(R,sensitivedata,true)`. Data can just be exchanged if a connection is established. Thus, a relation of type *connectsTo* must be contained in a topology. For this, two facts are used. One for the relation itself: `relation(C1,C2,R)` and one denoting the type: `relationOfType(R,connectsTo)`. An additional condition is that the two components for which this relation shall be established communicate over a public network, i.e., they are located in different locations: `differentLocations(C1,C2)`. This fact in turn is a rule encapsulating a complex query (not shown for brevity). The last condition is that two components communicate over an insecure channel. This is the case if security mechanisms are missing. By the fact `not(property(R,security,true))` it is expressed that this property cannot be found in the facts describing the deployment model. The problem described by the Secure Channel pattern can thus expressed with the following Prolog rule:

```
insecurePublicCommunication(C1, C2) :-
property(R, sensitivedata, true),
relationOfType(R, connectsTo),
relation(C1, C2, R),
differentLocations(C1, C2),
not(property(R, security, true)).
```

This rule can be used and applied to the deployment model in Figure 5 and based on that the problem is automatically detected. This problem can then be solved by the solution described by the pattern.

5.2 Application Component Proxy

To show the application in other pattern language domains, we use the cloud integration pattern *Application Component Proxy* included in the cloud computing pattern language by Fehling et al. (2014). An extract of the problem and context description is as follows:

Problem:
> How can an application component be accessed if direct access to its hosting environment is restricted?

Context:
> Application components of a distributed application are deployed in different cloud environments that form a hybrid cloud. These environments often have different privacy, security, and trust properties. [...] However, application components hosted in unrestricted environments, for example, a public cloud, may have to access application components hosted in a restricted environment, for example a private cloud or corporate data center, but direct access may be unavailable. [...]

Solution:
> The interface of a restricted application component is duplicated to form a proxy component. Synchronous and asynchronous communications with the proxy component is initiated and maintained from the restricted environment [...]

From the above given description of this pattern, similarities to the `insecurePublicCommunication` rule that formalizes the problem and context of the Secure Channel pattern can be seen: a component wants to access another component in a different environment. Thus, parts of the `insecurePublicCommunication` rule can be reused: `relationOfType(R,connectsTo)`, `relation(C1,C2,R)`, and `differentLocations(C1,C2)`. Furthermore, an essential statement is that the environment of the accessed component is restricted, i.e., that no inbound communication from the outside is allowed. The resulting rule for the formalized Application Component Proxy is the following:

```
directAccessToRestrictedEnvironment(C1, C2) :-      1
     relationOfType(R, connectsTo),                 2
     relation(C1, C2, R),                           3
     differentLocations(C1, C2),                    4
     property(H, inboundcommunication, false),      5
     hosting_stack(S),                              6
     member(C2, S),                                 7
     member(H, S).                                  8
```

The last four *conditions* are required to check whether a component in the hosting stack of the accessed component set the property *inboundcommunication* to false. The hosting stack is stored as a list and copied into `S` (line 6). The accessed components as well as the component with the property must be in the same stack `S`. This problem is also contained in the deployment model in Figure 5 and can automatically detected using this formalized pattern.

6 Validation based on TOSCA

The automated problem detection approach for restructured topology-based deployment models based on *Problem- and Context-Formalized* patterns is validated using TOSCA. We want to show how our approach can be used to detect problems in TOSCA topologies. The Topology and Orchestration Specification for Cloud Applications (TOSCA) is an OASIS standard to describe the deployment and management of applications in a portable manner (OASIS, 2013). It is a vendor-neutral, technology-independent topology-based metamodel that can be mapped to several existing technologies. In the following, we present in Section 6.1 the mapping of our metamodel to TOSCA. This is fundamental in order to apply the formalized patterns to TOSCA. In Section 6.2 the overall architecture of a problem detection framework for TOSCA is presented. Our prototypically implementation is shortly described in Section 6.3.

