From Pattern Languages to Solution Implementations

Michael Falkenthal, Johanna Barzen, Uwe Breitenbücher, Christoph Fehling, Frank Leymann
Institute of Architecture of Application Systems
University of Stuttgart
Stuttgart, Germany
{falkenthal, barzen, breitenbuecher, fehling, leymann}@iaas.uni-stuttgart.de

***Abstract*—Patterns are a well-known and often used concept in the domain of computer science. They document proven solutions to recurring problems in a specific context and in a generic way. So patterns are applicable in a multiplicity of specific use cases. However, since the concept of patterns aims at generalization and abstraction of solution knowledge, it is difficult to apply solutions provided by patterns to specific use cases, as the required knowledge about refinement and the manual effort that has to be spent is immense. Therefore, we introduce the concept of Solution Implementations, which are directly associated to patterns to efficiently support elaboration of concrete pattern implementations. We show how Solution Implementations can be aggregated to solve problems that require the application of multiple patterns at once. We validate the presented approach in the domain of cloud application architecture and cloud application management and show the feasibility of our approach with a prototype.**

Keywords-pattern; pattern languages; pattern-based solution; pattern application; cloud computing patterns

I. INTRODUCTION

Pattern and pattern languages are a well-established concept in different application areas in computer science and information technology (IT). Originally introduced to the domain of architecture [2], the concept of patterns recently got more and more popular in different domains such as education [18], design engineering [16], cloud application architecture [23] or costumes [17]. Patterns are used to document proven solutions to recurring problems in a specific context. However, since the concept of patterns aims at generalization and abstraction, it is often difficult to apply the captured abstracted knowledge to a concrete problem. This can require immense manual effort and domain-specific knowledge to refine the abstract, conceptual, and high-level solution description of a pattern to an individual use case. These following examples show that this problem occurs in several domains due to the abstraction of solution knowledge into patterns. For example, if a PHP: Hypertext Preprocessor (PHP) developer uses the Gang of Four patterns of Gamma et al. [19], he is faced with the problem that he has to translate the general solution concepts of the patterns to his concrete context, i.e., he has to implement solutions based on a given programming paradigm predefined by PHP. An enterprise architect who has to integrate complex legacy systems may use the enterprise application architecture patterns of Fowler [20] or the enterprise integration patterns of Hohpe and Wolf [12] to gain insight to proven solutions of his problems; but, these are still generic solutions and he has to find proper implementations for the systems to integrate. This can lead to huge efforts since besides paradigms of used programming languages he has also to consider many constraints given by the running systems and technologies. A teacher who uses the learning patterns of Iba and Miyake [18] has to adapt them to match his prevailing school system with all the teaching methods. To give a final example, a costume designer could use the patterns of Schumm et al. [17] to find clothing conventions for a cowboy in a western film but he still has to come up with a specific solution for his current film.

While patterns in general describe proven generic solutions at a conceptual level, the examples above show that it is still time consuming to work out concrete solutions of those generic solutions.

To overcome this problem, we suggest that patterns should be linked to the (i) original concrete solutions from which they have been deduced (if available) and (ii) to individual new concrete implementations of the abstractly described solution. This enables users that want to apply a certain pattern to take already existing implementations for their use cases, which eases applying patterns and reduces the required manual effort significantly.

The remainder of this paper is structured as follows: we clarify the difference between the common concept of pattern solutions and concrete solutions in Section II. In Section III, we discuss related work and the lack of directly usable concrete solutions in state of the art pattern research. We show how to keep patterns linked to concrete solution knowledge and how to select them to establish concrete solution building blocks, which can be aggregated in Section IV. In Section V, we give an example of how to apply the introduced concepts in the domains of cloud application architecture and cloud application management and verify the feasibility of the presented approach by means of an implemented prototype in Section VI. We conclude this paper with an outline of future work in Section VII.

II. MOTIVATION

Patterns document proven solution knowledge mainly in natural text to support human readers of a pattern. Patterns are often organized into pattern languages, i.e., they may be connected to each other. Pattern languages provide a common *template* for documenting all contained patterns.

 ISBN: 978-1-61208-343-8

This template typically defines different items to be documented such as “Problem”, “Context”, “Solution”, and “Known Uses”. The problem and context section describe the problem to be solved in an abstract manner where the solution describes the general characteristics of the solution – all only conceptually, in an abstract way. Thus, the general solution is refined for individual problem manifestations and use cases resulting in different concrete solutions every time the pattern is applied. The known uses section is the only place where concrete solutions from which the pattern has been abstracted are described. But these are commonly not extended as the pattern is applied nor do they guide pattern readers during the creation of their own solutions.

Therefore, due to the abstract nature of patterns and generalized issues, most pattern languages only contain some concrete solutions a pattern was derived from in the known uses section. This leads to the problem that the user of the pattern has to design and implement a specific solution based on his individual and concrete use case, i.e., a solution has to be implemented based on the user’s circumstances considering the given pattern. However, many patterns are applied several times to similar use cases. Thus, the effort has to be spent every time for tasks that were already executed multiple times. For example, the Model-View-Controller (MVC) Design Pattern is an often used pattern in the domain of software design. This pattern was, therefore, implemented for many applications in many programming languages from scratch, as patterns typically provide no directly usable concrete solutions for use cases in a concrete context. Patterns are not linked with a growing list of solutions that can be used as basis to apply them to individual use cases rapidly: each time a pattern should be applied, it has to be refined manually to the current use case. The provided sections such as “Known Uses” and “Examples”, which are part of the pattern structure in most pattern languages, therefore, support the reader in creating new solutions only partially [10][12][13]: they provide only partial solution refinements or solution templates as written text but not directly applicable implementations that can be used without additional effort. The major reasons for this problem are, that neither the concrete solution is documented in a way that enables reusing it efficiently nor it is obvious how to aggregate existing solutions if multiple patterns are applied together. Thus, the reader of a pattern is faced with the problem of creation and design to elaborate a proper solution based on a given pattern each time when it has to be applied – which results in time-consuming efforts that decrease the efficiency of using patterns.

