Solution Languages: Easing Pattern Composition in Different Domains

Michael Falkenthal, Johanna Barzen, Uwe Breitenbücher, and Frank Leymann
Institute of Architecture of Application Systems
University of Stuttgart
Stuttgart, Germany
Email: {lastname}@iaas.uni-stuttgart.de

Abstract—**Patterns and pattern languages are a pervasive means to capture proven solutions for frequently recurring problems. However, there is often a lack of concrete guidance to apply them to concrete use cases at hand. Since patterns capture the essence of many solutions, which have practically proven to solve a problem properly, the knowledge about applying them to concrete individual problems at hand is lost during the authoring process. This is because information about how to apply a pattern in particular fields, technologies, or environmental contexts is typically lost due to the abstract nature of the solution of a pattern. In our previous works, we presented (i) the concept of linking concrete solutions to patterns in order to ease the pattern application and (ii) how these concrete solutions can be organized into so-called Solution Languages. In this work, we build upon these concepts and show the feasibility of Solution Languages via their application in different domains. Finally, we show how Solution Languages can be authored via a wiki-based prototype.**

Keywords*–*Pattern Language; Solution Language; Pattern Application; Solution Selection; Digital Humanities.

I. Introduction

In many domains, expertise and proven knowledge about how to solve frequently recurring problems are captured into patterns. Besides the conceptual solution knowledge captured into patterns also concrete realizations can ease and guide the elaboration of overall solutions. In this work we build upon our concept of *Solution Languages* [1] to show its general feasibility by application scenarios in different domains. Thereby, we provide evidence for the generality of Solution Languages by means of applications in IT domains and, exemplarily, in the non-IT domain of the digital humanities.

Originated by Christopher Alexander et al. [2] in the domain of building architecture, the pattern concept was also heavily applied in many disciplines in computer science. Patterns were authored, e.g., to support object-oriented design [3], for designing software architectures [4], for human-computer interaction [5], to integrate enterprise applications [6], for documenting collaborative projects [7], to support application provisioning [8] or to foster the understanding of new emerging fields like the Internet of Things [9] [10]. Triggered by the successful application of the pattern concept in computer science, it also gains momentum in non-IT disciplines such as creative learning [11] as well as disaster prevention and surviving [12]. Recently, collaborative research led to the application of patterns to the domain of *digital humanities* [13].

In general, patterns capture domain knowledge as *nuggets of advice*, which can be easily read and understood. They are interrelated with each other to form pattern languages, which ease and guide the navigation through the domain knowledge. This is often supported by links between patterns that carry specific semantics that help to find relevant other patterns based on a currently selected one [14]. In previous work, we showed that this principle can be leveraged to organize patterns on different levels of abstraction into pattern languages [15]. *Refinement links* can be used to establish navigation paths through a set of patterns, which lead a user from abstract and generic patterns to more specific ones that, e.g., provide technology-specific implementation details about the problem – often presented as implementation examples. We further showed that also concrete solutions, i.e., concrete artifacts that implement a solution described by a pattern, can be stored in a solution repository and linked to patterns [16] [17]. Thus, we were able to show that pattern-based problem solving is not only limited to the conceptual level, but rather can be guided via pattern refinement towards technology-specific designs and, finally, the selection and reuse of concrete solutions.

However, this approach still lacks guidance for navigation through the set of concrete solutions. Currently, navigation is only enabled on the level of pattern languages, while it is not possible to navigate from one concrete solution to others, due to missing navigation structures. This hinders the reuse of available concrete solutions especially in situations when many different and technology-specific concrete implementations of patterns have to be combined. As a result, it is neither easily understandable which concrete solutions can be combined to realize an aggregated solution, nor which working steps actually have to be done to conduct an aggregation. Thus, an approach is missing that allows to systematically document such knowledge in an easily accessible, structured, and human-readable way.

Therefore, we present the concept of *Solution Languages*, which introduces navigable semantic links between concrete solutions. A Solution Language organizes concrete solution artifacts analogously to pattern languages organize patterns. Their purpose is to ease and guide the navigation through the set of concrete solutions linked to patterns of a pattern language and, thus, ease the actual implementation of patterns via the reuse of already present concrete solutions. Thereby, knowledge about how to aggregate two concrete solutions is documented on the semantic link connecting them. This concept results in navigable networks of concrete solution artifacts, which can be aggregated to ease the implementation of overall solutions. It connects the perspective design with concrete implementations resulting in a novel approach for pattern-based software engineering.

The remainder of this paper is structured as following: as in our previous work [1], we provide background information and give a more detailed motivation in Section II. Then, we introduce the concept of Solution Languages and a means to add knowledge about solution aggregation in Section III. Besides the application in the domain of cloud application architecture, we extend the validation of our approach by an application scenarios in the domain of cloud application management. Further, we discuss how the presented concept of Solution Languages can be applied in domains apart from information technology (IT) in the domain of costumes in films in Section IV. We show the technical feasibility of Solution Languages by a prototype based on wiki-technology and by implementing the presented use cases of cloud application architecture and cloud application management in Section V. We discuss related work in Section VI and, finally, conclude this work in Section VII by an outlook to future work.

II. Background and Motivation

Patterns document proven solutions for recurring problems. They are human-readable documentations of domain expertise. Thereby, their main purpose is to make knowledge about how to effectively solve problems easily accessible to readers. According to Meszaros and Doble [18], they are typically written and structured using a common format that predefines sections such as the *Problem*, which is solved by a pattern, the *Context* in which a pattern can be applied, the *Forces* that affect the elaboration of concrete solutions, the *Solution*, which is a description of how to solve the exposed problem, and a *Name* indicating the essence of a pattern's solution.

Patterns are typically not isolated from each other but are linked with each other to enable the navigation from one pattern to other ones, which are getting relevant once it is applied. In this manner, a navigable network of patterns is established — a *pattern language* [19]. Often, a pattern language is established by referring other patterns in the running text of a pattern by mentioning them. This applies, especially, to pattern languages that are published as a monograph. Using wikis as platforms for authoring and laying out a library of patterns has further enabled to establish semantic links between patterns [7] [20]. This allows to enrich a pattern language to clearly indicate different navigation possibilities by different link types. Such link types can state, e.g., *AND*, *OR*, and *XOR* semantics, describing that after the application of a pattern other patterns are typically also applied, that there are additional patterns, which can be alternatively applied, or that there is an exclusive choice of further patterns that can be applied afterwards, respectively [7]. Further, they can tell a reader, for example, that a pattern is dealing with the equivalent problem of another pattern, but gives solution advice on a more fine-grained level in terms of additional implementation- and technology-specific knowledge [15]. Thus, the navigation through and even between pattern languages can be eased significantly.

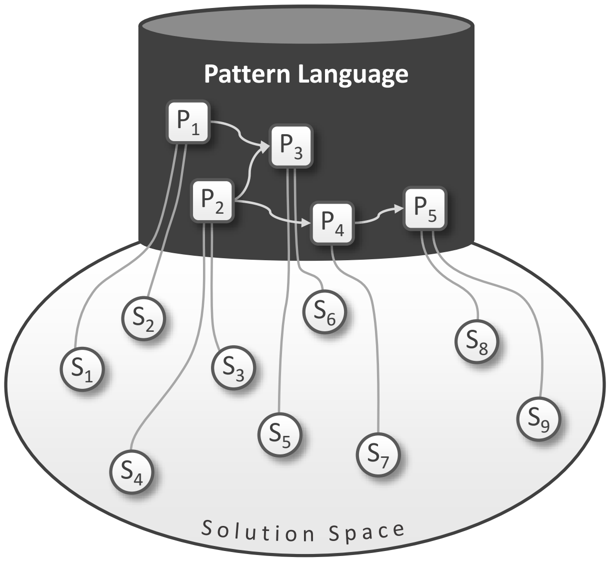


Figure 1. Missing Navigation Support through the Space of Concrete Solutions connected to a Pattern Language

Since patterns and pattern languages capture the essence and expertise from many concrete solutions of recurring problems, implementation details, such as technology-specific or environmental constraints, which affect the actual application of a pattern for specific problems at hand, are abstracted away during the pattern authoring process [21] [22]. As a result, this abstraction ensures that only the *conceptual core ideas* of how to solve a problem in a context are captured into a pattern, which makes the pattern applicable to many similar concrete use cases that may occur. In the course of this, the application of patterns for specific use cases gets unnecessarily hard because concrete solutions, i.e., implementations of a pattern, are lost during this authoring process. Thus, we showed that connecting concrete solutions to patterns in order to make them reusable when a pattern has to be applied is a valuable concept to save time consuming efforts [16] [17]. This concept is depicted in Fig. 1, where a pattern language is illustrated as a graph of connected patterns at the top. Based on the conceptual solution knowledge, the pattern language opens a solution space, illustrated as an ellipse below the pattern language, which is the space of all possible implementations of the pattern language. Concrete solutions that implement individual patterns of the pattern language are, consequently, located in the solution space and are illustrated as circles. They are linked with the pattern they implement, which enables to directly reuse them once a pattern is selected from the pattern language in order to be applied.