6.1 Mapping to TOSCA

For the mapping of TOSCA elements to the elements in the topology-based deployment model metamodel all not relevant aspects of TOSCA are skipped. For more details see the TOSCA specification (OASIS, 2013).[4]

In TOSCA, *topology templates* are used to specify the structure of an application. The application's components are modeled as *node templates* and their relations as *relationship templates*. This corresponds to the topology and its topology elements: components and relations. The semantic of the node and relationship templates is defined by their types. TOSCA has an extensive type system enclosing all TOSCA elements. Consequently, *node types* and *relationship types* can be defined and used in topology templates as node templates and relationship templates. Types for relations and components are also part of the our metamodel used to specify the semantic of topology elements. In the TOSCA Simple Profile Specification additionally normative types are defined (OASIS, 2016). These types encompass general types that have to be available and interpretable in each TOSCA-conform deployment engine. They include the

[4] The mapping to TOSCA is based on the XML specification of TOSCA, but the concepts can also be applied to the TOSCA Simple Profile (OASIS, 2016)

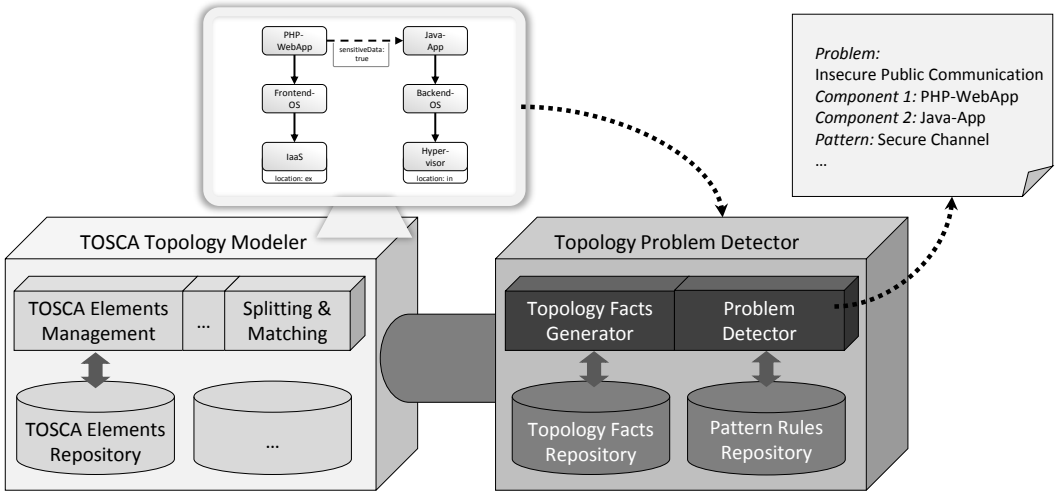

Fig. 6 System Architecture of the Topology Problem Detector used for TOSCA Topologies

relationship types *hostedOn* and *connectsTo* as well as node types such as *WebApplication* or *WebServer*. These predefined types ease the interpretation of node and relationship templates. This semantical knowledge can also be used for the transformation of TOSCA topologies.

In TOSCA properties that are defined for a specific type can be assigned to the respective node template or relationship template with a specific value. This corresponds to the properties assigned to topology elements. In our metamodel properties are restricted to key-value pairs which is the most commonly-occurring form of properties. These are the most important elements of TOSCA to apply the presented approach to TOSCA topologies. In the following the prototype and the processing steps are explained in detail.

6.2 System Architecture of a Pattern-based Problem Detection Framework for TOSCA

In the previous section, the mapping between TOSCA elements and our introduced metamodel is described. In this section, the architecture of a framework for problem detection in TOSCA topologies is presented.

Figure 6 shows the system architecture: On the left the TOSCA Topology Modeler and on the right the *Topology Problem Detector* are depicted. The architecture of the Topology Modeler is depicted in a simplified manner to focus on the important components for our approach. A more detailed view is given by Kopp et al. (2013). The Modeler is used to graphically model a topology template as directed graph with node templates, relationship templates, and attached properties. All TOSCA elements such as the node types or topology templates are management by the *TOSCA Elements Management* component and stored in the *TOSCA Elements Repository*. The *Splitting & Matching* component are used to split a topology template according to the assigned target labels (Saatkamp et al., 2017).

For detecting problems in a split TOSCA topology the *Topology Problem Detector* is used. It consists of two components: the *Topology Facts Generator* for TOSCA topologies and the *Problem Detector*. Patterns that have been formalized as rules by their problem and context description are stored in the *Pattern Rules Repository*. The authoring process for formalizing patterns is a manual process. However, this has to be done just once for each pattern and can be reused to check arbitrary topologies. The Topology Facts Generator transforms the TOSCA topology into a set of corresponding facts that are stored in the *Topology Facts Repository*.