As of today, patterns are typically created by small groups of experts. By abstracting the problems and solutions into patterns relying on their expertise, these experts determine the content of the patterns. This traditional way of pattern identification created the two issues already seen: first, the patterns are not verifiable because the concrete solutions they have been abstracted from are not traceable (“pattern provenance”) and second the patterns document abstracted knowledge, therefore manual effort and specific knowledge is needed to apply them to concrete problems.

Another problem occurs if multiple patterns have to be combined to create a concrete solution. Pattern languages tackle the problem of aggregating patterns to solve overall problems. As shown by Zdun [9], this can be supported by defining relationships between patterns within a pattern language, which assure that connected patterns match together semantically, i.e., that they are composable regarding their solutions. This means that patterns can be used as composable building blocks to create overall solutions. Once patterns are composed to create overall solutions the problem arises that concrete solutions have to be feasible in the context of concrete problem situations. Referring to the former mentioned example of a PHP developer, the overall concrete solution, consisting of the concrete solutions of the composed patterns, has to be elaborated that it complies with the constraints defined by the programming language PHP. So, the complexity of creating concrete solutions from composed patterns increases with the number of aggregated solutions, since integration efforts add to the efforts of elaborating each individual solution. Thus, to summarize the discussion above, we need a means to improve the required refinement from a pattern’s abstract solution description to directly applicable concrete solutions and their composition.

III. RELATED WORK

Patterns are human readable artifacts, which combine problem knowledge with generic solution knowledge. The template documenting a pattern contains solution sections presenting solution knowledge as ordinary text [2][19][13]. This kind of solution representation contains the general principle and core of a solution in an abstract way. Common solution sections of patterns do not reflect concrete solution instances of the pattern. They just act like manuals to support a reader at implementing a solution proper for his issues.

Iterative pattern formulation approaches as shown by Reiners et al. [6] and Falkenthal et al. [5] can enable that concrete solution knowledge is used to formulate patterns. Patterns are not just final artifacts but are formulated based on initial ideas in an iterative process to finally reach the status of a pattern. Nevertheless, in these approaches concrete solution knowledge only supports the formulation process of patterns but is not stored explicitly to get reused when a pattern is applied.

Porter et al. [15] have shown that selecting patterns from a pattern language is a question of temporal ordering of the selected patterns. They show that combinations and aggregations of patterns rely on the order in which the patterns have to be applied. This leads to so called pattern sequences which are partially ordered sets of patterns reflecting the temporal order of pattern application. This approach focuses on combinability of patterns, but not on the combinability of concrete solutions.

Many pattern collections and pattern languages are stored in digital pattern repositories such as presented by

 ISBN: 978-1-61208-343-8

Fehling [4], van Heesch [7] and Reiners [3]. Although these repositories support readers in navigating through the patterns they do not link concrete solutions to the patterns. Therefore, readers have to manually recreate concrete solutions each time when they want to apply a pattern.

Zdun [9] shows that pattern languages can be represented as graphs with weighted edges. Patterns are the nodes of the graph and edges are relationships between the patterns. The weights of the edges represent the semantics of the relationships as well as the effects of a pattern on the resulting context of a pattern. These effects are called goals and reflect the influence of a pattern on the quality attributes of software architectures. While this approach helps to select proper pattern sequences from a pattern language it does not enable to find concrete solutions and connect them together.

Demirköprü [8] shows that Hoare logic can be applied to patterns and pattern languages such that patterns are getting enriched by preconditions and postconditions. By considering this conditions, pattern sequences can be connected into aggregates respectively compositions of patterns where preconditions of the first pattern of the sequence are the preconditions of the aggregate and postconditions of the last pattern in the sequence are accordingly the postconditions of the aggregate. This approach also only tackles aggregation of patterns without considering concrete solutions.

Fehling et al. [31][33] show that their structure of cloud computing patterns can be extended to annotate patterns with additional implementation artifacts. Those artifacts can represent instantiations of a pattern on a concrete cloud platform. Considering those annotations, developers can be guided through configurations of runtime environments. Although patterns can be annotated with concrete implementation artifacts, this approach is only described in the domain of cloud computing and does not introduce a means to ease pattern usage and refinement in general.

IV. SOLUTION IMPLEMENTATIONS: BUILDING BLOCKS FOR APPLYING AND AGGREGATING CONCRETE SOLUTIONS FROM PATTERNS

In the section above, we summarized the state of the art and identify that (i) concrete solutions are not connected to patterns and that (ii) there are no approaches dealing with the aggregation of concrete solutions if multiple patterns have to be applied together. Even though there are approaches to derive patterns from concrete solution knowledge iteratively [5][6], concrete solutions are not stored altogether with the actual patterns nor are they linked to them. Concrete solutions, thus, cannot be retrieved from patterns without the need to work them out manually over and over again for the same kind of use cases. Therefore, we propose an approach that (i) defines concrete, implemented solution knowledge as reusable building blocks, (ii) that links these concrete solutions to patterns, and (iii) enables the composition of concrete solutions.

A. Solution Implementations

We argue that concrete solutions are lost during the pattern writing process since patterns capture general core solution principles in a technology and implementation agnostic way. In addition, applications of patterns to form new concrete solutions are not documented in a way that enables reusing the knowledge of refinement. As a result, the details of the concrete solutions are abstracted away and must be worked out again when a pattern has to be applied to similar use cases. Thus, the benefits of patterns in the form of abstractions lead to effort when using them due to the missing information of concrete realizations. We suggest keeping concrete solutions linked to patterns in order to ease pattern application and enable implementing new concrete solutions for similar use cases based on existing, already refined, knowledge. These linked solutions can be, for example, (i) the concrete solutions which were considered initially to abstract the knowledge into a pattern, (ii) later applications of the pattern to build new concrete solutions, or (iii) concrete solutions that were explicitly developed to ease applying the pattern.

