Institute of Architecture of Application Systems

Application Scenarios for Automated Problem Detection in TOSCA Topologies by Formalized Patterns

Karoline Saatkamp[1], Uwe Breitenbücher[1], Oliver Kopp[2], and Frank Leymann[1]

[1]Institute of Architecture of Application Systems, University of Stuttgart, Germany
[firstname.lastname]@iaas.uni-stuttgart.de

[2]Institute of Parallel and Distributed Systems, University of Stuttgart, Germany
kopp@ipvs.uni-stuttgart.de

BibTeX:

```
@inproceedings{SaatkampApplicationScenarios2018,
  author    = {Saatkamp, Karoline and Breitenb\"{u}cher, Uwe and Kopp, Oliver
               and Leymann, Frank},
  title     = {Application Scenarios for Automated Problem Detection in TOSCA
               Topologies by Formalized Patterns},
  booktitle = {Papers From the 12th Advanced Summer School on Service-Oriented
               Computing (SummerSOC'18)},
  year      = {2018},
  pages     = {43--53},
  publisher = {IBM Research Division}
}
```

The full version of this publication has been presented as a poster at the Advanced Summer School on Service Oriented Computing (SummerSOC 2018).
http://www.summersoc.eu

Universität Stuttgart
Germany

Application Scenarios for Automated Problem Detection in TOSCA Topologies by Formalized Patterns

Karoline Saatkamp[1], Uwe Breitenbücher[1], Oliver Kopp[2], and Frank Leymann[1]

[1] Institute of Architecture of Application Systems, University of Stuttgart
[2] Institute for Parallel and Distributed Systems, University of Stuttgart
Universitätsstraße 38, 70569 Stuttgart, Germany
[lastname]@informatik.uni-stuttgart.de

Abstract. There are several reasons why application components are redistributed to multiple environments: parts of the IT-infrastructure are outsourced or an application has to be deployed to different customers with different infrastructures. For an automated deployment, several technologies and standards already exist. With the TOSCA standard the deployment of an application can be modeled as topology representing the application's components and their relations. When such a topology-based deployment model is redistributed, problems can arise that were not present before, such as firewalls that prevent direct access to a component. Problems can be detected based on problem- and context-formalized patterns. In this paper, we present application scenarios for the pattern formalization approach to detect problems in restructured topology-based deployment models based on selected cloud computing patterns.

Keywords: Cloud Computing Patterns, Formalization, Prolog, TOSCA.

1 Introduction and Background

Over the last years, several technologies for the deployment and management of cloud applications, such as Docker[1], Kubernetes[2], or Cloud Foundry[3], have been developed. Besides these vendor-specific technologies, standards, such as the OASIS standard TOSCA (Topology and Orchestration Specification for Cloud Applications) [24], are published to describe the deployment and management of cloud applications in a vendor-independent manner. TOSCA enables the description of topology-based deployment models specifying an application's structure by its components and their relations [12]. Such topology-based deployment models are declarative deployment models that can be interpreted by a runtime that infers the deployment logic from the structural description [12]. Imperative models, on the other hand, define a process that specifies the deployment logic explicitly [12].

[1] https://www.docker.com/
[2] https://kubernetes.io/
[3] https://www.cloudfoundry.org/

TOSCA topologies are topology-based deployment models, describing the application's components and their relationships. Components can be application-specific components such as a PHP WebApp as well as middleware components such as a Tomcat or infrastructure components such as an OpenStack. A component can be, for example, a *PHP WebApp* that is *hosted on* an *Apache Web Server*, as depicted in **Fig. 1**. The *hostedOn* or *connectsTo* relations describe the relationships between the components. Due to several reasons, the components of an application might have to be redistributed to different environments: parts of the IT are outsourced or an application has to be deployed for different customers with different environmental conditions. Based on the *Split and Match* method introduced in a previous work [27], each application-specific component can be annotated with a target label that specifies the intended target location of the component. According to these labels, the topology-based deployment model is split and matched with the available infrastructure or platform service in the target location and this results in a restructured deployment model. In the example depicted in **Fig. 1**, two components that are formerly hosted on the same virtual machine can be redistributed, for example, to an AWS EC2 and an OpenStack.