For the problem recognition the Problem Detector component starts the Prolog Interpreter and loads the specific topology from the Topology Facts Repository and all available patterns from the Pattern Rules Repository. All rules are queried by the Interpreter. Thus, all contained problems and the patterns solving these problem are detected in the TOSCA topology. In the example in Figure 6, the `InsecurePublicCommunication` problem between the PHP-WebApp and the Java-App is detected that can be solved by the Secure Channel pattern. A detailed view on the underlying topology and the formalized pattern are given in Figure 3. In the following section our prototypical implementation for the concept validation is briefly described.

6.3 Prototypical Implementation

For the concept validation, we prototypically implemented the pattern-based problem detection framework

based on the existing TOSCA modeling tool *Winery*[5] and the newly developed *Topology ProDec*[6] for the problem detection in TOSCA topologies. The Topology ProDec uses the Prolog Interpreter SWI Prolog[7]. Each pattern is stored as Markdown file containing a textual description of the pattern and the Prolog rule. During runtime all available patterns are loaded and the Prolog rules are stored in a Prolog file.

For the test setup we have stored four patterns including the described patterns in Section 5 in the *Pattern Rules Repository*. We have used three TOSCA topologies, each with a different number of components: 10, 20 and 30 components. Each of them had n-1 relations. For each topologies two variants existed: one containing the problems `insecurePublicCommunication` and `directAccessToRestrictedEnvironment` and one without detectable problems. The topologies result in 45 facts (10 components), 87 facts (20 components), and 132 facts (30 components). We have evaluated the time required to read the rules from the markdown files, to transform the topology to facts, and to query all problem rules against the facts. The mean value for the process (loading Prolog rules, transforming the topology, querying the rules) for 10 repetitions result in the following: The problem detection for 10 components took 1.0 sec, for 20 components 1.8 sec, and for 30 components 2.6 sec. The result for the topologies without problems was similar: 1.1 sec, 2.0 sec, and 2.5 sec. Even if the performance could be better, it is not of great importance because the problem detection is done during design time and not during runtime. However, preloading of the patterns and topologies could improve the performance but this is considered in future work.

Our approach facilitates the automated problem detection in deployment models and thus also to identify patterns to solve these problems. For applying the concept the patterns must be formalized as Prolog rules. Even though creating these rules is hard and time-consuming, it has to be done only once and an operator no longer needs to know all patterns and potential problems. The usage of our formalized patterns is restricted to the problem detection in topology-based deployment models because of the underlying metamodel. However, this metamodel defines elements relevant for all deployment systems such as the components and their configurations. Of course, this approach can also be extended to specific characteristics of individual deployment systems, but the general idea remains the same.

[5] `https://eclipse.github.io/winery`
[6] `https://github.com/saatkamp/topology-prodec`
[7] `http://www.swi-prolog.org/`

7 Related Work

Several works address the recognition of patterns in UML diagrams (Di Martino and Esposito, 2016; Fontana and Zanoni, 2011; Kampffmeyer and Zschaler, 2007; Lim and Lu, 2006; Bergenti and Poggi, 2002). However, all of them focus on the recognition of solution structures based on the design patterns described by Gamma et al. (1994). We focus on the problem detection in restructured topology-based deployment models and we consider different pattern languages as several domains such as security and cloud computing are relevant for functional deployment models.

For detecting pattern structures, different methods have been used. Kampffmeyer and Zschaler (2007) presented an ontology-based approach with OWL, Di Martino and Esposito (2016) generated Prolog rules based on ODOL+OWL-representations of patterns, Lim and Lu (2006) and Bergenti and Poggi (2002) represented the solution structure and behavior as Prolog rules, and Fontana and Zanoni (2011) represented patterns as design pattern diagrams. Taibi and Ngo (2003) introduced the BPSL language that is based on the first-order logic to describe patterns. However, they also focus on the solution aspect of patterns only. Cortellessa et al. (2014) present an approach for detecting performance antipatterns in UML diagrams based on performance indices to improve the system performance. Antipatterns capture bad practices compared to patterns that describes best practices. Similar to the works detecting pattern structures, they detect antipattern structures.

Kim and Khawand (2007) formalize the problem domain of the design patterns for object-oriented software. They solely consider structural elements of the pattern as UML diagrams. By only comparing structural elements, the absence of elements cannot be checked. This is why we used logical programming in this work.

