Institute of Architecture of Application Systems

Towards Pattern-based Rewrite and Refinement of Application Architectures

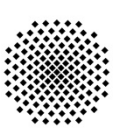
Jasmin Guth and Frank Leymann

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

BibTeX

```
@inproceedings{Guth2018_TowardsPatternBasedRewriteAndRefinement,
author      = {Guth, Jasmin and Leymann, Frank},
title       = {Towards Pattern-based Rewrite and Refinement of Application
               Architectures},
booktitle   = {Papers From the 12\textsuperscript{th} Advanced Summer School
               on Service-Oriented Computing (SummerSOC'18)},
year        = {2018},
pages       = {90--100},
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'18).
http://summersoc.eu



University of Stuttgart
Germany

Towards Pattern-based Rewrite and Refinement of Application Architectures

Jasmin Guth and Frank Leymann

Institute of Architecture of Application Systems, University of Stuttgart
Universitätsstr. 38, 70569 Stuttgart, Germany
[lastname]@iaas.uni-stuttgart.de

Abstract. With the ongoing growth of IT application systems, the development and modeling process of their architectures becomes increasingly complex. Architectural patterns capturing proven solutions for recurring problems in an abstract and human readable way should support this process. Due to the abstract character of patterns, they cannot be applied to a concrete architecture automatically: For each use case, patterns have to be read, understood, adapted to the respective use case, and realized manually. In this work, we tackle these issues by proposing an approach for an automated realization of architectural patterns within a given architectural graph based on graph transformation techniques.

Keywords: Application Architectures, Patterns, Solution Paths, Rewrite, Refinement, Graph Transformation.

1 Introduction

With the ongoing growth of IT application systems, the manual modeling and development phase of their architectures becomes increasingly complex. Diverse modeling tools are available to support this process, whereby each tool employs a different format and, hence, the graphical representation, if available, differs as well. This also affects the granularity of details depicted, like, e.g., abstract components are depicted or even implementation details are given. To enable a classification of architectures, Malan and Bredemeyer [1] delimit between three levels: (i) Conceptual architectures which are abstract, (ii) logical architectures which are detailed, and (iii) execution architectures which describe the process and deployment view. Further, each level can be subdivided into a behavioral and structural view. Since an application system architecture has significant impact on the prospective usability, performance, and maintainability, its development phase is of explicit importance [2]. To ease this process and to support the developer, diverse pattern languages can be used [3]. Patterns describe proven solutions for recurring problems documented in an abstract and human readable way [4]. Due to the abstract character of patterns, they cannot be applied to an architecture directly: Patterns have to be read, understood, adapted, and realized manually for each use case [4]. However, there are many pattern languages available, but a general approach to automatically apply patterns to architectures to refine and model the architecture by

selecting patterns is still missing. Hence, each pattern is realized over and over again manually, which leads to a multitude of possibly incorrect pattern realizations.

The result of the discussion above is that patterns are a profound support during the development and modeling phase of an architecture, missing a tool and method integration. The manual modeling and development of application system architectures is already a time consuming and error prone task. The lack of pattern integration leads to an amplification of those negative aspects and a multitude of possibly incorrect pattern realizations. Hence, the usage of patterns during the modeling phase, which should actually support and ease this phase, impedes it and makes it even more complex.

In this paper, we tackle these issues by introducing an approach to enable an automated pattern integration within the modeling and development process of the structural view of application system architectures on a conceptual level. In general, this leads to an adapted modeling process: (i) Abstract architectures can be refined by applying patterns, i.e., corresponding components and connectors are added which results in a more concrete architecture [5], and (ii) existing architectures can be rewritten based on applied patterns, i.e., components and connectors get exchanged or deleted [6].

Following, we will combine (i) the knowledge of proven solutions for recurring problems in terms of patterns, (ii) the modeling and development process of architectures, and (iii) architectural effects of patterns if applied onto architectures.

The remainder of this paper is structured as follows: Fundamentals to ease the understanding are presented in Section 2. We introduce the concept of our approach in Section 3 and work related to our approach is discussed in Section 4. We conclude this work and give a short overview of future work in Section 5.