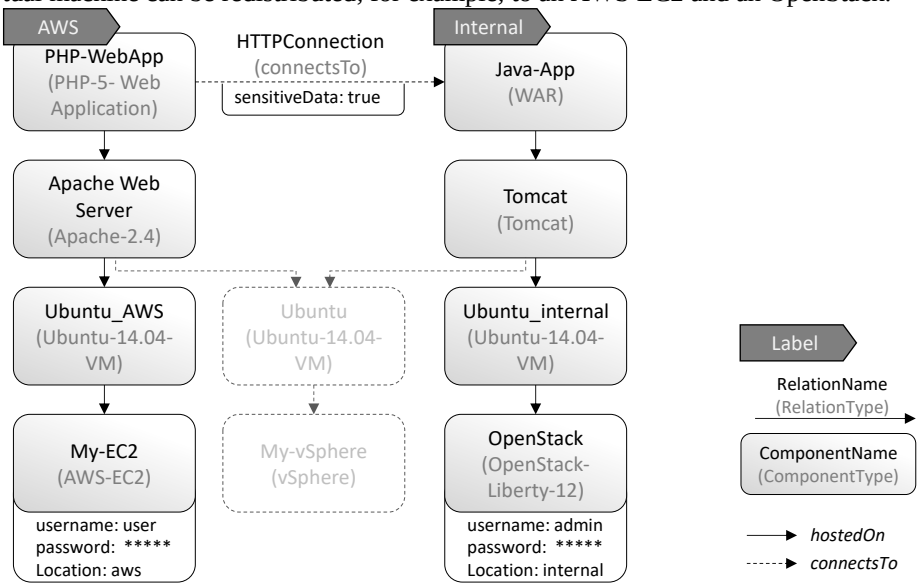

Fig. 1. Example of a restructured topology-based deployment model

In the initial topology in **Fig. 1** the *Apache Web Server* and the *Tomcat* are intended to be hosted on the same virtual machine *Ubuntu*, as shown by the dashed greyed out components. Based on the attached target labels *AWS* and *Internal*, the deployment model is split into two separate stacks. For this, the virtual machine is duplicated for each target location and the available infrastructure components from the target locations are selected and injected into the topology-based deployment model.

The restructuring of such deployment models can result in problems that have not existed before. In the example presented in **Fig. 1**, sensitive data that used to be exchanged within a single virtual machine is now exchanged over the internet, which can

lead to security issues. Furthermore, communication restrictions or incompatibilities can occur. To detect such problems in an automated manner, we presented an approach to automatically detect problems in restructured deployment models based on formalizing architecture and design patterns [26]. This concept is based on existing design and architecture knowledge in the form of patterns that describe best-practice solutions for recurring problems in a certain context [1]. Such architecture and design patterns are discovered and described in several domains, for example, for general architecture solutions [9], application integration [17], security mechanisms [28], and for cloud computing [13]. Independent of the domain, a pattern describes the *problem* solved by this pattern, the *context* when the problem occurs, and the *solution* in a technology-independent manner. However, patterns are only captured as textual descriptions.

To enable an automated problem detection, Saatkamp et al. [26] presented a pattern formalization approach to formalize the problem and context description of a pattern. In previous work [26], the applicability of the formalization approach has been presented for two patterns: the *Secure Channel* pattern which is part of the security pattern language by Schumacher et al. [28] and the *Application Component Proxy* which is part of the cloud computing pattern language by Fehling et al. [13]. For a prototypical validation the TOSCA standard has been chosen, because it is a generic standard and independent from a certain technology or provider [4]. A TOSCA topology-based deployment model is described as *Topology Template*. The application's components are specified as *Node Templates* and their relations as *Relationship Templates*. The semantic of these elements is specified by their *Node Type* or *Relationship Type* respectively. In this paper, we extend the validation of the problem detection approach based on formalized patterns [26] by further patterns. The TOSCA-based prototype, presented in the previous paper is used as basis for the validation.

The remainder of this paper is structured as follows: Section 2 gives on overview of the problem detection approach while Section 3 describes the application scenarios. In Section 4 related work is discussed and Section 5 concludes this paper.

2 Problem Detection Approach Overview

We presented the approach to automatically detect problems in restructured deployment models based on formalizing architecture and design patterns a previous work [26]. This is based on the concept of applying architectural and design knowledge in terms of patterns to restructured topology-based deployment models. The textual description of such patterns is based on a pattern format. Even if pattern formats differ slightly between different pattern languages, the essential parts are the same [23,30]: (i) the *problem* section that describes the problem solved by the pattern, (ii) the *context* section that describes the context in which the problem arises, and (iii) the *solution* section that gives a technology-independent description of the best-practice solution. For the detection of problems occurring in restructured topology-based deployment models, the problem as well as context description are of main interest. However, for an automated approach each pattern has to be formalized in a machine-readable manner. Thus, the

logic programming language Prolog is used to express the problem and context of patterns as *rules* that can be applied to topologies which are expressed as *facts*.