Concrete solutions, which we call *Solution Implementations (SI)*, are building blocks of concrete solution knowledge. Therefore, Solution Implementations describe concrete solution knowledge that can be reused directly. For example, in the domain of software development, Solution Implementations provide code, which can be used directly in the development of an own application. For example, a PHP developer faced with the problem to implement the Gang of Four Pattern MVC [19] in an application can reuse a Solution Implementation of the MVC pattern written in PHP code. Especially, patterns may provide multiple different Solution Implementations – each optimized for a special context and requirements. So, there could be a specific MVC Solution Implementation for PHP4 and another for PHP5, each one considering the programming concepts of the specific PHP version. Another Solution Implementation could provide a concrete solution of the MVC pattern implemented in Java. So, in this case also a Java developer could reuse a concrete MVC solution to save implementation efforts.

By connecting Solution Implementations to patterns, users do not have to redesign and recreate each solution every time a pattern is applied. The introduced Solution Implementations provide a powerful means to capture existing fine-grained knowledge linked to the abstract knowledge provided by patterns. So, users can look at the connected Solution Implementations once a pattern is selected and reuse them directly. To distinguish between pattern's abstract solutions and Solution Implementations, we point out that the solution section of patterns describes the core solution principles in text format and the Solution Implementations represent the real solution objects – which may be in different formats (often depending on the problem domain), e.g., executable code in software development or real clothes in the domain of costumes. Thus, while patterns

 ISBN: 978-1-61208-343-8

are documented commonly in natural text, their Solution Implementations depend mainly on the domain of the pattern language and can occur in various forms. Since many specific Solution Implementations can be linked to a pattern, we need a means to select proper Solution Implementations of the pattern to be applied.

B. Selection of Solution Implementations from Patterns

Once a user selects a pattern, he is faced with the problem to decide which Solution Implementation solves his problem in his context properly. To enable selecting proper Solution Implementations of a pattern we introduce *Selection Criteria (sc)*, which determine when to use a certain Solution Implementation. The concept of keeping Solution Implementations linked to the corresponding pattern and supporting the selection of a proper Solution Implementation is shown in Figure 1. Selection Criteria are added to relations between Solution Implementations and patterns. Selection Criteria may be human readable or software interpretable descriptions of when to select a Solution Implementation. They provide a means to guide the selection using additional meta-information not present in the Solution Implementation itself.

To exemplify the concept, we give an example of Solution Implementations from the domain of architecture. In this domain addressed by Christopher Alexander [1][2], a Solution Implementation would be, e.g., a real entrance of a building or a specific room layout of a real floor, which are described in detail and linked to the corresponding pattern [1][2]. To find the most appropriate Solution Implementation for a particular use case, Selection Criteria such as the cost of the architectural Solution Implementation or the choice of used material can be considered. For example, two Solution Implementations for the pattern mentioned above that deals with room layouts might differ in the historical style they are built or by the functional purpose like living, industrial or office, etc. Thus, based on such criteria, the refinement of a pattern's abstract solution can be configured by specifying desired requirements and constraints.

To summarize the concept of Solution Implementations it has to be pointed out that solutions in the domain of patterns are abstract descriptions that are agnostic to concrete implementations and written in ordinary text to support readers. In contrast to this abstract description, we grasp Solution Implementations as fine-grained artifacts, which provide concrete implementation information for particular use cases of a pattern. Solution Implementations are linked to patterns where Selection Criteria are added to the relation between the pattern and the Solution Implementation to guide pattern users during the selection of Solution Implementations.

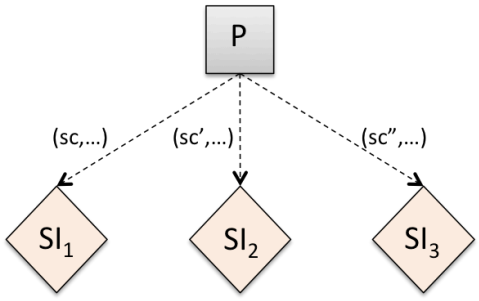

Figure 1. Solution Implementations (SI) connected to a pattern (P) are selectable under consideration of defined Selection Criteria (sc).

C. Aggregation of Solution Implementations

The concepts of Solution Implementations and Selection Criteria enable to reuse concrete solutions, which are linked to patterns. But most often problems have to be solved by combining multiple patterns. Therefore, we also need a means to combine Solution Implementations of patterns to solve an overall problem altogether. For this purpose, Solution Implementations connected to patterns can have additional interrelations with other Solution Implementations of other patterns affecting their composability. For example, Solution Implementations in the domain of software development are possibly implemented in different programming languages. Therefore, there may exist various Solution Implementations for one pattern in different programming languages, remembering the above example of the PHP and Java Solution Implementations of the MVC pattern. To be combined, both Solution Implementations often have to be implemented in the same programming language.

This leads to the research question "How to compose Solution Implementations selected from multiple patterns into a composed Solution Implementation?"

Patterns are often stored and organized in digital pattern repositories. These repositories, such as presented by Fehling [4], van Heesch [7] and Reiners [3], support users in searching for relevant patterns and navigating through the whole collection of patterns, respectively a pattern language formed by the relations between patterns. To support navigation through pattern languages, these relations can be formulated at the level of patterns indicating that some patterns can be "combined" into working composite solutions, some patterns are "alternatives", some patterns can only be "applied in the context of" other patterns etc. Zdun [9] has shown that pattern languages can be formalized to enable automated navigation through pattern languages based upon semantic and quality goal constraints reflecting a pattern's effect once it is applied. This also enables combining multiple patterns based on the defined semantics. The approach supports the reader of a pattern language to select proper pattern sequences for solving complex problems that require the application of multiple patterns at once. But, once there are Solution Implementations linked to patterns this leads to the requirement to not only compose patterns but also their concrete Solution Implementations into overall solutions.