2 Fundamentals

Within this section we give an overview of fundamentals to ease the understanding of our approach. First, background of application architectures and architectural graphs that we use to represent an application system architecture is given. Then fundamentals of patterns, pattern languages, and solution paths are presented.

2.1 Application System Architectures & Architectural Graphs

The structure of an IT system is described by its architecture, depicting the system's components and their relationships, as well as their external visible properties [7]. Thus, an IT architecture describes the composition of architectural elements without implementation details. For the documentation and visualization of an architecture, multiple architecture description languages, modeling languages, and modeling tools are present, such as ACME[1], ArchiMate[2], or particular UML[3] diagrams. Since each language and tool differs within their capabilities and, hence, their representation range, and due

[1] http://www.cs.cmu.edu/~acme/
[2] http://www3.opengroup.org/subjectareas/enterprise/archimate-overview
[3] http://www.uml.org

to the fact, that in most cases diverse people are involved, like, e.g., business process managers and programmers, which develop and discuss the architecture, the level of details within an architecture differs [1]. It has become common practice to use drawings of boxes to represent components and lines representing relationships [8]. This informal proceeding adapts Le Métayer [9] and describes a representation and formalization of software architectures as graphs, in which nodes represent components and edges represent relationships among them. Within this work, we use such a graph representation of architectures, referred to as an architectural graph to formalize an application system architecture. Those architectural graphs represent structural architectures on the conceptual level [1]. **Fig. 1** shows an exemplary architectural graph, whereby nodes represent the components of the architecture and edges represent the connectors among the components. Furthermore, each node is mapped to a type, such as application, server, or virtual machine. This mapping is required for a verification if the corresponding pattern is applicable to the architectural graph, since a specific pattern cannot be applied to all kind of architectural graphs due to possibly required components and relationships, a detailed description follows in Sect. 3.3.

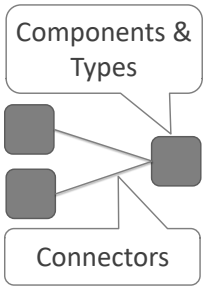


Fig. 1. Architectural Graphs

2.2 Patterns, Pattern Languages & Solution Paths

Patterns describe proven solutions for recurring problems within a certain context in an abstract and human readable way [4]. For example, a pattern may provide an abstract solution for common problems on how to design an application architecture. Patterns abstractly document an approach on how to solve a certain problem, since they are independent of the underlying technologies, such as specific runtime environments or programming languages [4]. Typically, concrete implementation realizations, such as code snippets, are not documented. Hence, patterns need to be adapted to the respective use case, and can, therefore, be used within various kinds of IT environments [4].

A pattern language comprises a collection of related patterns, forming a network as depicted in **Fig. 2**, in which one can navigate from one pattern to another related one that might become relevant after the application of the first pattern [3,10]. Several pattern languages are available in the field of IT, whereby each language has a different focus. For example, Gamma et al. [11] published design patterns focusing on object-oriented software, Fowler [12] introduced patterns for the development of enterprise application systems, and Fehling et al. [13] published patterns which focus on cloud computing and its architectures. Besides such general pattern languages, several patterns are available, published and realized by platform providers, such as the AWS Cloud Design Patterns [14]. Such provider-specific patterns are excluded from this work since they are bound to the provider and implement provider-specific realizations.

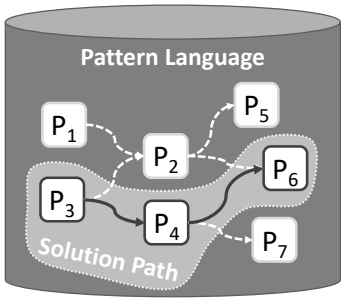


Fig. 2. Pattern Language and Exemplary Solution Path [15]

As described above, a pattern language forms a network of patterns, which connects related patterns [3,10]. After an entry point, i.e., a pattern which solves the problem at least partially, to this network is found and selected, subsequent and related patterns can be navigated to and selected as well [15]. Each such a possible path of selected patterns within this network forms a solution path [15,16], one exemplary solution path is shown in **Fig. 2**. Within our approach, we use solution paths to define the application order of patterns to architectural graphs, since solution paths are directed paths.