However, while navigation through conceptual solutions is provided by pattern languages in terms of links between patterns, such navigation capabilities are currently not present on the level of concrete solutions due to the absence of links between the concrete solutions. Thus, if a concrete solution is selected, there is no guidance to navigate through the set of all available and further relevant concrete solutions.

Navigation is only possible on the conceptual level of patterns by navigation structures of the pattern language. This is time consuming if experts have their conceptual solution already in mind and want to quickly traverse through available concrete solutions in order to examine if they can reuse some of them for implementing their use case at hand. Further, if a set of concrete solutions is already present that provides implementation building blocks for, e.g., a specific technology, it is often necessary to quickly navigate between them in order to understand their dependencies for reusing them. This is specifically the case if concrete solutions cannot be reused directly but need to be adapted to a specific use case, especially if they have to be used in combination. Then, they still can provide a valuable basis for starting adaptions instead of recreating a concrete solution from scratch. Finally, if some concrete solutions have proven to be typically used in combination it is valuable to document this information to ease their reuse. While this could be done on the level of a pattern language, we argue that this is bad practice because implementation details would mix up with the conceptual character of the pattern language. This would require to update a pattern language whenever implementation insights have to be documented. It can get cumbersome, if concrete solutions are collected over a long period of time and technology shifts lead to new implementations and approaches on how to aggregate them, while the more general pattern language stays the same.

Therefore, to summarize the above discussed deficits, there is (i) a lack of organization and structuring at the level of concrete solutions, which (ii) leads to time consuming efforts for traversing concrete solutions, and that (iii) prevents the documentation of proven combinations of concrete solutions.

III. Solution Languages: A Means to Structure, Organize, and Compose Concrete Solutions

To overcome the discussed deficits, we introduce the concept of *Solution Languages*. The core idea of Solution Languages is to transfer the capabilities of a pattern language to the level of concrete solutions having the goal of easing and guiding the application of patterns via reusing concrete solutions in mind. Specifically, the following capabilities have to be enabled on the level of concrete solutions: (R_1) navigation between concrete solutions, (R_2) navigation guidance to find relevant further concrete solutions, and (R_3) documentation capabilities for managing knowledge about dependencies between concrete solutions, e.g., how to aggregate different concrete solutions to elaborate comprehensive solutions based on multiple patterns.

A. Ease and Guide Traversing of Concrete Solutions

In our previous work we have shown how concrete solutions can be linked with patterns in order to ease pattern application [16] [17]. While this is indicated in Fig. 1 by the links between the patterns and the concrete solutions in the solution space, the systematic structuring and organization of the concrete solutions is still an open issue (cf. Section II).

Thus, to realize the requirements (R_1) and (R_2), a Solution Language establishes links between concrete solutions, which are annotated by specific semantics that support a user to decide if a further concrete solution is relevant to solve his or her problem at hand. This is especially important, if multiple patterns have to be applied and, thus, also multiple linked concrete solutions have to be combined. Thereby, the semantics of a link can indicate that concrete solutions connected to different patterns *can be aggregated* with each other, that individual concrete solutions are *variants* that implement the same pattern, or if exactly one of more *alternative* concrete solutions can be used in combination with another one.

Depending on the needs of users also additional link semantics can be added to a Solution Language. To give one example, semantic links can be introduced that specifically indicate that selected concrete solutions *must not be aggregated*. This is useful in cases when concrete solutions can be technically aggregated on the one hand, but on the other hand implement non-functional attributes that prevent to create a proper aggregated solution. Such situations might occur, e.g., in the field of cloud computing, where applications can be distributed across different cloud providers around the world. Then, this is also implemented by the concrete solutions that are building blocks of such applications. Different concrete solutions can force that individual parts of an application are deployed in different regions of the world. In some cases, law, local regulations, or compliance policies of a company can restrict the distribution of components of an application to specific countries [23]. In such situations, it is very valuable to document these restrictions on the level of concrete solutions via the latter mentioned link type. This can prevent users from unnecessarily navigating to concrete solutions that are irrelevant in such use cases. Nevertheless, the concrete solutions that are not allowed to be used in a specific scenario can be kept in a Solution Language, e.g., for later reuse if preventing factors change or as a basis for adaptions that make them compliant.

While (R_1) and (R_2) can be realized by means of semantically typed links between concrete solutions as introduced above, (R_3) requires to introduce the concept of a *Concrete Solution Aggregation Descriptor (CSAD)*. A CSAD allows to annotate a link between concrete solutions by additional documentation that describes how concrete solutions can be aggregated in a human-readable way. This can be, e.g., a specific description of the working steps required to aggregate the concrete solutions connected by the annotated link. Beyond that, a CSAD can also contain any additionally feasible documentation, such as a sketch of the artifact resulting from the aggregation, which supports the user. The actual content of a CSAD is highly specific for the domain of the concrete solutions. The aggregation of concrete solutions that are programming code can, for example, often be described by adjustments of configurations, by manual steps to be performed in a specific integrated development environment (IDE), or by means of additional code snippets required for the aggregation. In other domains, such as the non-technical domain of costumes in films, the required documentation to aggregate concrete solutions looks quite different and can be, for example, a manual about how to combine different pieces of clothing, which in this case are concrete solutions, in order to achieve a desired impression

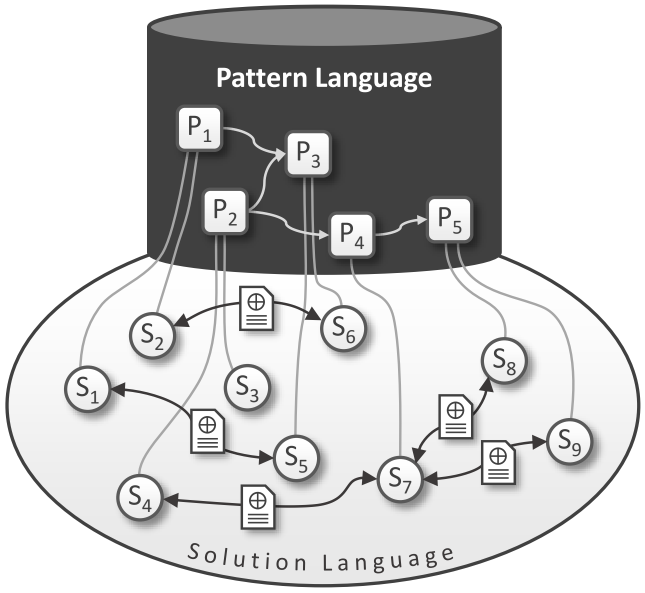


Figure 2. A Solution Language structures the Solution Space of a Pattern Language and enables navigation through relevant Concrete Solutions

of a character in a movie. Thus, a CSAD can be leveraged to systematically document concrete implementation knowledge about how to create aggregated overall solutions.

As a result, CSADs are the means to add arbitrary documentation to a Solution Language about how to aggregate concrete solutions. Hence, a Solution Language can be iteratively extended over time to preserve the expert knowledge of a domain on the implementation level the same way as pattern languages do on the conceptual level. Especially in situations when technologies are getting outdated and experts, which are required to maintain systems implemented in such technologies are getting only scarcely available, Solution Languages can be valuable instruments that preserve technology-specific implementation expertise and documentation. Since concrete solutions are also connected to the patterns they implement, conceptual as well as implementation knowledge can be kept easily accessible and inherently connected.