 ISBN: 978-1-61208-343-8

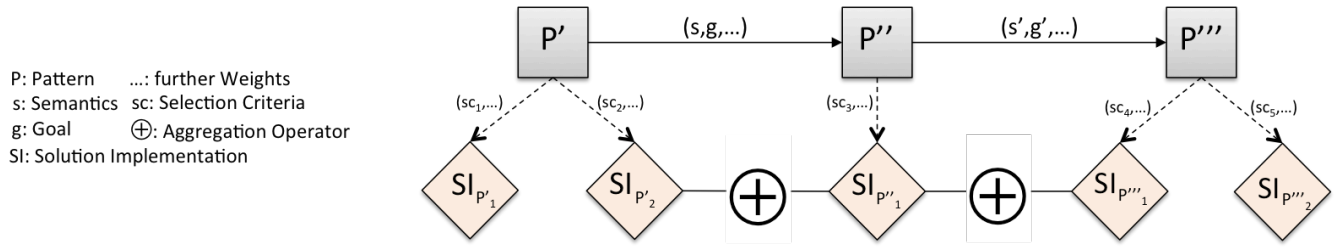

Figure 2. Aggregating Solution Implementations (SI) along the sequence of selected patterns (P).

We extend the approach of Zdun to solve the problem of selecting appropriate patterns to also select and aggregate appropriate Solution Implementations along the selected sequence of patterns.

To assure that Solution Implementations are building blocks composable with each other, we introduce the concept of an *Aggregation Operator,* as depicted in Figure 2. The Aggregation Operator is the connector between several Solution Implementations. Solution Implementations can just be aggregated if a proper Aggregation Operator implements the necessary adaptations to get two Solution Implementations to work together. Adaptions may be necessary to assure that Solution Implementations match together based on their preconditions and postconditions. Preconditions and postconditions are functional and technical dependencies, which have to be fulfilled for Solution Implementations. In Figure 2., the three patterns P′, P′′and P′′′ show a sequence of patterns, which can be selected through the approach of Zdun considering semantics (s) of the relations, goals (g) of the patterns and further weights. Solution Implementations are linked with the patterns and can be selected according to the Selection Criteria introduced in the section above. Furthermore, there are two Solution Implementations associated with pattern P′ but only Solution Implementation $SI_{P'_2}$ can be aggregated with Solution Implementation $SI_{P''_1}$ of the succeeding pattern P′′ due to the Aggregation Operator between those two Solution Implementations. There is no Aggregation Operator implemented for $SI_{P'_1}$, so that it cannot be aggregated with $SI_{P''_1}$, but, nevertheless, it is a working concrete solution of P′. So, in the scenario depicted in Figure 2 an Aggregation Operator has to be available to aggregate $SI_{P'_1}$ and $SI_{P''_1}$.

In general, Aggregation Operators have to be available to compose Solution Implementations for complex problems requiring the application of multiple patterns. Solution Implementations aggregated with such an operator are concrete implementations of the aggregation of the selected patterns. Aggregated Solution Implementations are, therefore, concrete building blocks solving problems addressed by a pattern language.

Aggregation Operators depend on the connected Solution Implementations, i.e., they are context-dependent due to the context of the Solution Implementations. In contrast to the context section of a pattern, which is used together with the problem section to describe the circumstances when a pattern can be applied, the Solution Implementations' context is more specific in terms of the concrete solution. For example, if an Aggregation Operator shall connect two Solution Implementations consisting of concrete PHP code, the operator itself could also be concrete PHP code wrapping functionality from both Solution Implementations. If the Solution Implementations to aggregate are Java class files, e.g., an Aggregation Operator could resolve their dependencies on other class files or libraries and load all dependencies. Afterwards it could configure the components to properly work together and execute them in a Java runtime. Thus, an Aggregation Operator composes and adapts multiple Solution Implementations considering their contexts. Another example on how the Aggregation Operators can be used in very different domains is an example of the domain of costumes in films. When dressing the characters of a western movie usually the sheriff costume pattern and the outlaw costume pattern need to be applied. But there are numerous Solution Implementations of these patterns in terms of concrete sheriff and outlaw costumes, e.g., for different historical time periods. To make sure the costumes of the sheriff and outlaw match together, an Aggregation Operator, for example, can ensure that certain Solution Implementations originate from the same time period or the same country and can be used together in one movie. Further the Aggregation Operator adapts Solution Implementations to suit to the settings of a scene in a film, i.e., by adapting the color of the costumes. Thus, the costumes' Solution Implementations are aggregated to solve a problem in combination. Those examples show that Solution Implementations of patterns from different domains have to be aggregated using specific Aggregation Operators. Since different pattern languages deal with different contexts, they can contain different Aggregation Operators to compose Solution Implementations.

V. VALIDATION

To validate the proposed concept of Solution Implementations, this section explains the application of

Solution Implementations in the domains of cloud application architecture and cloud management.

A. Deriving Solution Implementations in the Domain of Cloud Application Architecture

To explain the concept of Solution Implementations in the domain of cloud computing patterns, the example depicted in Figure 3 shows the three patterns *stateless component*, *stateful component*, and *elastic load balancer* from the pattern language and pattern catalogue of Fehling et al. [10][31]. The stateless component and stateful component patterns describe how an application component can handle state information. They both differentiate between session state – the state with the user interaction within the application and application state – the data handled by the application, for example, customer addresses etc. While the stateful component pattern describes how this state can be handled by the component itself and possibly be replicated among multiple component instances, the stateless component pattern describes how state information is kept externally of the component implementation to be provided with each user request or to be handled in other data storage offerings. The elastic load balancer pattern describes how application components can be scaled out: their performance is increased or decreased through addition or removal of component instances, respectively. Decisions on how many component instances are required are made by monitoring the amount of synchronous requests to the managed application components. The elastic load balancer pattern is related to both of the other depicted patterns as it conceptually describes how to scale out stateful components and stateless components: while stateless components can be added and removed rather easily, internal state may have to be extracted from stateful components upon removal or synchronized with new instances upon addition.