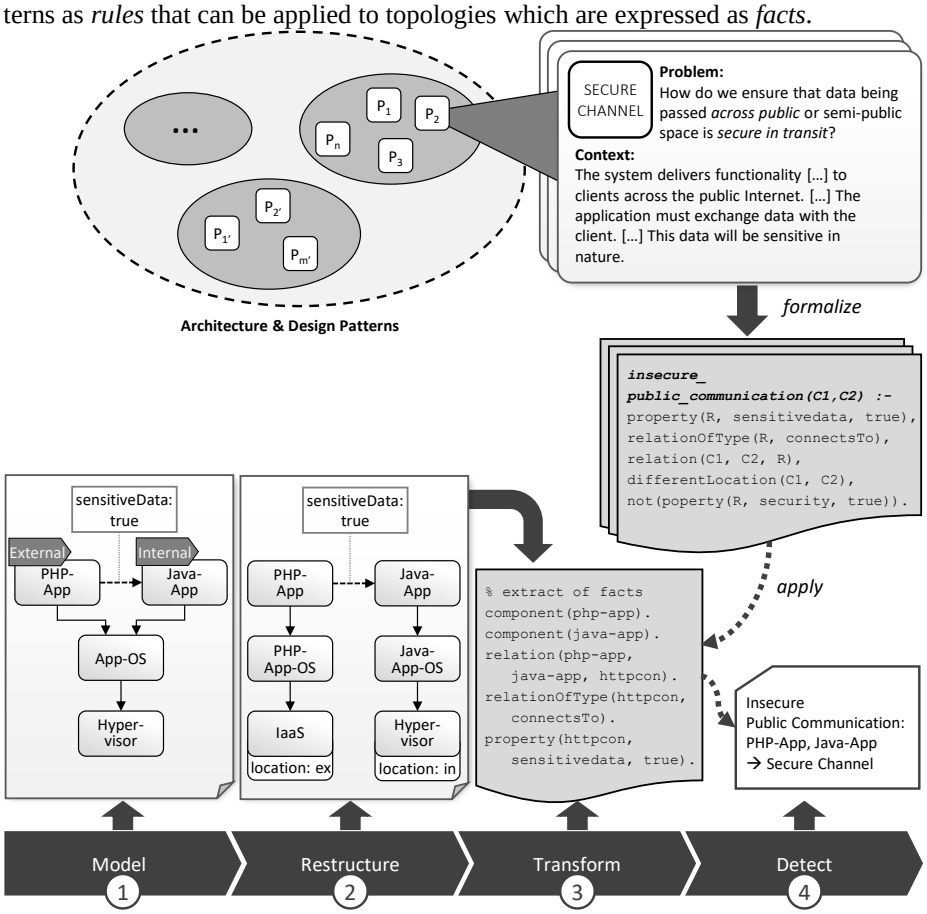

Fig. 2. Overview of the Problem Detection Approach Based on Formalized Patterns [26]

In **Fig. 2** an overview of the problem detection approach based on formalized patterns is depicted. At the top of the figure the different pattern languages are sketched. Each area represents a pattern language, for example, the enterprise integration patterns [17], the cloud computing patterns [13], or the security patterns [28] which also include the *Secure Channel* pattern. The Secure Channel pattern addresses the problem to ensure that data being passed across a public network is secure in transit. This textual description is formalized as a Prolog rule to enable an automated problem detection process. Even if the formalization has to be done manually, the resulting rule expressing the problem and context of the pattern can be reused for arbitrary topologies. The rule `insecure_public_communication(C1, C2)` queries a fact base for two components *C1* and *C2* that are connected by a relation R which has a key-value property *sensitivedata: true* attached. This relation R must be of type *connectsTo*. In addition, the two components C1 and C2 must have different locations, indicated by the key-

value property *location*, and no security mechanisms are used for the connection between the components. This rule formalizes the problem and context of the Secure Channel pattern and can be applied to facts representing the structure of a topology.