The overall concept of a Solution Language is illustrated in Fig. 2. There, concrete solutions are linked to the patterns they implement. This enables a user to navigate from patterns to concrete implementations that can be reused, as described in our earlier work [16] [17]. Additionally, the concrete solutions are also linked with each other in order to allow navigation on the level of concrete solutions. For the sake of simplicity and clarity, Fig. 2 focusses on links that represent *can be aggregated with* semantics, thus, we omitted other link types. Nevertheless, the relations between the concrete solutions can capture arbitrary semantics, such as those discussed above. The semantic links between concrete solutions and the fact that they are also linked to the patterns, which they implement, enables to enter the Solution Language at a certain concrete solution and allows to navigate among only the relevant concrete solutions that are of interest for a concrete use case at hand. For example, if concrete solutions are available that implement patterns in different technologies, then they typically cannot be aggregated. Thus, entering the Solution language at a certain concrete solution and then navigating among only those concrete solutions that are implemented using the same technology, using semantic links indicating this coherence (e.g., *can be aggregated with*), can reduce the effort to elaborate an overall composite solution significantly. Finally, Fig. 2 depicts CSADs attached to links between concrete solutions in the form of documents. These enrich the semantic links and provide additional arbitrary documentation on how to aggregate the linked concrete solutions. This way, a Solution Language delegates the principles of pattern languages to the level of concrete solutions, which helps to structure and organize the set of available concrete solutions. While a pattern language guides a user through a set of abstract and conceptual solutions in the form of patterns, a Solution Language provides similar guidance for combining concrete solutions to overall artifacts, all provided by semantic links between concrete solutions and additional documentation about how to aggregate them. Navigation support between concrete implementations of patterns cannot be given by a pattern language itself, because one pattern can be implemented in many different ways, even in ones that did not exist at the time of authoring the pattern language. Thus, the elucidated guidance is required on the solution level due to the fact that a multitude of different and technology-specific concrete solutions can implement the concepts provided by a pattern language.

B. Mapping Solution Paths from Pattern Languages to Solution Languages

Since pattern languages organize and structure patterns to a navigable network, they can be used to select several patterns to solve a concrete problem at hand by providing conceptual solutions. A user typically tries to find a proper entry point to the pattern language by selecting a pattern that solves his or her problem at least partially. Starting from this pattern, he or she navigates to further patterns in order to select a complete set of patterns that solve the entire problem at hand in combination. This way, several patterns are selected along paths through the pattern language. Thus, the selected patterns are also called a *solution path* through the pattern language [15] [24]. Figure 3 shows such a solution path by the selected patterns P_2, P_4, and P_5. If several solution paths proof to be successful for recurring use cases, this can be documented into the pattern language to present stories that provide use case-specific entry points to the pattern language [25]. Further, if several concrete solutions are often aggregated by means of the same CSAD, then this can reveal that there might be a candidate of a composite pattern that can be added to the pattern language by abstracting the underlying solution principles. This might be supported and automated by data mining techniques in specific domains [26].

Due to the fact that concrete solutions are linked with the patterns they implement, solution paths through a pattern language can support to find suitable entry points to the

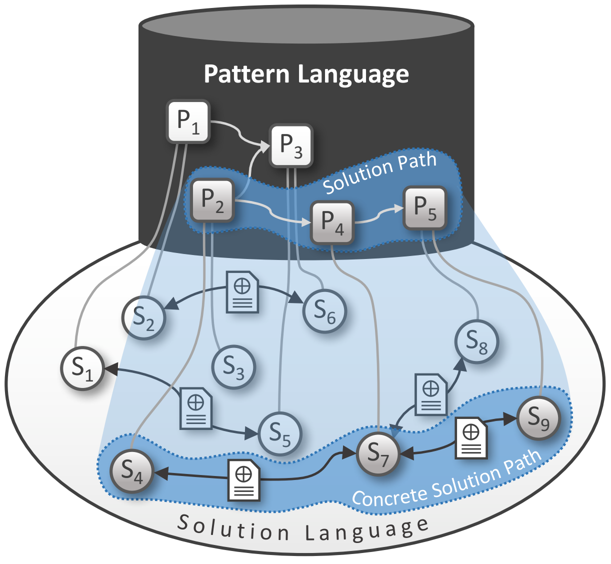


Figure 3. Solution Path from a Pattern Language projected to a Solution Language

corresponding Solution Language. Accordingly, a user can navigate from P_2 to the concrete solution S_4. From there, the Solution Language provides navigation support to find further concrete solutions that can be aggregated with S_4. If concrete solutions are available for all patterns contained in the solution path, and if these can be aggregated with each other, then the solution path can be mapped to a *concrete solution path* in the Solution Language. This is illustrated in Fig. 3 by the highlighted path from S_4 via S_7 to S_9 through the Solution Language. Concrete solution paths allow to translate design decisions that are taken on the conceptual level of the pattern language to reusable concrete solutions that are organized into the Solution Language. The mapping of the solution path to a corresponding set of concrete solutions of the Solution Language can, consequently, provide knowledge about how to elaborate an aggregated solution of the selected patterns by CSADs of the Solution Language, which can significantly speed up the elaboration of an overall composite concrete solution.

IV. Application of Solution Languages

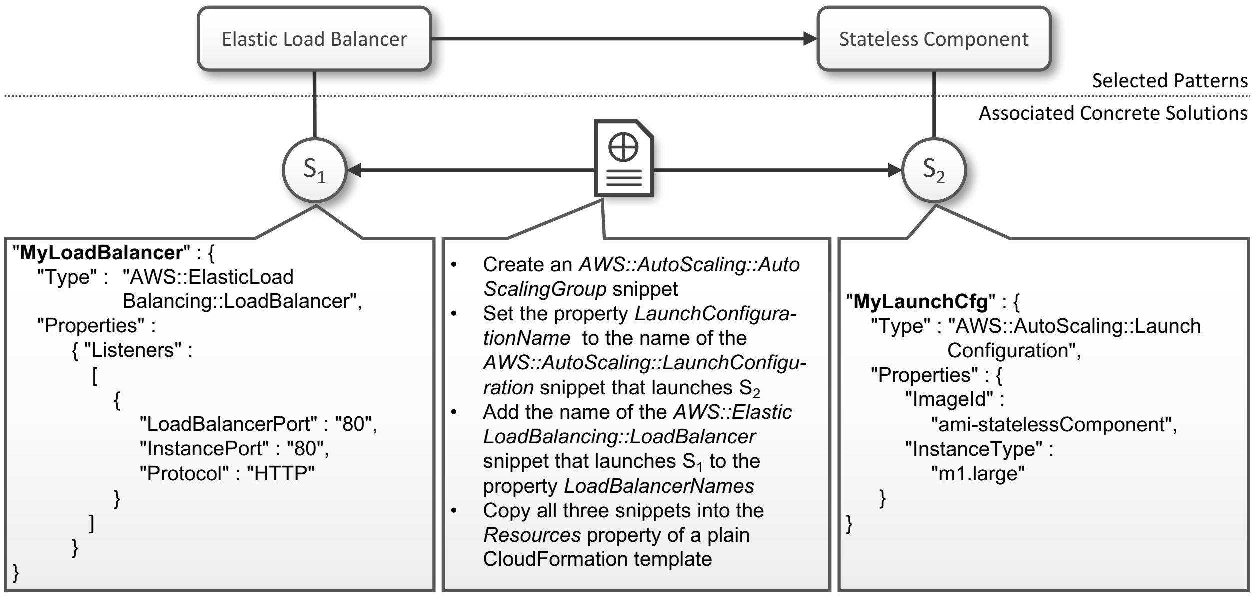
In the following, we show the feasibility of the Solution Languages concept on the basis of application scenarios in domains in which we have already investigated and researched pattern languages. These are the IT domains *Cloud Application Architecture* and *Cloud Application Management* as well as the non-IT domain of *Costumes in Films*. The latter scenario demonstrates that the concept of Solution Languages is not tied to IT but can be rather applied to other domains, too.