As depicted in Figure 3, the stateless component and stateful component pattern both provide Solution Implementations, which implement these patterns for Java web applications packaged in the web archive (WAR) format that are hosted on Amazon Elastic Beanstalk [21] which is part of Amazon Web Services (AWS) [30]. The elastic load balancer has three Solution Implementations implementing the described management functionality for stateful components and stateless components for WAR-based applications on Amazon Elastic Beanstalk and Microsoft Azure [22]. The Selection Criteria "WAR is deployed on Microsoft Azure" respectively "WAR is deployed on Elastic Beanstalk" support the user to choose the proper Solution Implementation. For example, if SI_2 is selected the user knows that this results in a concrete load balancer in the form of a deployed WAR file on Elastic Beanstalk. Since a load balancer scales components, it needs concrete instances of either stateless component or stateful component to work with. Thus, the user can select a proper Solution Implementation for the components based on his concrete requirements considering the Selection Criteria of the relations between the patterns stateless component and stateful component and their Solution Implementations. To assure that Solution Implementations are composable, i.e., that they properly work together, they refine and enrich the pattern relationships to formulate preconditions respectively postconditions on the Solution Implementation layer. The preconditions and postconditions of the elastic load balancer Solution Implementations, therefore, capture which related pattern – stateless component or stateful component – they expect to be implemented by managed components. Furthermore, they capture the supported deployment package – WAR in this example – and runtime environment for which they have been developed: $SI_{3.1}$ of stateless component has the postcondition "WAR on Elastic

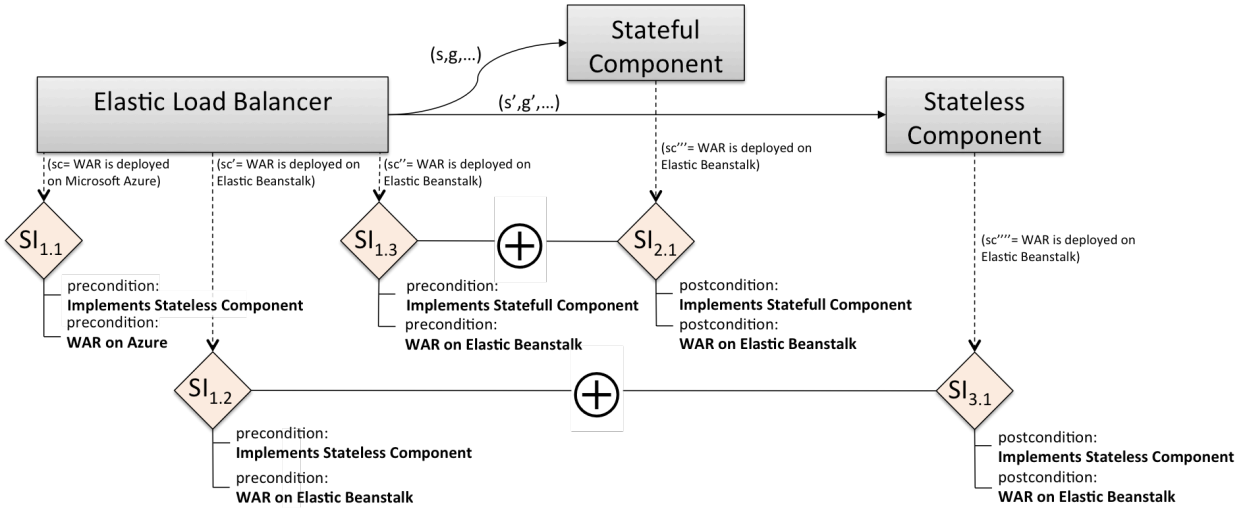

Figure 3. Solution Implementations in the domain of cloud application architecture linked to patterns and aggregated by Aggregation Operators.

Beanstalk" while $SI_{1.2}$ of elastic load balancer is enriched with the precondition "WAR on Elastic Beanstalk" and $SI_{1.1}$ with "WAR on Azure". The previously introduced Aggregation Operator interprets these dependencies and, for example, composes $SI_{3.1}$ and $SI_{1.2}$. During this task, the configuration parameters of the solutions are adjusted by the operator, i.e., the elastic load balancer is configured with the address of the stateless component to be managed. As some of this information may only become known after the deployment of a component, the configuration may also be handled during the deployment.

Following, this example is concretely demonstrated by an AWS Cloud Formation template [28] generated by the Aggregation Operator in Listing 1. An AWS Cloud Formation template is a configuration file, readable and processable by the AWS Cloud to automatically provision and configure cloud resources. For the sake of simplicity the depicted template in Listing 1 shows only the relevant parts of the template, which are adapted by the Aggregation Operator. To run the example scenario on AWS, three parts are needed within the AWS Cloud Formation template to reflect the aggregation of $SI_{3.1}$ and $SI_{1.2}$: (i) an elastic load balancer (MyLB), which is able to scale components, (ii) a launch configuration (MyCfg), which provides configuration parameters about an Amazon Machine Image (AMI) containing the implementation of stateless component as well as a runtime to execute the component in the form of an AWS Elastic Compute Cloud (EC2) [32] instance and, (iii) an autoscaling group (MyAutoscalingGroup) to define scaling parameters used by the elastic load balancer and the wiring of the elastic load balancer and the launch configuration.

```
"MyLB" : {
    "Type" : "AWS::ElasticLoadBalancing::LoadBalancer",
    "Properties" : {
        "Listeners" : [ {
            "LoadBalancerPort" : "80",
            "InstancePort" : "80",
            "Protocol" : "HTTP"
        } ],
    }
},
"MyCfg" : {
    "Type" : "AWS::AutoScaling::LaunchConfiguration",
    "Properties" : {
        "ImageId" : { "ami-statelessComponent" },
        "InstanceType" : { "m1.large" },
    }
},
"MyAutoscalingGroup" : {
    "Type" : "AWS::AutoScaling::AutoScalingGroup",
    "Properties" : {
        …
        "LaunchConfigurationName" : { "Ref" : "MyCfg"},
        "LoadBalancerNames" : [ { "Ref" : "MyLB" } ]
        …
    }
}
```

Listing 1. Extract from AWS Cloud Formation template produced by an Aggregation Operator to aggregate configuration snippets to aggregate elastic load balancer and stateless component.