The concepts presented by Breitenbücher et al. (2013a, 2014a,b) and Breitenbücher (2016) are also based on structural elements of management patterns in order to apply them to enterprise topology graphs that represent the current state of an application through its components and relations. To identify if a certain management pattern is applicable, a target topology fragment is defined. If this fragment is detected in a topology, the management pattern is applicable. However, with these topology fragments only existing structural elements in a topology can be detected. Similar approaches to detect structures in topologies to facilitate predefined transformation steps are presented by Arnold et al. (2007) and Eilam et al. (2006). Guth and Leymann (2018) use graph fragments to detect subgraphs in architectural graphs that can be rewrote or refined. However, their

approaches are also based on graph matching and the absence of elements cannot be detected. This is important for detecting problems.

Zdun and Avgeriou (2005) define pattern primitives that represents the smallest logical entities used to model architecture patterns. These reusable entities can be used to model different patterns. This concept is applied to annotate software components to detect possible patterns that can be applied to these components (Haitzer and Zdun, 2015).

As Prolog enables the detection of the absence of elements, we decided to use Prolog rules for the formalization of the problem and context domain of patterns. In addition, in this work we focus on a specific domain of patterns application, topology-based deployment models, rather than on a specific pattern domain.

8 Conclusion

In this paper, we applied architecture and design patterns to restructured topology-based deployment models to detect problems that prevent a valid deployment. For this, we presented (i) how problems can be detected in restructured deployment models based on architecture and design patterns and (ii) how the problem detection can be automated by formalizing the problem and context stated by a pattern. Even though these patterns are intended to be used for creating new software systems, we demonstrated the applicability of these patterns for problem detection in restructured deployment models. Because detecting problems is a highly non-trivial challenge, we further introduced a formalization approach for the problem and context description of patterns to automated the problem detection. For validating our concepts, we prototypically implemented the problem detection framework for TOSCA topologies. An extended validation is presented by Saatkamp et al. (2018).

This approach is not limited to the presented patterns. It could also be applied, for example, to detect problems in distributed data sources (Strauch et al., 2013) and can be extended to non-restructured topology-based deployment models for their validation. This will be investigated in future works. We plan to extend this approach not only for the automated detection of problems but also for solving the detected problems. Furthermore, we want to improve the authoring process of the patterns by a semantic model and want to extend the tool support to ease the creation of new rules.

Acknowledgements This work was partially funded by the BMWi projects *IC4F* (01MA17008G) and *SmartOrchestra* (01 MD16001F) and the German Research Foundation (DFG) project ADDCompliance (636503).

References

Alexander C, Ishikawa S, Silverstein M (1977) A Pattern Language: Towns, Buildings, Construction. Oxford University Press

Arnold W, Eilam T, Kalantar M, Konstantinou AV, Totok AA (2007) Pattern Based SOA Deployment. In: Proceedings of the Fifth International Conference on Service-Oriented Computing, Springer, pp 1–12

Bergenti F, Poggi A (2002) Improving UML Designs Using Automatic Design Pattern Detection, World Scientific, pp 771–784

Bergmayr A, Breitenbücher U, Ferry N, Rossini A, Solberg A, Wimmer M, Kappel G, Leymann F (2018) A systematic review of cloud modeling languages. ACM Computing Surveys (CSUR) 51(1):22:1–22:38

Breitenbücher U (2016) Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements. Dissertation, University of Stuttgart, Faculty 5

Breitenbücher U, Binz T, Kopp O, Leymann F (2013a) Pattern-based Runtime Management of Composite Cloud Applications. In: Proceedings of the 3rd International Conference on Cloud Computing and Services Science, SciTePress, pp 475–482

Breitenbücher U, Binz T, Kopp O, Leymann F, Wettinger J (2013b) Integrated Cloud Application Provisioning: Interconnecting Service-Centric and Script-Centric Management Technologies. In: On the Move to Meaningful Internet Systems, Springer, pp 130–148

Breitenbücher U, Binz T, Kopp O, Leymann F (2014a) Automating Cloud Application Management Using Management Idioms. In: Proceedings of the 6th International Conferences on Pervasive Patterns and Applications, Xpert Publishing Services, pp 60–69

Breitenbücher U, Binz T, Kopp O, Leymann F, Wieland M (2014b) Context-Aware Cloud Application Management. In: Proceedings of the 4th International Conference on Cloud Computing and Services Science, SciTePress, pp 499–509

Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M (1996) Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. Wiley

Clocksin WF, Mellish CS (2003) Programming in Prolog. Springer

Cortellessa V, Marco AD, Trubiani C (2014) An approach for modeling and detecting software performance antipatterns based on first-order logics. Software & Systems Modeling 13(1):391–432