A. Application in the Domain of Cloud Architecture

The first application scenario deals with designing application architectures that natively support cloud computing characteristics and technologies. In this context, the pattern language of Fehling et al. [27] provides knowledge about tailoring applications to leverage the capabilities of cloud environments such as Amazon Web Services (AWS) [28]. One important capability in terms of cloud computing is the automatic and elastic scaling of compute resources. To enable this, the pattern language provides the patterns *Elastic Load Balancer* and *Stateless Component*. *Elastic Load Balancer* describes how the workload of an application can be distributed among multiple instances of the application. If the workload increases, additional instances are added to keep the application responsive. Once the workload decreases unnecessary instances are decommissioned to save processing power and expenses. Thereby, the actual workload, i.e., requests from clients, is spread among the different application instances by means of a so-called load balancer component, which maintains and manages the available endpoints of the instances. The *Elastic Load Balancer* pattern links to the *Stateless Component* pattern, which describes how components that contain the business logic of an application can manage their state externally, e.g., in an additional database. This behavior enables to scale them elastically because recently created instances can fetch state from the external datastore and write changes back to it. As a result, state synchronization among the different application instances is handled via the database and no further synchronization and state replication mechanisms are required. Both patterns are depicted at the top of Fig. 4, whereby a directed edge connects them indicating the pattern language structure expressing that once *Elastic Load Balancer* is used then also *Stateless Component* is ordinarily used.

Realizations of these patterns can be connected to them, as depicted in the figure by S_1 and S_2. These concrete solutions implement the patterns by means of AWS CloudFormation [29] snippets, which allows to describe collections of AWS-resources by means of a java script object notation (JSON)-based configuration language. Such configurations can be uploaded to AWS CloudFormation, which then automatically provisions new instances of the described resources. An excerpt of the CloudFormation snippet that describes a load balancer is shown on the left of Fig. 4. The *MyLoadBalancer* configuration defines properties of the load balancer, concretely the port and protocol, which are required to receive and forward workload. The corresponding CloudFormation snippet, which implements the concrete solution S_2 is shown on the right. So-called *Amazon Machine Images (AMI)* allow to package all information required to create and start virtual servers in the AWS cloud. Therefore, the *MyLaunchCfg* snippet of S_2 contains a reference to the AMI *ami-statelessComponent*, which is able to create and start a new virtual server that hosts an instance of a stateless component. The link between S_1 and S_2 illustrates that they *can be aggregated* in order to obtain an overall composite solution, which results in a complete configuration that allows the load balancer instance to distribute workload over instances of virtual servers hosting the stateless component.

If a user wants to aggregate both snippets, he or she can study the CSAD attached to the link between both concrete solutions, which is outlined in the middle of the



```
"MyLoadBalancer" : {
    "Type" :  "AWS::ElasticLoad
              Balancing::LoadBalancer",
    "Properties" :
        { "Listeners" :
            [
              {
                "LoadBalancerPort" : "80",
                "InstancePort" : "80",
                "Protocol" : "HTTP"
              }
            ]
        }
}
```

- Create an *AWS::AutoScaling::AutoScalingGroup* snippet
- Set the property *LaunchConfigurationName* to the name of the *AWS::AutoScaling::LaunchConfiguration* snippet that launches S_2
- Add the name of the *AWS::ElasticLoadBalancing::LoadBalancer* snippet that launches S_1 to the property *LoadBalancerNames*
- Copy all three snippets into the *Resources* property of a plain CloudFormation template

```
"MyLaunchCfg" : {
    "Type" : "AWS::AutoScaling::Launch
              Configuration",
    "Properties" : {
        "ImageId" :
            "ami-statelessComponent",
        "InstanceType" :
            "m1.large"
    }
}
```

Figure 4. Concrete Solution Aggregation Descriptor documenting how to aggregate concrete solutions in the form of two CloudFormation snippets

figure. The CSAD provides detailed information about the actual working steps that have to be performed in order to combine both CloudFormation snippets. Therefore, the CSAD describes that both snippets have to be aggregated via a so-called *AutoScalingGroup*, which is itself also a CloudFormation snippet. The AutoScalingGroup references both, the *MyLoadBalancer* and the *MyLaunchCfg* snippets via the properties *LaunchConfigurationName* and *LoadBalancerNames*. Finally, all three snippets have to be integrated into the property *Resources* of a plain CloudFormation template. By documenting all this information into the Solution Language, (i) the link from concrete solutions in form of CloudFormation snippets to the patterns they implement, (ii) the semantic link between these concrete solutions indicating that they *can be aggregated*, and (iii) the detailed documentation about how to perform the aggregation can significantly ease the application of the two patterns to elaborate an overall combined solution.

B. Application in the Domain of Cloud Management

In this section, we apply the concept of Solution Languages to the domain of cloud application management. Considering that we build upon the application scenario described in the previous section. Thereby, we show how patterns from the pattern language by Fehling et al. [27] can also be used to deal with the question about how to systematically describe and model the interplay of the components of an application to enable the automated provisioning of new application instances via standard-compliant provisioning engines. Hence, in combination with the previous one, this use case shows that Solution Languages can be created and maintained addressing completely different aspects of a domain captured into overall and comprehensive pattern languages. Such a pattern language is the above introduced one by Fehling et al. [27], which covers different viewpoints of the field of cloud computing and, thus, acts as an entry point to Solution Languages providing and organizing completely different solution artifacts. Thereby, we specifically focus on the provisioning of cloud applications based on the OASIS cloud standard *Topology and Orchestration Specification for Cloud Applications* (TOSCA) [30].

Thus, in the following, we firstly explain the patterns of this application scenario to understand the cloud management specific issues to be tackled. Secondly, we give a brief description of the main concepts provided by TOSCA, which are required to understand the concrete solutions we use in this scenario. Finally, we show how CSADs can be used either to provide descriptions for manually aggregating concrete solutions or, respectively, also to automate the aggregation to overall composite solutions.

The application scenario is depicted in Fig. 5. There, the three patterns *Stateless Component*, *Elastic Infrastructure*, and *Key Value Store* are illustrated. While the pattern *Stateless Component* describes that the state of an application should be kept externally from its processing components, the pattern *Key Value Store* provides the conceptual solution of a datastore, which is specifically designed to store and retrieve data via identifying keys. For instance, if an application has to handle workload, which requires user sessions such as shopping carts of a web store, the user sessions constitute the state of the interaction with the user. Although components of the application have to process this state it should not be kept in the actual processing components as discussed above but moved to an external store, which can be a *Key Value Store* while the unique session ids of the user sessions can be used to select and manipulate the data. Thus, both *Stateless Component* and *Key Value Store* provide conceptual solutions, which have to be combined to an overall solution in order to create an application,