MyLB defines an AWS elastic load balancer for scaling Hypertext Transfer Protocol (HTTP) requests on port 80. Further, MyCfg defines the AMI ami-statelessComponent in the property ImageId, which is used for provisioning new instances by an elastic load balancer. The autoscaling group MyAutoscalingGroup wires the elastic load balancer and the stateless component instances by means of referencing the properties LoadBalancerNames and LaunchConfigurationName to MyLB and MyCfg, respectively. Since all the mentioned properties are in charge of enabling an elastic load balancer instance to automatically scale and load balance instances of components contained in an AMI, an Aggregation Operator can dynamically adapt those properties based on the selected Solution Implementations to be aggregated. So, presuming that ami-statelessComponent contains an implementation of $SI_{3.1}$, an Aggregation Operator can aggregate $SI_{3.1}$ and $SI_{1.2}$ by adapting the mentioned properties and, therefore, provides an executable configuration template for AWS Cloud Formation. The same principles can be applied to aggregate $SI_{1.3}$ and $SI_{2.1}$ because of their matching preconditions and postconditions. By adapting the ImageId of the LaunchConfiguration to an AMI, which runs an AWS EC2 instance with a deployed stateful component, the Aggregation Operator can aggregate $SI_{1.3}$ and $SI_{2.1}$.

Further, $SI_{1.1}$ has precondition "WAR on Azure" and is, therefore, incompatible with $SI_{2.1}$ and $SI_{3.1}$, i.e., $SI_{1.1}$ cannot be combined with these Solution Implementations due to their preconditions and postconditions. The selection of a Solution Implementation, therefore, may restrict the number of matching Solution Implementations of the succeeding pattern since postconditions of the first Solution Implementation have to match with preconditions of the second. This way, the space of concrete solutions is reduced based on the resulting constraints of a selected Solution Implementation. To elaborate a solution to a overall problem described by a sequence of patterns exactly one Solution Implementation has to be selected for each pattern in the sequence considering its selection criteria to match non-functional requirements, as well as postconditions of the former Solution Implementation.

B. Deriving Solution Implementations in the Domain of Cloud Application Management

In this section, we show how the presented approach can be applied in the domain of cloud application management. Therefore, we describe how applying *management patterns* introduced in [10][29] to cloud applications can be supported

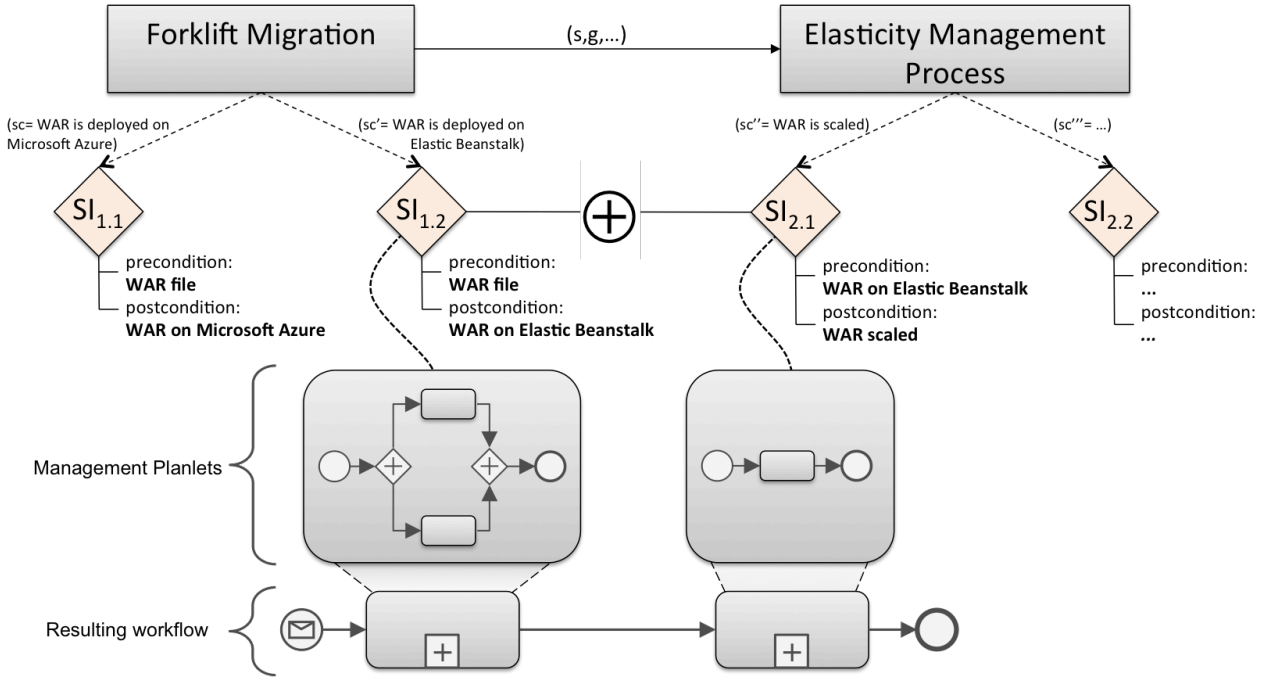

Figure 4. Management Planlets are Solution Implementations in the domain of cloud management linked to patterns and aggregated by an Aggregation Operator.

by reusing and aggregating predefined Solution Implementations in the form of executable management workflows.

In the domain of cloud application management, applying the concept of patterns is quite difficult as the refinement of a pattern's abstract solution to an executable management workflow for a certain use case is a complex challenge: (i) mapping abstract conceptual solutions to concrete technologies, (ii) handling the technical complexity of integrating different heterogeneous management APIs of different providers and technologies, (iii) ensuring non-functional cloud properties, (iv) and the mainly remote execution of management tasks lead to immense technical complexity and effort when refining a pattern in this domain. The presented approach of Solution Implementations enables to provide completely refined solutions in the form of executable *management workflows* that already consider all these aspects. Thus, if they are linked with the corresponding pattern, they can be selected and executed directly without further adaptations. This reduces the (i) required management knowledge and (ii) manual effort to apply a management pattern significantly. To apply the concept of Solution Implementations to this domain, two issues must be considered: (i) selection and (ii) aggregation of Solution Implementations in the form of management workflows.