The procedure to detect problems in a restructured topology-based deployment model, for example in a TOSCA topology, starts with modeling the application's structure (step 1). The topology is then annotated with target labels that indicate the desired distribution of the components to different target locations. In this example, the *PHP-App* shall be hosted in an external environment and the *Java-App* shall be located internally. In the restructuring step (step 2), the topology is split and matched using the Split and Match method by Saatkamp et al. [27]. In a third step, the graph-based description of the application's structure is transformed into Prolog facts. This can be automated based on transformation rules, as descripted in [26]. For each element in the topology a fact is created. An extract of the facts representing the exemplary topology is shown in **Fig. 2**. By applying all formalized patterns to the topology facts, problems can be detected in the restructured topology-based deployment model. As a result, the detected problems, the affected components, and the pattern addressing this problem are returned. For the transformation of TOSCA topologies to Prolog facts and the problem detection in such topologies the *Topology ProDec*[4] tool can be used [26].

3 Application Scenarios Based on Cloud Integration Patterns

The described approach in Section 2 can be applied to several pattern languages. In [26] the approach has been applied to the Secure Channel pattern from the security patterns [28] and to the Application Component Proxy from the cloud computing patterns [1]. In this paper, two additional application scenarios are presented based on two cloud integration patterns [1]: *Message Mover* and *Integration Provider*. In the following, the two patterns are described in more detail and the Prolog rules for formalizing these patterns are presented. They are also part of the patterns listed in the *Topology ProDec* tool and can be used for an automated problem detection in TOSCA topologies.

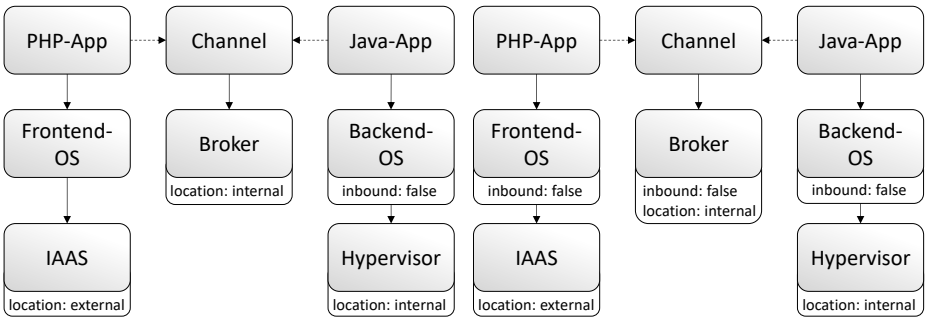

Fig. 3. Required Patterns: Message Mover (left) and Integration Provider (right)

[4] https://github.com/saatkamp/topology-prodec

3.1 Application Scenario: Message Mover

The Message Mover is a pattern of the cloud computing pattern language [1]. This pattern is just applicable to a message-based communication. Therefore, in **Fig. 3** on the left a topology with a message-based communication is depicted. In this example, the PHP-WebApp publishes data to a Queue and the Java-App receives data from it. After the redistribution the Java-App and the Broker are hosted in the internal datacenter and the PHP-WebApp is hosted in a public cloud, for example provided by Amazon. The Java-App is located in a restricted environment and thus, the access from outside the location is not permitted. To ensure the accessibility to a queue for each component, a queue shall be available in each location. The integration problem of these distributed queues can be solved by the *Message Mover*. However, the problem must be detected first. In the following the problem and context description of the Message Mover is presented [13]:

Problem:
How can message queues of different providers be integrated without an impact on the application component using them?

Context:
The application components comprising a distributed application (160) often exchange data using messaging. These messages are stored in message queues. [...] If these queues reside in different cloud environments that form a hybrid cloud (75) accessibility to queues of one environment may be restricted for application components that are deployed in another environment. [...] Therefore, each of the application components shall access a message queue hosted in the cloud environment where the application component itself is hosted. [...]