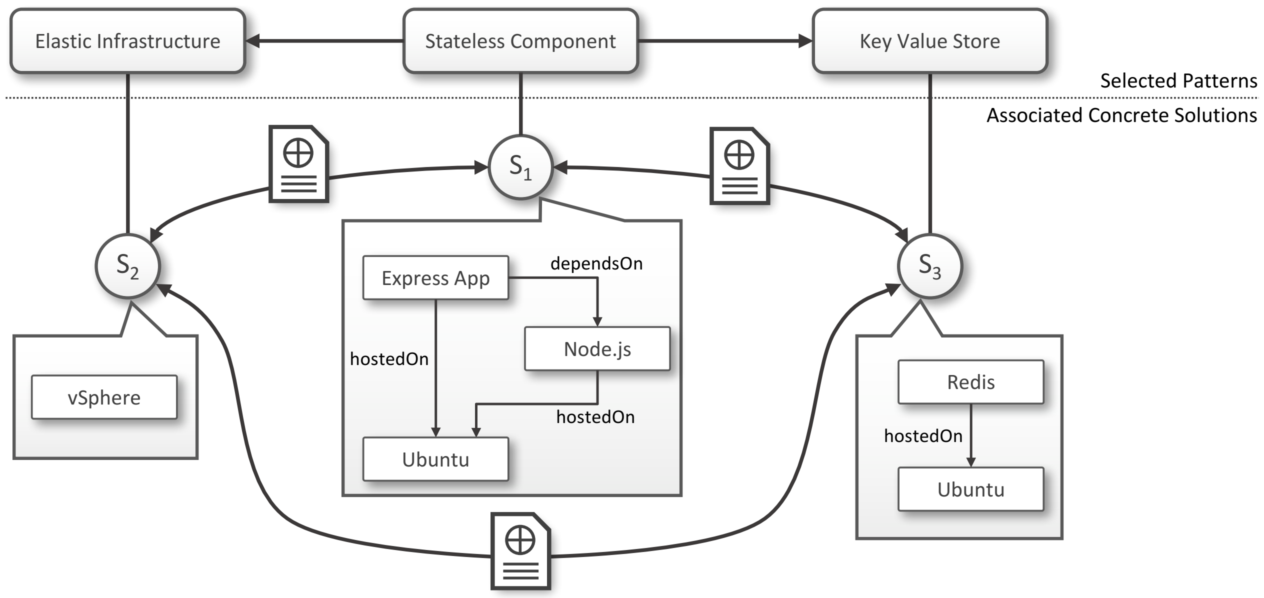


Figure 5. Solution Language comprising of Service Templates

which on the one hand can handle state and, on the other hand, can assure that processing components are stateless. However, to run such an application also a hosting environment is required. In the domain of cloud computing, where automation is important, hosting environments commonly follow the concept of an *Elastic Infrastructure*, which enables the automated provisioning and decommissioning of compute resources in the form of virtual machines. So, besides public cloud offerings, such as Amazon Elastic Compute Cloud [31] or Microsoft Azure Virtual Machines [32], also private cloud technologies, such as VMWare vSphere [33] or OpenStack [34], provide an application programming interface (API) allowing to access these offerings programmatically. Thus, from the perspective of cloud management *Elastic Infrastructure* provides a conceptual solution for elastically hosting cloud applications.

The conceptual knowledge provided by these three patterns can be complemented by an implementation-specific Solution Language capturing concrete solutions, which can be aggregated for the complete provisioning of applications. In this scenario, we leverage the expressiveness of TOSCA to populate the Solution Language as indicated in Fig. 5 by means of TOSCA *service templates*. A service template is the core entity of TOSCA to describe the *topology* of an application, i.e., its structure in terms of its components and the relations between them. Components are called *node templates* and relations *relationship templates*, which both can be abstracted into *node types* and *relationship types*, respectively, to enable their reuse in different service templates [35]. Besides these structural description also all required software executables (*deployment artifacts*) and management logic (*implementation artifacts*) to install, configure, start, stop, and terminate the application are contained. Thus, a service template can be grasped as a blue print of the application, which can be automatically instantiated to create new running instances of an application. Such service templates can be processed by TOSCA-compliant provisioning engines, such as OpenTOSCA [8] [36]. Thereby, the provisioning engine parses the modeled application topology and triggers all actions in terms of API-calls and executions of management logic to create a new application instance in the specified hosting environment.

A Solution Language comprising of service templates as concrete solutions is depicted in Fig. 4 by the concrete solutions S_1, S_2, and S_3. S_1 provides a realization of the *Stateless Component* pattern by means of a service template that describes the topology of an Express [37] application, which is a web framework based on Node.js [38].

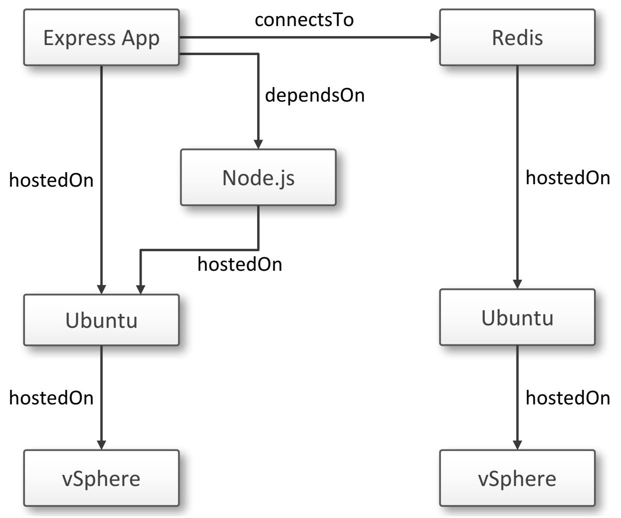


Figure 6. Aggregated Service Template

This reflects in the *dependsOn* relation between the Express App and the Node.js component in the depicted topology. Accordingly, both components are *hostedOn* an Ubuntu [39] operating system. A concrete solution of the *Key Value Store* pattern is provided by the concrete solution S_3 via a service template containing a node template that is capable of installing Redis [40], which is an open source key value store, onto an Ubuntu operating system. Finally, to bring up an entire application, S_2 provides an implementation of the *Elastic Infrastructure* pattern by means of a service template wrapping the API of a vSphere infrastructure.

The CSADs attached to the links between these concrete solutions can either be descriptions about how to manually combine these service templates in order to get an overall solution that can be provisioned, or they can also comprise of programs, which allow to automatically aggregate the different service templates. In the first case, the CSAD between S_1 and S_2 explains that the Ubuntu node template of S_1 has to be linked via a *hostedOn* relation with the vSphere node template of S_2 in order to be provisioned in a proper cloud environment based on the vSphere technology. Likewise, the Ubuntu node template of S_2 has to be wired with a vSphere node template. Further, the CSAD between S_1 and S_3 explains that the Express App node template of S_1 has to be linked via a *connectsTo* relation with the Redis node template of S_3. This can be done, for example, through a TOSCA-compliant modeling tool such as Eclipse Winery [41] but also directly in the respective TOSCA-files. Thus, the CSADs can provide different descriptions respective to available tooling.

However, in the second case, the CSADs can also link to programmatic solutions that allow the aggregation of different service templates. The TOSCA concepts of *requirements* and *capabilities* can be used to specify constrained requirements that have to be resolved and fulfilled by a TOSCA provisioning engine to generate provisionable service templates [42]. Using these concepts, on the one hand, the node templates Ubuntu of S_1 and S_3 can be associated with requirements stating that a hosting environment is required, which can provision and run Ubuntu. On the other hand, the vSphere node template of S_2 can expose the very same as a capability. Then, a TOSCA-compliant provisioning engine can aggregate S_1 and S_2 as well as S_3 and S_2 to resolve the defined requirements providing complete application stacks for provisioning and hosting. However, the resulting stacks, finally, have to be wired. Thus, to connect them the node template Express App, which implements the *Stateless Component* pattern, can be associated with a requirement stating that a *Key Value Store* is required to manage its state. Further, the Redis node template can, respectively, expose a capability fulfilling this requirement in order to enable the automatic aggregation of both service templates into an overall one [43] [42]. This results in a service template in which the aggregated topologies of S_1 and S_3 are connected via a *connectsTo* relationship between the node templates Express App and Redis. In this case, the aggregation is done automatically via the TOSCA completion mechanisms based on requirements and capabilities [43] [42]. Note that also the deployment and implementation artifacts of all node templates are packed into the overall service template to keep the newly created service template provisionable. The resulting aggregated application topology is depicted in Section IV-B and represents the service template based on the different concrete solutions linked to the patterns *Stateless Component*, *Key Value Store*, and *Elastic Infrastructure*. This service template can then be used as a starting point to add arbitrary business logic in the Express App to be processed as a stateless component. The application scenario has shown that CSADs can either be used to provide human-readable manuals about how to aggregate concrete solutions but also that CSADs can be provided as corresponding automation logic that enables the automation of aggregating concrete solutions to comprehensive ones.

C. Application in the Domain of Costumes in Films

In addition to the domain of IT, Solution Languages are also promising for rather different domains of pattern languages, as we want to prove by applying these concepts to the domain of costume languages in films [21]. Costume languages capture the knowledge of proven solutions about how to communicate a certain stereotype, its character traits or transformations, as well as geographical and historical setting of a film by the costumes worn by roles [21]. As depicted in Fig. 7, an example of a costume pattern is the *Sheriff*: a pattern describing all significant elements — like the shirt, the trousers, the ammunition belt and the wild west vest, the neckerchief, the boots and spurs, the cowboy hat and the sheriffs star — to communicate the stereotype of a sheriff in a western genre movie to the audience [44].