To tackle these issues, we employ the concept of *management planlets*, which was introduced in our former research on cloud application management automation [24]. Management planlets are *generic management building blocks* in the form of workflows that implement management tasks such as installing a web server, updating an operating system, or creating a database backup. Each planlet exposes its functionality through a formal specification of its *effects* on components, i.e., its *postconditions*, and defines optional *preconditions* that must be fulfilled to execute the planlet. Therefore, each specific precondition of a planlet must be fulfilled by postconditions of other planlets. Thus, planlets can be combined to implement a more sophisticated management task, such as scaling an application. If two or more planlets are combined, the result is a *composite management planlet (CMP)*, which can be recursively combined with other planlets again: the CMP inherits all postconditions of the orchestrated planlets and exposes all their preconditions, which are not fulfilled already by the other employed planlets. Thus, management planlets provide a recursive aggregation model to implement management workflows. Based on these characteristics, management planlets are ideally suited to implement management patterns in the form of concrete Solution Implementations. We create Solution Implementations, which implement a pattern's refinement for a certain use case by orchestrating several management planlets to an overall composite management planlet that implements the required functionality in a modular fashion as depicted in Figure 4.

As stated above, selection and aggregation of Solution Implementations must be considered, the latter if multiple patterns are applied together. For example, Figure 4 shows two management patterns: (i) *forklift migration* [29] – application functionality is migrated with allowing some downtime and (ii) *elasticity management process* [10] – application functionality is scaled based on experienced workload. Both patterns are linked to two Solution

Implementations each in the form of composite management planlets. The forklift migration pattern provides two Solution Implementations: one migrates a Java-based web application (packaged as WAR file) to Microsoft Azure [22], another to Amazon Elastic Beanstalk [21]. Thus, if the user selects this pattern and chooses the Selection Criteria defining that a WAR application shall be migrated to Elastic Beanstalk, $SI_{1.2}$ is selected. Whether this Solution Implementation is applicable at all depends on the context: if the application to be migrated is a WAR application, then the Solution Implementation is appropriate. Equally to this pattern, the elasticity management process pattern shown in Figure 4 provides two Solution Implementations: one provides executable workflow logic for scaling a WAR application on Elastic Beanstalk ($SI_{2.1}$). Thus, if these two patterns are applied together, the selection of $SI_{1.2}$ restricts the possible Solution Implementations of the second pattern, as only $SI_{2.1}$ is applicable (its preconditions match the postconditions of $SI_{1.2}$). As a result, the selection of appropriate Solution Implementations can be reduced to the problem of (i) matching Selection Criteria to postconditions of Solution Implementations and (ii) matching preconditions and postconditions of different Solution Implementations to be combined.

After Solution Implementations of different patterns have been selected, the second issue of aggregation has to be tackled to combine multiple Solution Implementations in the form of workflows into an overall management workflow that incorporates all functionalities. Therefore, we implement a single Aggregation Operator for this pattern language as described in the following: to combine multiple Solution Implementations, the operator integrates the corresponding workflows as subworkflows [27]. The control flow, which defines the order of the Solution Implementations, i.e., the subworkflows, is determined based on the patterns' solution path depicted in Figure 2. So in general, if a pattern is applied before another pattern, also their corresponding Solution Implementations are applied in this order.

VI. SOLUTION IMPLEMENTATIONS PROTOTYPE

To prove the approach's technical feasibility, we implemented a prototype. That consists of two integrated components: (i) a pattern repository and (ii) a workflow generator. The pattern repository aims to capture patterns and their cross-references in a domain-independent way to support working with patterns. Based on semantic wiki-technology [11] it enables capturing, management and search of patterns. To adapt to different pattern domains, the pattern format is freely configurable. The pattern repository already contains various patterns from different domains like cloud computing patterns, data patterns and costume patterns to demonstrate the genericity of our approach. The cross-references between the patterns enable an easy navigation through the pattern languages. Links like "apply after" or "combined with" supports to connect the patterns to result in a pattern language. The pattern repository does not only contain the patterns and their cross-references but can be connected to a second repository containing the solution implementations of these patterns. Also, based on semantic wiki-technology we implemented a Solution Implementation repository for the domain of costume patterns [14]. Here, for example, the concrete costumes of a sheriff occurring in a film can be understood as the Solution Implementation of a sheriff costume pattern. By connecting the pattern to a Solution Implementation as a concrete solution of the abstracted solution of the pattern the application of the pattern in a certain context is facilitated.

The combination of several concrete Solution Implementations has been prototyped for the domain of cloud management patterns. A workflow generator has been built that is used to combine different management planlets to an overall workflow implementing a solution to a problem that requires the use of multiple patterns. The input for this generator is a partial order of (composite) management planlets, i.e., Solution Implementations that have to be orchestrated into an executable workflow. This partial order is determined by the relations of combined patterns: if one pattern is applied after another pattern, also their Solution Implementations, i.e., management planlets, have to be executed in this order. The workflow generator creates BPEL-workflows while management planlets are also implemented using BPEL. As BPEL is a standardized workflow language, the resulting management plans are portable across different engines and cloud environments supporting BPEL as workflow language, which is in line with TOSCA [25][26].

VII. CONCLUSION AND FUTURE WORK

In this paper, we introduced the concept of Solution Implementations as concrete instances of a pattern's solution. We showed how patterns and pattern languages can be enriched by Solution Implementations and how this approach can be integrated into a pattern repository. To derive concrete solutions for problems that require the application of several patterns we proposed a mechanism to compose these solutions from concrete solutions of the required patterns by means of operators. We concretized the general concept of Solution Implementations in the domain of cloud management by introducing management planlets as examples for Solution Implementations. We verified the approach by means of a prototype of an integrated pattern repository and workflow generator.

Currently, we extend the implemented repository for solution knowledge in the domain of costume design to capture Solution Implementations more efficiently. This repository integrates patterns and linked Solution Implementations in this domain and we are going to enlarge the amount of costume Solution Implementations. We are also going to tackle the limitation of the presented approach to not only work on solution implementation sequences but also on aggregations of concrete solution instances not ordered temporally due to pattern sequences. Since Solution Implementations are composed by Aggregation Operators we are going to enhance our pattern repositories to also store and manage the Aggregation Operators. Finally, we will investigate Aggregation Operators in domains, besides the above mentioned to formulate a general theory of Solution Implementations and Aggregation Operators.