3 Concept of Pattern-based Rewrite & Refinement of Application Architectures

The aim of our approach is to enable a pattern-based rewrite and refinement of architectures through an automated application of patterns. Therefore, architectures are represented as architectural graphs as described in Sect. 2.1. To define the application order of patterns, we use the selected solution path, as described in Sect. 2.2. **Fig. 3** gives an overview of our approach. On the upper left side, the input is shown, i.e., the basis architectural graph and a selected solution path, which is described in Sect. 3.1. The remaining steps are iterative, since each pattern of the selected solution path is applied individually: First, the pattern which is applied next is defined, as described in Sect. 3.2.

Then the requirements of this pattern are checked, as explained in Sect. 3.3. In the last step, the architectural graph is rewritten or refined, as described in Sect. 3.4.

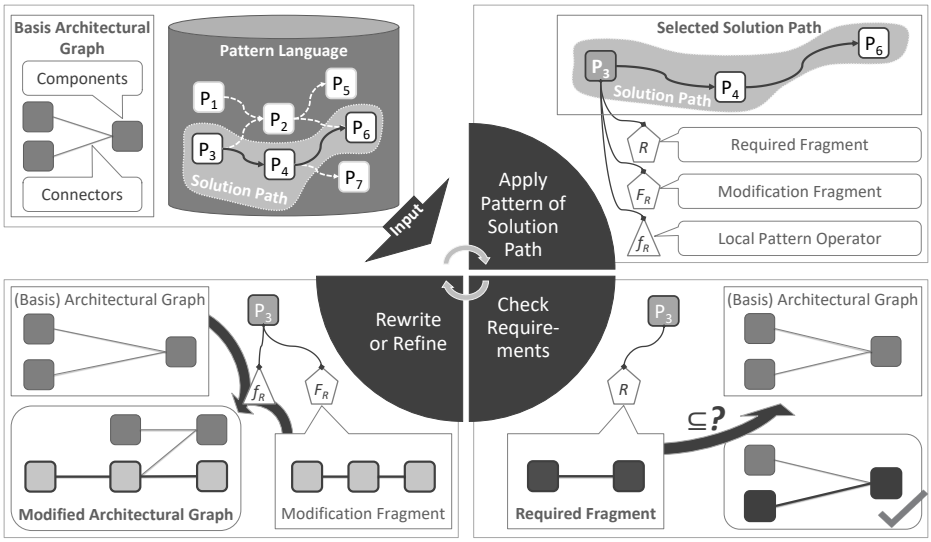


Fig. 3. Concept of Pattern-based Rewrite and Refinement of Application Architectures

3.1 Input

A basis architectural graph and a selected solution path of a specific pattern language forms the input of our iterative approach, as shown on the top left of **Fig. 3**. The basis architectural graph is a possibly abstract architecture represented as a graph, as described within Sect. 2.1, on which the modifications resulting of the application of a pattern are performed. For instance, a client server architecture could be such a basis architectural graph with three nodes, i.e., two nodes of type client and another node of type server, as well as two edges connecting both client components with the server component. Furthermore, the selected solution path, as described within Sect. 2.2, forms the second part of the input. The selected solution path comprises all patterns that have to be applied, and the order of the directed solution path defines the application order in which the patterns are applied to the basis architectural graph. For each pattern of the solution path all following described three steps are performed. The resulting architectural graph of one iteration serves as an input for the next iteration. Within the first iteration the basis architectural graph of the input is used to operate on.