However, not only entire outfits are captured as costume patterns. Also proven principles describing, e.g., how to specifically wear or modify certain costumes or parts of them to indicate different character traits or how conditions of a costume can transport specific moods of a character are also captured into costume patterns. Therefore, as illustrated on the right of the pattern language in Fig. 7, another possible pattern is the pattern *Active Character* capturing actions and modifications to make a character look more active in the sense of being more energetic. Each of the depicted costume patterns has multiple concrete solutions connected to it representing concrete costumes in films. The Sheriff costume pattern, e.g., is linked to the concrete costumes worn by sheriffs in different western films, such as *John T. Chance* (John Wayne) in *Rio Bravo* (1959) or *Burnett* (Frank Wolf) in *Il grande silenzio* (1968), as indicated by S_1 and S_3 in Fig. 7. Moreover, concrete solutions of the *Active Character* pattern are, e.g., *Jake Lonergan* (Daniel Craig) in *Cowboys and Aliens* (2011) or *John T. Chance* (John Wayne) in *Rio Bravo* (1959). To create different arrangements of these patterns for specific scenes in films a costume designer can hark back to these captured concrete solutions to get an idea about the required costume and its style.

Other than the IT domain, which deals with concrete solutions that are intangible in the sense that they are often programming code or other forms of digital artifacts, in the domain of costumes in films, the concrete solutions are tangible artifacts. Thus, they could be kept in a wardrobe

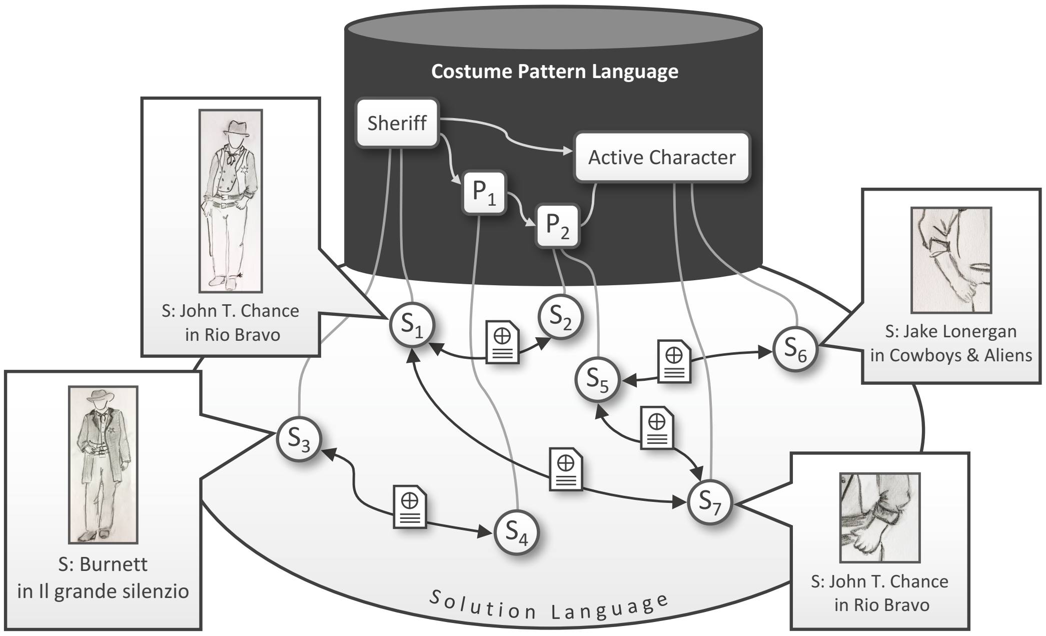

Costume Pattern Language
Sheriff
Active Character
P1
P2
S1
S2
S3
S4
S5
S6
S7
S: John T. Chance in Rio Bravo
S: Burnett in Il grande silenzio
S: Jake Lonergan in Cowboys & Aliens
S: John T. Chance in Rio Bravo
Solution Language

Figure 7. Example of a Costume Pattern Language linked to its Solution Language

of the costume designer at a specific film set or could also be provided by a costume rental. While the aggregation of intangible solutions can often be automated [16] [17] as described above, e.g., by merging code snippets to an aggregated solution, the aggregation of tangible solutions, such as concrete costumes, often has to be done manually. However, also in the case of tangible solutions, the concept of Solution Languages can be used for documenting knowledge about how to combine concrete solutions. Such knowledge is typically not systematically captured yet because of a missing methodical approach. Therefore, CSADs can be used to overcome this problem by documenting procedures and manuals describing the working steps to combine solutions.

In case of costumes in films [21], a Solution Language linking together all concrete costumes as concrete solutions of a costume pattern can be authored that allows to reuse already existing costumes for dressing actors. Thereby, a CSAD can describe, for example, how characters have to be dressed in order to create an intended effect. This information can be used by costume designers to create suitable costumes as required for particular scenes in a film. While Fig. 7 illustrates the general coherence between costumes as concrete solutions and costume patterns, in the following we describe a CSAD in the domain of costumes in films exemplarily.

When aiming to give the impression to the audience via the costume that a sheriff is particularly active in a specific scene, the *Sheriff* pattern has to be combined with the *Active Character* pattern. Further, concrete solutions are selected, which are linked with each other indicating that they can be combined as depicted in Fig. 8. Also a CSAD is attached to the relation between both concrete solutions containing all relevant information for combining them to achieve the desired impression by rolling up the sleeves of the sheriff's shirt. The CSAD specifically describes the actions to be performed by a costume designer in order to combine the concrete solutions under consideration. The CSAD depicted in Fig. 8 briefly illustrates this and shows how the concrete solution S_1 of the *Sheriff* pattern and the concrete solution S_2 of the *Active Character* pattern can be combined. The resulting modified costume, i.e., the combination of S_1 and S_2 is depicted on the right as a new combined concrete solution S_{new}. Note that this combined concrete solution can indicate further solution principles, which might worth to be captured into an additional pattern if created many times for different scenes and films.

V. PROTOTYPE

To proof the technical feasibility of the presented approach of Solution Languages, we implemented a prototype on the basis of *PatternPedia* [20]. PatternPedia is a wiki that is built upon the MediaWiki [45] technology and the Semantic MediaWiki extensions [46]. We implemented the application scenario about cloud architectures presented in the previous section. Therefore, we captured the cloud computing patterns in form of wiki pages into PatternPedia and added links between them accordingly to the pattern language of Fehling et al. [27]. We also added the concrete solutions in the form of AWS CloudFormation snippets to PatternPedia so that each AWS CloudFormation snippet is represented by a separate wiki page that references a file containing the corresponding JSON-code. Then, we linked the wiki pages of the concrete solutions with wiki pages representing the patterns they implement to enable the navigation from abstract solution principles captured in patterns to technology-specific implementations in the form of concrete