The pattern context is similar to the *Application Component Proxy*. The Application Component Proxy addresses the problem that a component directly accesses a component located in a restricted environment. Because direct access is not permitted to restricted environments an Application Component Proxy is required to access this component. More details on the formalization of this pattern can be found in [26]. However, in this case instead of a proxy an additional queue in the unrestricted environment and a message mover integrating the queues are required. The pattern aims to solve the accessibility of a more concrete system component, a queue. Based on the knowledge from messaging patterns, the problem description is based on the generic *Message Channel* in order to not exclude components, like e.g., topics which are publish-subscribe channels [17]. The resulting `distributed_messaging` rule for the formalized *Message Mover* is the following:

```
distributed_messaging(C1, C2):-
  messaging_communication(Channel, C1, C2),
  components_in_different_locations(C1, C2),
  hybrid_environment(C1, C2).
```

The shown facts serve as conditions for the pattern rule. Each of the facts in turn is a rule encapsulating a complex query (not shown for brevity, but can be found in the

Topology ProDec tool[5]). The first *condition* checks whether messaging is used in the topology because the pattern just relates to message-based systems. In addition, the problem and context description refers to distributed applications that are used in hybrid environments. Therefore, the second condition checks whether the communicating components are located in different environments and if they form a hybrid environment (third condition).

This rule can be applied to the deployment model depicted in **Fig. 3** on the left. Based on that, the problem is automatically detected and it can be solved according to the descripted solution for the *Message Mover* pattern.

3.2 Application Scenario: Integration Provider

The *Integration Provider* pattern is another cloud integration pattern from the cloud computing patterns [13]. In **Fig. 3** on the right a topology with two components (PHP-App and Java-App) communicate using messaging is shown. In contrast to the topology on the left, both components as well as the Channel are located in restricted environments. Thus, the components as well as the Channel are not accessible from outside the location. This problem can be solved by the Integration Provider pattern that describes the problem and its context as follows [13]:

Problem:
How can components in different environments be integrated through a 3rd-party provider?

Context:
When companies collaborate or one company has to integrate applications of different regional offices, different applications or the components of a distributed application are distributed among different hosting environments. Communication between these environments may be restricted. Especially, hosting environments may restrict any incoming communication initiated from the outside. Communication leaving the restricted environments is, however, often allowed. Therefore, additional integration components are required that have to be accessible from restricted environments. […]

From the above given description of this pattern, the difference to the `distributed_messaging` that formalizes the problem and context description of the Message Mover pattern can be seen: Instead of a hybrid environment consisting of a restricted and an unrestricted environment, these are two restricted environments that must be integrated to enable communication between the components. Besides that, the Integration Provider pattern is not limited to deployment models using messaging. In the following the `integration_of_restricted_environments` rule formalizing the problem and context of the Integration Provider pattern is shown:

[5] https://github.com/saatkamp/topology-prodec/blob/master/pattern_prologfiles/helper.pl

```
integration_of_restricted_environments(C1, C2):-
  components_in_different_locations(C1, C2),
  component_in_restricted_environment(C1),
  component_in_restricted_environment(C2),
  ((messaging_communication(Channel, C1, C2),
  component_in_restricted_environment(Channel));
  direct_communication(C1, C2)).
```

The problem only occurs in case the communicating components are located in different locations (first condition) and if these locations are restricted. The second condition is checked by the `component_in_restricted_environment` fact, which is in turn a rule encapsulating a complex query. Indicator for a restricted environment is the key-value property *inbound_communication: false*. To identify if an integration is required, the components located in different restricted environments have to communicate. For this, either a message-based or a direct communication must be part of the deployment model. In case of messaging the Integration Provider pattern only has to be applied in case the Channel, and thus the Message Broker, is also located in a restricted environment. Otherwise, an additional integration provider is not required.

The `integration_of_restricted_environments` rule can be applied to the topology presented in **Fig. 3** on the right. In this example, a problem is detected because the two communicating components (PHP-App, Java-App) as well as the used messaging system are located in different restricted environments. After detecting the problem, the solution described by the pattern can be applied.

The two problem and context formalized patterns, Message Mover and Integration Provider, show how the problem detection approach presented by Saatkamp et al. [26] can be used for detecting problems in topology models. Relevant patterns for problem recognition are not restricted to only one pattern language. They can be found in different pattern languages. In application scenarios presented in this work and in [26] patterns from the security [28] and the cloud computing [13] pattern language have been selected. Using the Topology ProDec tool the different patterns are validated based on TOSCA topologies modeled with the TOSCA modeling tool Winery[6] [21]. The application of the pattern formalization approach results in reusable rules that serve as conditions to express the actual pattern rules. For example, the rules `components_in_different_locations` and `component_in_restricted_environments` are used several times. Such reusable condition rules ease the formalization of further patterns.