3.2 Apply Pattern of Solution Path

Within the step *Apply Pattern of Solution Path*, as shown on the top right-hand side of **Fig. 3**, the next pattern of the solution path, as defined within the input, is taken to be applied to the (basis) architectural graph. The solution path defines the application order, i.e., starting with the first pattern P_3 of the solution path, within the next iteration

the second pattern P_4 gets applied, etc. To achieve a proper modification of the architectural graph, each pattern consists of three attachments: (i) R - The *required fragment*, which describes a possibly empty subgraph containing the required components and connectors of which the pattern cannot be applied without. If the pattern comprises no requirements the required fragment is an empty graph. (ii) F_R - The *modification fragment*, comprising a possibly empty graph, contains all components and connectors which have to be embedded in the architectural graph. Embedding a modification fragment covers adding and removing the fragment or parts of it, as well as replacing components and connectors of the architectural graph with components of the modification fragment or even replacing the whole graph. (iii) f_R - The *local pattern operator* defines the modification of the architectural graph, i.e., how the modification fragment is embedded, like e.g., the fragment is added and connected to a specific node.

3.3 Check Requirements

Within the step *Check Requirements*, as shown on the bottom right side of **Fig. 3** the above described requirements of the pattern to be applied get verified. Therefore, it is checked if the required fragment R is a subgraph of the (basis) architectural graph. For a positive verification, the (basis) architectural graph has to contain the required fragment R, i.e., R is a subgraph of the (basis) architectural graph. If the required fragment R is not contained within the (basis) architectural graph, i.e., the required fragment R is not a subgraph of the (basis) architectural graph, the pattern cannot be applied. This verification is done within each iteration, checking if the actual pattern of the solution path can be applied. Corresponding to the example within **Fig. 3**, for applying the pattern P_3 the (basis) architectural graph must contain a subgraph with two nodes of a specific type, as well as an edge connecting both components with each other. This procedure ensures that for the (basis) architectural graph only suitable patterns are applied and, thus, that only reasonable architectural graphs result of the next step.

3.4 Rewrite or Refine

The *Rewrite or Refine* step, shown on the bottom left side of **Fig. 3** modifies the architectural graph based on the modification fragment and the local pattern operator. As described above, the modification fragment F_R consists of all components and connectors to be embedded within the (basis) architectural graph and the local pattern operator f_R defines how the modification fragment gets embedded. This results in a modified architectural graph. As shown in **Fig. 3**, embedding the modification fragment F_R in the basis architectural graph means that the lower left node of the basis architectural graph is replaced by the middle node of the modification fragment and the remaining two component nodes and the corresponding connectors are added to the architectural graph. The modification may either result in a refined architectural graph, i.e., more details in terms of added components and connectors are depicted, or it results in a rewritten architectural graph, i.e., components and connectors are exchanged or deleted. The modified architectural graph is then used within the next iteration as the underlying architectural graph on which the next pattern of the solution path gets applied.

4 Related Work

In this section we delimit our approach against existing works that combine the knowledge of proven solutions for recurring problems, in terms of patterns, and the modeling and development process of architectures of application systems.

Eden et al. [17] introduce an approach for an automated application of design patterns. For this, programmers have to specify a pattern in an abstract way and the realization of the pattern in a specific program. Following, the pattern can be applied automatically, whereby the programmer may edit the implementation manually. Contrary to our work, patterns are used to add source code to a given program and not to model and define the architecture of an application system on a conceptual level.

Bergenti and Poggi [18] introduce the IDEA (Interactive DEsign Assistant) system to detect design patterns within UML class and collaboration diagrams. The system further enables to improve the detected pattern realizations within an UML diagram. This work focuses on the detection and improvement of design patterns and operates on a logical architecture, whereby the structural as well as the behavioral view is considered [1]. Contrary, our work operates on the conceptual level, focusing on the structural view and uses patterns to automatically model and define the architecture.

Bolusset and Oquendo [5] introduce a formal approach to refine software architectures based on transformation patterns using rewriting logic. Within their approach the refinement of an architecture does not change the architecture but specifies the components of an architecture in more detail, such as the definition of ports. Contrary, our approach results in a more detailed and possibly changed architecture. Furthermore, they use transformation patterns in terms of rewriting logic rules and equations and not in terms of best-practices and proven solutions for recurring problems.

Zdun and Avgeriou [19] present an approach to model architectural patterns through architectural primitives using UML profiles. This work focuses on modeling architectural patterns. Contrary, they do not use patterns to model and define an architecture.