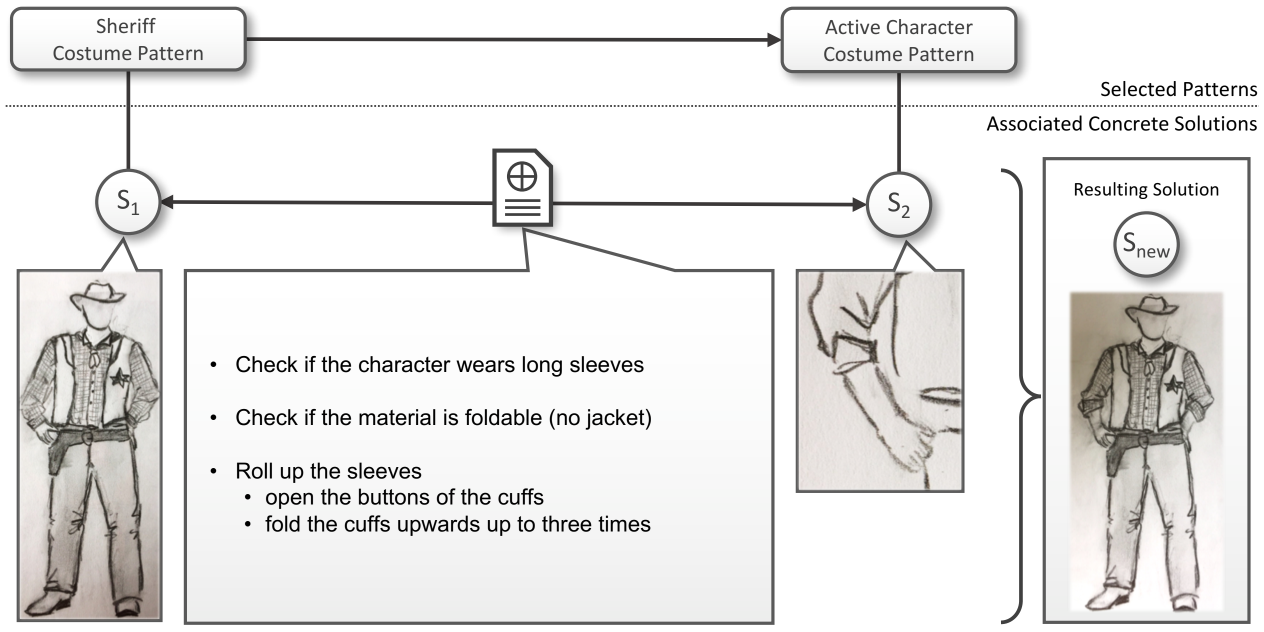

Sheriff Costume Pattern
Active Character Costume Pattern
Selected Patterns
Associated Concrete Solutions
S1
S2
Resulting Solution
Snew
• Check if the character wears long sleeves
• Check if the material is foldable (no jacket)
• Roll up the sleeves
• open the buttons of the cuffs
• fold the cuffs upwards up to three times

Figure 8. Concrete Solution Aggregation Descriptor documenting how to aggregate concrete solutions in the domain of costumes in films

solutions. So, we were able to navigate from patterns to concrete solutions and select them for reuse once a pattern has to be applied. To establish a Solution Language we declared a new property *can be aggregated with* using the Semantic MediaWiki extensions. Properties can be used to define arbitrary semantics, which can be added to wiki pages. The defined property accepts one parameter as a value, which we used to reference wiki pages that represent concrete solutions. This way, concrete solutions can be semantically linked with each other by adding the property into the markdown of their wiki pages and providing the link to the wiki page of the concrete solution, which the *can be aggregated with* semantics holds.

To annotate the link between two specific concrete solutions with information required for their aggregation, we added a CSAD as a separate wiki page containing a detailed description of the working steps required for aggregating them. Finally, we used the query functionality of the Semantic MediaWiki extensions to attach the CSAD to the semantic link between two concrete solutions. We utilized the parser function *#ask* of the Semantic MediaWiki extensions to query the two concrete solutions that are semantically linked with each other via the *can be aggregated with* property. This allowed us to also navigate from one concrete solution to other relevant concrete solutions based on the information of the semantic links, by also providing information about how to aggregate both concrete solutions to an overall one in a human-readable way.

Similarly, we also realized the second application scenario about cloud management via PatternPedia. The concrete solutions in this scenario, however, are represented by so-called TOSCA cloud service archives (CSAR). These archives are used to bundle service templates as well as all related deployment and implementation artifacts in a self-contained way and were linked with the wiki-pages representing them as concrete solutions. We further linked them with each other and added CSADs to describe how the several service templates can be combined as described in Section IV. Finally, we also added links to an instance of Eclipse Winery [41], which is a TOSCA-compliant modeling tool capable of merging different service templates automatically. Thus, Winery provides the automation of CSADs as conceptually described in the application scenario.

VI. Related Work

The term pattern language was introduced by Alexander et al. [2]. They use this term metaphorically to express that design patterns are typically not just isolated junks of knowledge, but are rather used and valuable in combination. At this, the metaphor implies that patterns are related to each other like words in sentences. While each word does only sparsely provide any information only the combination to whole sentences creates an overall statement. So, also patterns only unfold their generative power once they are applied in combination, while they are structured and organized into pattern languages in order to reveal their combinability to human readers.

Mullet [14] discusses how pattern catalogues in the field of human-computer interaction design can be enhanced to pattern languages to ease the application of patterns in combination. He reveals the qualities of pattern languages by discussing structuring elements in the form of different semantics of pattern relations. Further, the possibility to connect artifacts to patterns, such as detailed implementation documentation or also concrete implementations is identified as future research.

Zdun [24] formalizes pattern language in the form of pattern language grammars. Using this approach, he tackles the problem of selecting patterns from a pattern language. He reflects design decisions by annotating effects on quality attributes to a pattern language grammar. Relationships between patterns express semantics, e.g., that a pattern *requires* another pattern, a pattern is an *alternative* to another one, or that a pattern is a *variant* of another pattern. Thus, he describes concepts of pattern languages, which are transferred in this work to the level of concrete solutions and Solution Languages.

Reiners et al. [47] present a requirements catalogue to support the collaborative formulation of patterns. These requirements can be used as a basis to implement pattern repositories. While the requirements mainly address the authoring and structuring of pattern languages, they can also be used as a basis to detail the discussion about how to design and implement repositories to author Solution Languages. Pattern Repositories [7] [20] [48] have proven to support the authoring of patterns. They enable to navigate through pattern languages by linking patterns with each other. Some (c.f. [7] [20]) also enable to enrich links between patterns by semantics to further ease the navigation. Also, conceptual approaches exist that allow to connect a pattern repository with a solution repository, which can be the foundation to implement the concepts introduces in this work. These concepts and repository prototypes can be combined with our approach to develop sophisticated solution repositories.

Barzen and Leymann [21] present a general approach to support the identification and authoring of patterns based on concrete solutions. Their approach is based on research in the domain of costumes in films, where they formalize costume languages as pattern languages. Costumes are concrete solutions that solve specific design problems of costume designers. They enable to hark back to concrete solutions a pattern is evolved from by keeping them connected. They also introduce the terminus Solution Language as an ontology that describes types of clothes and their relations in the form of metadata, as well as instances of these types. This completely differs from the concept of a Solution Language as described in this work.

Fehling et al. [22] present a method for identifying, authoring and applying patterns. The method is decomposed into three phases, whereby, in the pattern application phase, they describe how abstract solutions of patterns can be refined towards concrete implementations. To reduce the efforts to spend for implementing patterns, they apply the concept of concrete solutions by means of code repositories that contain reference implementations of patterns. While our approach is designed and detailed for organizing concrete solutions the argumentation in their work is mainly driven by considerations about patterns and pattern languages. Thus, the method does not introduce how to systematically combine semantics and documentation in order to organize concrete solutions for reuse, which is the principal contribution of our work.

VII. Conclusion and Future Work

In this work, we presented the concept of Solution Languages that allows to structure and organize concrete solutions, which are implementations of patterns. We showed how Solution Languages can be created and how they can support the navigation through the solution space of pattern languages based on semantic links, all targeting to ease and guide pattern application. We further presented the concept of Concrete Solution Aggregation Descriptors, which allows to add arbitrary human-readable documentation to links between concrete solutions. Besides these concepts, we showed that concrete solutions addressing different aspects of a pattern language can be organized into Solution Languages and linked to their respective patterns. Finally, the generality of the presented concepts is shown via comprehensive application scenarios in the domains of cloud application architecture, cloud management, and costumes in films. While the first two show the plurality of concrete solutions and Solution Languages, especially, the application to the domain of costumes in films provide evidence that Solution Languages are not bound to the domain of IT but can also be used to ease and guide the application and combination of concrete solutions in general.

In future work, we are going to conduct research on how to analyze Solution Languages in order to derive new pattern candidates based on Concrete Solution Aggregation Descriptors annotated to links between concrete solutions, but also on the question if a Solution Language can indicate new patterns in a pattern language, for instance, in the case if links between concrete solutions are missing or if aggregation documentation cannot be clearly authored. We are also going to apply the concept of Solution Languages to domains besides cloud computing, e.g., to the emerging field of the Internet of Things. Finally, we are going to develop an algorithm in order to automate the selection of suitable concrete solution paths on the basis of a selected sequence of patterns and additional user constraints. This automation approach for selectiing concrete solutions will also make it possible to evaluate the concept of solution languages experimentally via runtime mesasurements.

Acknowledgment

This work is partially funded by the BMWi project SePiA.Pro (01MD16013F) as part of the Smart Service World.

References

[1] M. Falkenthal and F. Leymann, "Easing Pattern Application by Means of Solution Languages," in Proceedings of the 9th International Conferences on Pervasive Patterns and Applications. Xpert Publishing Services (XPS), 2017, pp. 58–64.