Di Martino B, Esposito A (2016) A rule-based procedure for automatic recognition of design patterns in uml diagrams. Software: Practice and Experience

46(7):983–1007

Eilam T, Kalantar M, Konstantinou A, Pacifici G, Pershing J, Agrawal A (2006) Managing the configuration complexity of distributed applications in Internet data centers. Communications Magazine 44(3):166–177

Endres C, Breitenbücher U, Falkenthal M, Kopp O, Leymann F, Wettinger J (2017) 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, pp 22–27

Falkenthal M, Barzen J, Breitenbücher U, Fehling C, Leymann F, Hadjakos A, Hentschel F, Schulze H (2015) Leveraging Pattern Application via Pattern Refinement. In: Proceedings of the International Conference on Pursuit of Pattern Languages for Societal Change, epubli

Fehling C, Leymann F, Retter R, Schupeck W, Arbitter P (2014) Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications. Springer

Fontana FA, Zanoni M (2011) A tool for design pattern detection and software architecture reconstruction. Information sciences 181(7):1306–1324

Gamma E, Helm R, Johnson R, Vlissides J (1994) Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley

Guth J, Leymann F (2018) Towards Pattern-based Rewrite and Refinement of Application Architectures. In: Proceedings of the 12th Advanced Summer School on Service Oriented Computing, IBM Research Division

Haitzer T, Zdun U (2015) Semi-automatic architectural pattern identification and documentation using architectural primitives. Journal of Systems and Software 102:35–57

Hohpe G, Woolf B (2004) Enterprise integration patterns: Designing, building, and deploying messaging solutions. Addison-Wesley Professional

Jamshidi P, Pahl C, Chinenyeze S, Liu X (2014) Cloud Migration Patterns: A Multi-Cloud Service Architecture Perspective. In: Service-Oriented Computing - ICSOC 2014 Workshops, Springer, pp 6–19

Kampffmeyer H, Zschaler S (2007) Finding the pattern you need: The design pattern intent ontology. In: International Conference on Model Driven Engineering Languages and Systems, Springer, pp 211–225

Kim DK, Khawand CE (2007) An approach to precisely specifying the problem domain of design patterns. Journal of Visual Languages and Computing 18(6):560–591

Kopp O, Binz T, Breitenbücher U, Leymann F (2013) Winery – A Modeling Tool for TOSCA-based Cloud Applications. In: Proceedings of the 11th International Conference on Service-Oriented Computing, Springer, pp 700–704

Lim DK, Lu L (2006) Inference of design pattern instances in uml models via logic programming. In: 11th IEEE International Conference on Engineering of Complex Computer Systems, IEEE, pp 10–29

Meszaros G, Doble J (1997) MetaPatterns: A Pattern Language for Pattern Writing. In: Proceedings of International Conference on Pattern languages of program design, pp 164–200

OASIS (2013) Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0

OASIS (2016) TOSCA Simple Profile in YAML Version 1.0

Saatkamp K, Breitenbücher U, Kopp O, Leymann F (2017) Topology Splitting and Matching for Multi-Cloud Deployments. In: Proceedings of the 7th International Conference on Cloud Computing and Services Science, SciTePress, pp 247–258

Saatkamp K, Breitenbücher U, Kopp O, Leymann F (2018) Application Scenarios for Automated Problem Detection in TOSCA Topologies by Formalized Patterns. In: Proceedings of the 12th Advanced Summer School on Service Oriented Computing, IBM Research Division

Schumacher M, Fernandez-Buglioni E, Hybertson D, Buschmann F, Sommerlad P (2006) Security Patterns: Integrating Security and Systems Engineering. John Wiley & Sons, Inc.

Strauch S, Andrikopoulos V, Breitenbücher U, Sáez SG, Kopp O, Leymann F (2013) Using Patterns to Move the Application Data Layer to the Cloud. In: Proceedings of the 5th International Conference on Pervasive Patterns and Applications, Xpert Publishing Services, pp 26–33

Taibi T, Ngo DCL (2003) Formal specification of design patterns - a balanced approach. Journal of Object Technology 2(4):127–140

Wellhausen T, Fiesser A (2012) How to Write a Pattern?: A Rough Guide for First-time Pattern Authors. In: Proceedings of the 16th European Conference on Pattern Languages of Programs, ACM

Zdun U, Avgeriou P (2005) Modeling Architectural Patterns Using Architectural Primitives. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM, pp 133–146