Arnold et al. [20,21] introduce an approach for an automated realization of deployment patterns, which describe service deployment best-practices as model-based patterns capturing the structure of a solution without the binding to a specific resource instance. Therefore, deployment patterns have to be defined and modeled by experts so that deployer can use them. In contrast to our work, they do not use architectural patterns to model and define the architecture of an application system.

Zimmermann et al. [22] present an architectural design method based on the combination of pattern languages and reusable architectural design decision models. Contrary to our approach, within their work they use patterns in terms of architectural decisions.

Eilam et al. [23] present an approach for an automated transformation of deployment models onto workflow models. Within this work, transformation operations are represented as automation signatures including a model pattern representing the effects of operations on resources. Those automation signatures, i.e., model-based patterns are matched to a deployment desired state model. Hence, in this work patterns are used in terms of automation signatures and not as proven solutions for recurring problems.

Fehling et al. [24] present an approach to enrich application architecture diagrams by pattern annotations during the development phase. In contrast to our approach, those

pattern annotations express the requirements of application components and usage dependencies on each other and on the runtime. They do not use patterns to model or define the refinement of the architecture of an application system itself.

Breitenbücher [25] and Breitenbücher et al. [26,27] introduce an approach to automatically apply management patterns onto topologies to enable the management of composite cloud applications. Contrary to our approach, they do not focus on modeling an application architecture through a selection and application of patterns. Since they operate on topology graphs and use graph isomorphism and subgraph isomorphism methods to verify if a pattern can be applied to a topology graph, this can be used within our approach as a basis to check if a pattern is applicable to a given architectural graph. Furthermore, the algorithm used to transfer the topology, i.e., the application of the pattern, can be used as a basis for the modification of architectural graphs as well.

Jamshidi et al. [28] describe a set of patterns which document how to migrate an application to a cloud environment. Within each pattern the initial as well as resulting architecture of the application is described. Even though this can be used for an automated migration, each pattern is applied to an architecture manually.

Lytra et al. [29] introduce an approach and prototypical implementation for transformation actions and consistency checking rules to (semi-)automatically map architectural design decisions onto architectural component models. Furthermore, they introduce an architectural knowledge transformation language to define and realize the mentioned mapping. In contrast to our approach, this work does not use patterns to define an architecture, but the selection of architecture design decisions results in pattern implementations. Thus, pattern realizations are one result of their approach, but patterns are not used to model or define the refinement or rewrite of an architecture.

Hirmer and Mitschang [30] describe an approach to transform non-executable data mashup plans into an executable format by selecting an appropriate pattern and further parameters. This approach is based on rule-based transformations and focuses on data processing and integration scenarios. In contrast to our work, this approach uses predefined modularized implementation fragments which are selected and scripted together.

Amato and Moscato [31] present an approach for a manual formalization of patterns resulting in workflows and automatic verification of soundness. Contrary, the developer has to model or formalize the pattern by hand and cannot apply it automatically.

Lehrig [32] and Lehrig et al. [33] introduce the architectural template (AT) method, which enables design-time analyses of quality-of-service properties of software systems based on reusable modeling templates capturing architectural knowledge. Contrary, their approach operates on existing architectures aiming the analysis of its behavioral models, and, therefore architectural templates are embedded within the architecture automatically. Nevertheless, the embedding of architectural templates can serve as a basis for the refinement and rewrite of architectural graphs.

Saatkamp et al. [34,35] present an approach to automatically detect problems in restructured deployment models by formalizing the problem and context domain of architecture and design patterns. This approach can be adapted and integrated within our approach to verify if a pattern is applicable to an architectural graph. Nevertheless, they do not apply patterns to a deployment model or use patterns to model or define the rewrite or refinement of an architecture of an application system.

Falkenthal et al. [36,37] introduce concrete solutions of patterns capturing implementation realizations, such as code snippets. Based on the selected concrete solution path, they further present a method to aggregate multiple concrete solutions into an overall solution. Contrary to our work, Falkenthal et al. do not operate on the structural view of conceptual architectures. Nevertheless, a pattern realization within an architectural graph can be considered as a concrete solution, this indicates that the aggregation of concrete solutions is also possible within our approach. In future work we will investigate if their approach is applicable to the structural view of conceptual architectures and if an aggregated application of patterns is a promising approach for our work.