4 Related Work

The underlying approach applied in this paper is presented by Saatkamp et al. [26]. We applied the approach to further cloud integration patterns and extracted reusable condition rules that ease the formalization of problem and context descriptions of further patterns. The formalization of patterns is already addressed by several other works

[6] https://github.com/eclipse/winery

[3,10,14,19,22]. However, in contrast to our work the solution provided by a pattern is formalized to identify the implemented patterns in a model instead of the problem solved by the pattern. In this paper, we present how the context and problem described by a specific pattern can be formalized to detect possibly occurring problems that can be solved by applying the respective pattern to the topology-based deployment models.

Kim and Khawand [20] presented an approach to formalize the problem domain of design patterns. They specify the problem domain as UML diagrams. Compared to logic programming, this approach has the disadvantage that the non-existence of elements cannot be specified. Furthermore, the context of the pattern is important to identify if a problem exists, this is not considered in this work. As a result of a formalized problem domain of patterns, patterns can be identified that solve the detected problems.

An approach presented by Haitzer and Zdun [16] is based on predefined architectural primitives that represent entities used in several patterns. They can be used to annotate software components to identify if a pattern is applicable. This semi-automated approach focuses on the applicability of patterns in software code. The applicability of patterns is also focused by the automated management approach presented by Breitenbücher [5] and Breitenbücher et al. [6,7,8]. Based on the cloud computing patterns [13] management idioms are defined that specify the transformation from a target topology fragment that represents the current state of an application by its components and their relations to a desired state that reflects the applied management pattern. These topology fragments are graphs that must be matched to a subgraph in the overall topology representing the current state of the application. Based on similar mechanisms, i.e., using graph matching, Arnold et al. [2] and Eilam et al. [11] presented concepts to facilitate the transformation from abstract topology-based deployment models to concrete configurations of the contained components. Also Guth and Leymann [15] use graph fragments for rewriting and refining architectural graphs. However, their approaches are based on subgraph isomorphism to identify the target fragment and thus, the non-existence of elements cannot be detected which is important to detect problems in topology-based deployment models, as shown in the formalization of the `Secure Channel` pattern in [26].

5 Conclusion

In this work we presented two application scenarios of the problem detection approach using formalized patterns by Saatkamp et al. [26]. We applied the approach to the Message Mover and the Integration Provider pattern. Both patterns are related to distributed applications and, thus, relevant for restructured topology-based deployment models. We have demonstrated that the approach is also applicable to message-based systems. Furthermore, reusable rules that serve as conditions to express the actual pattern rules are defined. In future work, we want to extend the pattern collection and want to improve the tool support to ease the authoring process for new rules.

This approach is not limited to the presented cloud computing patterns [13] or security patterns [28]. The approach could also be extended to other patterns, such as the

cloud data patterns [29] or the Internet of Things patterns [25]. Furthermore, the approach can also be used for a general validation of topology-based deployment models and is not limited to the usage in restructured deployment models. The extension to further domains will be investigated in future works.

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

References

1. Alexander, C., Ishikawa, S., Silverstein, M.: A Pattern Language: Towns, Buildings, Construction, Oxford University Press (1977).
2. Arnold, W., Eilam, T., Kalantar, M., Konstantinou, A.V., Totak, A.A.: Pattern Based SOA Deployment. In: Proceedings of the 5th International Conference on Service-Oriented Computing, pp. 1-12. Springer (2007).
3. Bergenti, F., Poggi, A.: Improving UML Designs Using Automatic Design Pattern Detection. Handbook of Software Engineering and Knowledge Engineering, 771-784 (2002).
4. Bergmayr, A., Breitenbücher, U., Ferry, N., Rossini, A., Solberg, A., Wimmer, M., Kappel, G., Leymann, F.: A Systematic Review of Cloud Modeling Languages. ACM Computing Surveys 51(1), Article 22, 38 pages (2018).
5. Breitenbücher, U.: Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements. Dissertation, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology (2016).
6. Breitenbücher, U., Binz, T., Kopp, O., Leymann, F.: Pattern-based Runtime Management of Composite Cloud Applications. In: Proceedings of the 3rd International Conference on Cloud Computing and Service Science, pp. 475-482. SciTePress (2013).
7. Breitenbücher, U., Binz T., Kopp, O., Leymann, F.: Automating Cloud Application Management Using Management Idioms. In: Proceedings of the 6th International Conference on Pervasive Patterns and Applications, pp. 60-69. Xpert Publishing Services (2014).
8. Breitenbücher, U., Binz, T., Kopp, O., Leymann, F., Wieland, M.: Context-Aware Cloud Application Management. In: Proceedings of the 4th International Conference on Cloud Computing and Services Science, pp. 499-509. SciTePress (2014).
9. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture, Volume 1 – A System of Patterns. Wiley (1996).
10. Di Martino, B., Esposito, A.: A rule-based procedure for automatic recognition of design patterns in uml diagrams. Software: Practice and Experience 46(7), 983-1007 (2016).
11. Eilam, T., Kalantar, M., Konstantinou, A., Pacifici, G., Pershing, J., Agrawal, A.: Managing the configuration complexity of distributed applications in Internet data centers. Communications Magazine 44(3), 166-177 (2006).
12. Endres, C., Breitenbücher, U., Falkenthal, M., Kopp, O., Leymann, L., Wettinger, J.: Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications. In Proceedings of the 9th International Conference on Pervasive Patterns and Applications, pp. 22-27. Xpert Publishing Services (2017).
13. Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: Cloud Computing Patterns – Fundamentals to Design, Build, and Manage Cloud Applications. Springer (2004).