 ISBN: 978-1-61208-343-8

REFERENCES

[1] C. Alexander, "The timeless way of building," Oxford University Press, 1979.

[2] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, "A pattern language: towns, buildings, constructions," Oxford University Press, 1977.

[3] R. Reiners, Bridge Pattern Library, http://bridge-pattern-library.fit.fraunhofer.de/pattern-library/, last accessed on 2014.01.30.

[4] C. Fehling, F. Leymann, R. Mietzner, and W. Schupeck, "A collection of patterns for cloud types, cloud service models, and cloud-based application architectures," http://www.cloudcomputingpatterns.org, last accessed on 2014.01.30, University of Stuttgart, Report 2011/05, Mai 2011.

[5] M. Falkenthal, D. Jugel, A. Zimmermann, R. Reiners, W. Reimann, and M. Pretz, "Maturity assessments of service-oriented enterprise architectures with iterative pattern refinement," Lecture Notes in Informatics - Informatik 2012, September 2012, pp. 1095–1101.

[6] R. Reiners, "A pattern evolution process – from ideas to patterns," Lecture Notes in Informatics – Informatiktage 2012, March 2012, pp. 115–118.

[7] U. van Heesch, Open Pattern Repository, http://www.patternrepository.com, last accessed on 2014.01.30.

[8] M. Demirköprü, "A new cloud data pattern language to support the migration of the data layer to the cloud," in German "Eine neue Cloud-Data-Pattern-Sprache zur Unterstützung der Migration der Datenschicht in die Cloud," University of Stuttgart, diploma thesis no. 3474, 2013.

[9] U. Zdun, "Systematic pattern selection using pattern language grammars and design space analysis," Software: Practice and Experience, vol. 37, 2007, pp. 983–1016.

[10] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, "Cloud computing patterns," Springer, 2014.

[11] N. Fürst, "Semantic wiki for capturing design patterns," in German "Semantisches Wiki zur Erfassung von Design-Patterns," University of Stuttgart, diploma thesis no. 3527, 2013.

[12] G. Hohpe and B. Wolf, "Enterprise integration patterns: designing, building, and deploying," Addison-Wesley, 2004.

[13] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, "Pattern-oriented software architecture volume 1: a system of patterns," Wiley, 1996.

[14] D. Kaupp, "Application of semantic wikis for solution documentation and pattern identification," in German "Verwendung von semantischen Wikis zur Lösungsdokumentation und Musteridentifikation," University of Stuttgart, diploma thesis no. 3406, 2013.

[15] R. Porter, J. O. Coplien, and T. Winn, "Sequences as a basis for pattern language composition," in Science of Computer Programming, Special issue on new software composition concepts, vol. 56, April 2005, pp. 231–249.

[16] F. Salustri, "Using pattern languages in design engineering," Proceedings of the International Conference on Engineering Design, August 2005, pp. 248–362.

[17] D. Schumm, J. Barzen, F. Leymann, and L. Ellrich, "A pattern language for costumes in films," Proceedings of the 17th European Conference on Pattern Languages of Programs (EuroPLoP), July 2012, pp. C4-1–C4-30.

[18] T. Iba, T. Miyake, "Learning patterns: a pattern language for creative learners II," Proceedings of the 1st Asian Conference on Pattern Languages of Programs (AsianPLoP 2010), March 2010, pp. I-41 – I-58.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: elements of reusable object-oriented software," Addison-Wesley, 1995.

[20] M. Fowler, "Patterns of enterprise application architecture," Addison-Wesley, 2003.

[21] Amazon, Elastic Beanstalk, http://www.amazon.com/elasticbeanstalk, last accessed on 2014.01.30.

[22] Microsoft, Microsoft Azure, http://www.windowsazure.com, last accessed on 2014.01.30.

[23] C. Fehling, F. Leymann, R. Retter, D. Schumm, and W. Schupeck, "An architectural pattern language of cloud-based applications", Proceedings of the 18th Conference on Pattern Languages of Programs (PLoP), October 2011, pp. A-20–A-30.

[24] U. Breitenbücher, T. Binz, O. Kopp, and F. Leymann, "Pattern-based runtime management of composite cloud applications", Proceedings of the 3rd International Conference on Cloud Computing and Service Science (CLOSER), May 2013, pp. 475–482.

[25] OASIS, Topology and Orchestration Specification for Cloud Applications Version 1.0, http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html, last accessed on 2014.01.30.

[26] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, "TOSCA: portable automated deployment and management of cloud applications," in Advanced Webservices, A. Bouguettaya, Q. Z. Sheng, F. Daniel, Eds., Springer, 2014, pp. 527–549.

[27] O. Kopp, H. Eberle, and F. Leymann, "The subprocess spectrum", Proceedings of the 3rd Business Process and Services Computing Conference (BPSC), September 2010, pp. 267–279.

[28] Amazon, AWS Cloud Formation, http://aws.amazon.com/cloudformation/, last accessed on 2014.01.30.

[29] C. Fehling, F. Leymann, S. T. Ruehl, M. Rudek, and S. Verclas "Service migration patterns – decision support and best practices for the migration of existing service-based applications to cloud environments," Proceedings of the IEEE International Conference on Service Oriented Computing and Applications (SOCA), December 2013, in press.

[30] Amazon, Amazon Web Services, http://aws.amazon.com, last accessed on 2014.01.30.

[31] C. Fehling, F. Leymann, J. Rütschlin, D. Schumm, "Pattern-based development and management of cloud applications," Future Internet, vol. 4, 2012, pp. 110–141.

[32] Amazon, AWS EC2, http://aws.amazon.com/de/ec2/, last accessed on 2014.04.10.

[33] C. Fehling, F. Leymann, R. Retter, D. Schumm, W. Schupeck, "An architectural pattern language of cloud-based applications," Proceesings of the 18th Conference on Pattern Languages of Programs (PLoP), Oct. 2011, pp. A-20 – A-21

 ISBN: 978-1-61208-343-8