5 Conclusion & Future Work

Up to this point, patterns have to be read, understood, adapted, and implemented for each use case manually. This procedure has to be integrated into the modeling process of architectures of application systems. Our approach eases the development and modeling process of architectures by combining this process with the knowledge of proven solutions for recurring problems in terms of patterns.
Within this work we introduced our approach to enable a refinement and rewrite of architectures based on selected patterns. Therefore, architectures are depicted as architectural graphs, with nodes representing components and edges representing connectors among the components of an architecture of an application system. Following a selected solution path, patterns are applied to the architectural graph successively. To achieve a coherent resulting architectural graph, patterns are only applied if their requirements are fulfilled, i.e., if the required possibly empty subgraph is present within the underlying (basis) architectural graph. The modification of the architectural graph resulting of the application of a specific pattern is based on a modification fragment, which depicts a possibly empty graph to be embedded within the architectural graph, and on the local pattern operator, which defines how the modification fragment is embedded within the architectural graph. As a result, the architectural graph can be refined through the application of patterns, i.e., components and connectors are added, or the architectural graph can be rewritten, i.e., already present components and connectors are exchanged or even deleted. Hence, our approach enables a pattern-based refinement and rewrite of application architectures by using graph transformation techniques.

Within future work, we will formalize and further elaborate the presented approach. Furthermore, we will investigate, whether there are better ways to define the application order of patterns, for example, if an application order based on the required fragments or modification fragments of all patterns of the selected solution path are more effective. Additionally, we will consider the approach of Falkenthal et al. [36,37] to investigate if an application of the aggregated patterns of the selected solution path is effective.

Acknowledgements. This work was partially funded by the German Research Foundation (DFG) project ADDCompliance (636503).

References

1. Malan, R., Bredemeyer, D.: Software architecture: Central concerns, key decisions. Software Architecture Action Guide. Architecture Resources Pubs., Bredemeyer Consulting (2002).
2. Kaartinen, J., Palviainen, J., Koskimies, K.: A pattern-driven process model for quality-centered software architecture design - A case study on usability-centered design. In: Proceedings of the Australian Software Engineering Conference, pp. 17–26. IEEE (2007).
3. Alexander, C., Ishikawa, S., Silverstein, M.: A Pattern Language: Towns, Buildings, Construction. Oxford University Press (1977).
4. Alexander, C.: The Timeless Way of Building. Oxford University Press (1979).
5. Bolusset, T., Oquendo, F.: Formal Refinement of Software Architectures Based on Rewriting Logic. Proceedings of the International Workshop on Refinement of Critical Systems: Methods, Tools and Experience 29(5), 1-20 (2002).
6. Meseguer, J.: Research Directions in Rewriting Logic. In: Computational Logic. Springer (1999).
7. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley (2003).
8. Allen, R., Garlan, D.: Formalizing architectural connection. In: Proceedings of the 16th International Conference on Software Engineering, pp. 71–80. IEEE (1994).
9. Le Métayer, D.: Describing software architecture styles using graph grammars. IEEE Transactions on Software Engineering 24, 521–533 (1998).
10. Fehling, C., Barzen, J., Falkenthal, M., Leymann, F.: PatternPedia – Collaborative Pattern Identification and Authoring. In: Proceedings of Pursuit of Pattern Languages for Societal Change. The Workshop 2014, pp. 252–284 (2015).
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley (1994).
12. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley (2002).
13. Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications. Springer (2014).
14. Amazon Web Services LLC: AWS Cloud Design Pattern. http://en.clouddesignpattern.org/index.php/Main_Page, last accessed 2018/06/22.
15. Falkenthal, M., Barzen, J., Breitenbücher, U., Fehling, C., Leymann, F., Hadjakos, A., Hentschel, F., Schulze, H.: Leveraging Pattern Applications via Pattern Refinement. In: Proceedings of the International Conference on Pursuit of Pattern Languages for Societal Change, pp. 38–61. epubli (2016).