14. Fontana, F.A., Zanoni, M.: A tool for design pattern detection and software architecture reconstruction. Information sciences 181(7), 1306-1324 (2011).
15. Guth, J., Leymann, F.: Towards Pattern-based Rewrite and Refinement of Application Architectures. In: Proceedings of the 12th Advanced Summer School on Service Oriented Computing. IBM Research Division (2018).
16. Haitzer, T., Zdun, U.: Semi-automatic architectural pattern identification and documentation using architectural primitives. Journal if Systems and Software 102, 35-57 (2015).
17. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Design, Building, and Deploying Messaging Solutions. Addison-Wesley Professional (2004).
18. Jamshidi, P. Pahl, C., Chinenyeze, S., Liu X.: Cloud Migration Patterns: A Multi-Cloud Service Architecture Perspective. In: Service-Oriented Computing – ICSOC 2014 Workshop, pp. 6-19. Springer (2014).
19. Kampffmeyer, H., Zschaler, S.: Finding the pattern you need: The design pattern intent ontology. In: International Conference on Model Driven Engineering Languages and Systems, pp. 211-225. Springer (2007).
20. Kim, D.K., Khawand, C.E.: An approach to precisely specifying the problem domain of design patterns. Journal of Visual Languages and Computing 18(6), 560-591 (2007).
21. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F.: Winery – A Modeling Tool for TOSCA-based Cloud Applications. In: Processing of the 11th International Conference on Service-Oriented Computing, pp. 700-704. Springer (2013).
22. Lim, D.K., Lu, L.: Inference of design pattern instances in uml models via logic programming. In: 11th IEEE International Conference on Engineering of Complex Computer Systems, pp. 10-29. IEEE (2006).
23. Meszaros, G., Doble, J.: MetaPatterns: A Pattern Language for Pattern Writing. In: Proceedings of International Conference on Pattern Languages of Program Design, pp. 164-200. ACM (1997).
24. OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0 (2013).
25. Reinfurt, L., Breitenbücher, U., Falkenthal, M., Leymann, F., Riegg, A.: Internet of Things Patterns. In: Proceedings of the 21th European Conference on Pattern Languages of Programs, Article Nr. 5. ACM (2016).
26. Saatkamp, K., Breitenbücher, U., Kopp, O., Leymann, F.: An Approach to Automatically Detect Problems in Restructured Deployment Models Based on Formalizing Architecture and Design Patterns. Computer Science – Research and Development (2018).
27. Saatkamp, K., Breitenbücher, U., Kopp, O., Leymann, F.: Topology Splitting and Matching for Multi-Cloud Deployments. In: Proceedings of the 7th International Conference on Cloud Computing and Services Science, pp. 247-258. ScitePress (2017).
28. Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., Sommerlad, P.: Security Patterns – Integration Security and System Engineering. John Wiley & Sons (2006).
29. Strauch, S., Andrikopolous, V., Breitenbücher, U. Sáez, S.G., Kopp, O., Leymann, F.: Using Patterns to Move the Application Data Layer to the Cloud. In: Proceedings of the 5th International Conference on Pervasive Patterns and Applications, pp. 26-33. Xpert Publishing Services (2013).
30. Wellhausen, T., Fiesser, A.: 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 (2012).