[2] C. Alexander, S. Ishikawa, and M. Silverstein, A pattern language: towns, buildings, construction. New York: Oxford University Press, 1977.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Abstraction and reuse of objectoriented design," in European Conference on Object-Oriented Programming, 1993, pp. 406–431.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and P. Stal, Pattern-oriented software architecture: A system of patterns, 1996, vol. 1.

[5] M. van Welie and G. C. van der Veer, "Pattern Languages in Interaction Design : Structure and Organization," in Human-Computer Interaction '03: IFIP TC13 International Conference on Human-Computer Interaction. IOS Press, 2003, pp. 527–534.

[6] G. Hohpe and B. Woolf, Enterprise Integration Patterns: Designing, Building, And Deploying Messaging Systems. Addison-Wesley, 2004.

[7] R. Reiners, "An Evolving Pattern Library for Collaborative Project Documentation," PhD Thesis, RWTH Aachen University, 2013.

[8] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, and J. Wettinger, "Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications," in Proceedings of the 9th International Conferences on Pervasive Patterns and Applications. Xpert Publishing Services (XPS), 2017, pp. 22–27.

[9] L. Reinurt, U. Breitenbücher, M. Falkenthal, F. Leymann, and A. Riegg, "Internet of things patterns," in Proceedings of the 21th European Conference on Pattern Languages of Programs, 2016.

[10] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, and A. Riegg, "Internet of Things Patterns for Devices," in Proceedings of the 9th International Conferences on Pervasive Patterns and Applications. Xpert Publishing Services (XPS), 2017, pp. 117–126.

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

[12] T. Furukawazono, I. Studies, S. Seshimo, I. Studies, D. Muramatsu, and T. Iba, "Survival Language : A Pattern Language for Surviving Earthquakes," in Proceedings of the 20th Conference on Pattern Languages of Programs. ACM, 2013, p. Article No. 30.

[13] J. Barzen et al., "The vision for MUSE4Music," Computer Science - Research and Development, vol. 22, no. 74, 2016, pp. 1–6.

[14] K. Mullet, "Structuring pattern languages to facilitate design. chi2002 patterns in practice: A workshop for ui designers," 2002. [Online]. Available: https://www.semanticscholar.org/paper/Structuring-Pattern-Languages-to-Facilitate-Design-Mullet/2fa5e4c25eea30687605115649191cd009a8f33c

[15] M. Falkenthal et al., "Leveraging pattern application via pattern refinement," in Proceedings of the International Conference on Pursuit of Pattern Languages for Societal Change. epubli GmbH, pp. 38–61.

[16] M. Falkenthal, J. Barzen, U. Breitenbuecher, C. Fehling, and F. Leymann, "From Pattern Languages to Solution Implementations," in Proceedings of the 6th International Conferences on Pervasive Patterns and Applications, 2014, pp. 12–21.

[17] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, and F. Leymann, "Efficient Pattern Application : Validating the Concept of Solution Implementations in Different Domains," International Journal On Advances in Software, vol. 7, no. 3&4, 2014, pp. 710–726.

[18] G. Meszaros and J. Doble, "A Pattern Language for Pattern Writing," in Pattern Languages of Program Design 3. Addison-Wesley, 1997, ch. A Pattern Language for Pattern Writing, pp. 529–574.

[19] C. Alexander, S. Ishikawa, and M. Silverstein, A Pattern Language: Towns, Buildings, Construction. Oxford University Press, Aug. 1977.

[20] C. Fehling, J. Barzen, M. Falkenthal, and F. Leymann, "PatternPedia Collaborative Pattern Identification and Authoring," in Pursuit of Pattern Languages for Societal Change - The Workshop 2014: Designing Lively Scenarios With the Pattern Approach of Christopher Alexander. epubli GmbH, 2015, pp. 252–284.

[21] J. Barzen and F. Leymann, "Costume Languages as Pattern Languages," in Pursuit of Pattern Languages for Societal Change - The Workshop 2014: Designing Lively Scenarios With the Pattern Approach of Christopher Alexander, 2015, pp. 88–117.

[22] C. Fehling, J. Barzen, U. Breitenbücher, and F. Leymann, "A Process for Pattern Identification, Authoring, and Application," in Proceedings of the 19th European Conference on Pattern Languages of Programs, 2015, article no. 4.

[23] U. Breitenbücher et al., "Policy-Aware Provisioning and Management of Cloud Applications," International Journal On Advances in Security, vol. 7, no. 1 & 2, 2014, pp. 15–36.

[24] U. Zdun, "Systematic pattern selection using pattern language grammars and design space analysis," Software: Practice and Experience, vol. 37, no. 9, jul 2007, pp. 983–1016.

[25] F. Buschmann, K. Henney, and D. C. Schmidt, Pattern-Oriented Software Architecture: On Patterns and Pattern Languages. Wiley & Sons, 2007, vol. 5.

[26] M. Falkenthal et al., "Pattern research in the digital humanities: how data mining techniques support the identification of costume patterns," Computer Science - Research and Development, vol. 22, no. 74, 2016.

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

[28] Amazon, "Amazon Web Services," 2017. [Online]. Available: http://aws.amazon.com/

[29] ——, "Amazon Cloud Formation," 2017. [Online]. Available: https://aws.amazon.com/cloudformation

[30] OASIS, Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0, Organization for the Advancement of Structured Information Standards (OASIS), 2013.

[31] Amazon, "Amazon Elastic Compute Cloud," 2017. [Online]. Available: https://aws.amazon.com/ec2

[32] Microsoft, "Microsoft Azure Virtual Machines," 2017. [Online]. Available: https://azure.microsoft.com/en-us/services/virtual-machines

[33] VMWare, Inc, "Vmware vsphere," 2017. [Online]. Available: https://www.vmware.com/products/vsphere.html

[34] OpenStack Project, "Openstack," 2017. [Online]. Available: https://www.openstack.org

[35] T. Binz, G. Breiter, F. Leymann, and T. Spatzier, "Portable Cloud Services Using TOSCA," IEEE Internet Computing, vol. 16, no. 03, May 2012, pp. 80–85.

[36] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "OpenTOSCA - A Runtime for TOSCA-based Cloud Applications," in Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013). Springer, Dec. 2013, pp. 692–695.

[37] Node.js Foundation, "Express," 2017. [Online]. Available: https://expressjs.com

[38] ——, "Node.js," 2017. [Online]. Available: https://nodejs.org/en

[39] Canonical Ltd, "Ubuntu," 2017. [Online]. Available: https://www.ubuntu.com

[40] redislabs, "Redis," 2017. [Online]. Available: https://redis.io

[41] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, "Winery – A Modeling Tool for TOSCA-based Cloud Applications," in Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013). Springer, Dec. 2013, pp. 700–704.

[42] M. Zimmermann, U. Breitenbücher, M. Falkenthal, F. Leymann, and K. Saatkamp, "Standards-based Function Shipping How to use TOSCA for Shipping and Executing Data Analytics Software in Remote Manufacturing Environments," in Proceedings of the 21st International Enterprise Distributed Object Computing Conference. IEEE, 2017, in press.

[43] P. Hirmer, U. Breitenbücher, T. Binz, and F. Leymann, "Automatic Topology Completion of TOSCA-based Cloud Applications," in Lecture Notes in Informatics - Informatik 2014, 2014, pp. 247–258.

[44] D. Schumm, J. Barzen, F. Leymann, and L. Ellrich, "A Pattern Language for Costumes in Films," in Proceedings of the 17th European Conference on Pattern Languages of Programs, 2012, article no. 7.

[45] Wikimedia Foundation, "MediaWiki," 2017. [Online]. Available: https://www.mediawiki.org/

[46] M. Krötzsch, "Semantic MediaWiki," 2017. [Online]. Available: https://www.semantic-mediawiki.org/

[47] R. Reiners, M. Falkenthal, D. Jugel, and A. Zimmermann, "Requirements for a Collaborative Formulation Process of Evolutionary Patterns," in Proceedings of the 18th European Conference on Pattern Languages of Programs, 2013, article no. 16.

[48] U. van Heesch, "Open Pattern Repository," 2009. [Online]. Available: http://www.cs.rug.nl/search/ArchPatn/OpenPatternRepository

All links were last accessed on December, 1st 2017.