16. Zdun, U.: Systematic Pattern Selection Using Pattern Language Grammars and Design Space Analysis. In: Software: Practice & Experience 37, 983–1016 (2007).
17. Eden, A. H., Yehudai, A., Gil, J.: Precise Specification and Automatic Application of Design Patterns. In: Proceedings of the 12th IEEE International Conference Automated Software Engineering, pp. 143–152. IEEE (1997).
18. Bergenti, F., Poggi, A.: Improving UML Designs Using Automatic Design Pattern Detection. Handbook of Software Engineering and Knowledge Engineering: Volume II: Emerging Technologies, pp. 771-784 (2002).
19. Zdun, U., Avgeriou, P.: Modeling Architectural Patterns Using Architectural Primitives. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, pp. 133–146. ACM (2005).
20. Arnold, W., Eilam, T., Kalantar, M., Konstantinou, A.V., Totok, A.A.: Pattern Based SOA Deployment. Proceedings of the 5th International Conference on Service-Oriented Computing, pp. 1–12. Springer (2007).

21. Arnold, W., Eilam, T., Kalantar, M., Konstantinou, A.V., Totok, A.A.: Automatic Realization of SOA Deployment Patterns in Distributed Environments. Proceedings of the 6th International Conference on Service-Oriented Computing, pp. 162–179. Springer (2008).
22. Zimmermann, O., Zdun, U., Gschwind, T., Leymann, F.: Combining Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and Comprehensible Design Method. In: 7th Working IEEE/IFIP Conference on Software Architecture, pp. 157–166. IEEE (2008).
23. Eilam, T., Elder, M., Konstantinou, A.V, Snible, E.: Pattern-based Composite Application Deployment. In: Proceedings of the 12th IFIP/IEEE International Symposium on Integrated Network Management, pp. 217–224. IEEE (2011).
24. Fehling, C., Leymann, F., Rütschlin, J., Schumm, D.: Pattern-Based Development and Management of Cloud Applications. In: Future Internet 4, 110–141 (2012).
25. Breitenbücher, U.: Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements. Dissertation, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology (2016).
26. 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 Services Science, pp. 475–482. SciTePress (2013).
27. Breitenbücher, U., Binz, T., Kopp, O., Leymann, F.: Automating Cloud Application Management Using Management Idioms. In: Proceedings of the 6th International Conferences on Pervasive Patterns and Applications, pp. 60–69. Xpert Publishing Services (2014).
28. 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).
29. Lytra, I., Tran, H., Zdun, U.: Harmonizing architectural decisions with component view models using reusable architectural knowledge transformations and constraints. In: Future Generation Computer Systems 47, 80–96 (2015).
30. Hirmer, P., Mitschang, B.: FlexMesh - Flexible Data Mashups Based on Pattern-Based Model Transformation. In: Rapid Mashup Development Tools, pp. 12–30. Springer (2016).
31. Amato, F., Moscato, F.: Pattern-based orchestration and automatic verification of composite cloud services. In: Computers and Electrical Engineering 56, 842–853. Elsevier Ltd (2016).
32. Lehrig, S. M.: Efficiently Conducting Quality-of-Service Analyses by Templating Architectural Knowledge. Dissertation, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology (2018).
33. Lehrig, S., Hilbrich, M., Becker, S.: The architectural template method: templating architectural knowledge to efficiently conduct quality-of-service analyses. In: Software: Practice and Experience 48, 268–299 (2018).
34. 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).
35. Saatkamp, K., Breitenbücher, U., Kopp, O., Leymann, F.: 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 (2018).
36. Falkenthal, M., Barzen, J., Breitenbücher, U., Fehling, C., Leymann, F.: Efficient Pattern Application: Validating the Concept of Solution Implementations in Different Domains. In: International Journal On Advances in Software 7, 710–726 (2014).
37. Falkenthal, M., Barzen, J., Breitenbücher, U., Leymann, F.: On the Algebraic Properties of Concrete Solution Aggregation. In: Computer Science - Research and Development (2018).