Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

A Concept for Describing Concrete Solutions to Support their Automated Selection from Patterns

Ghareeb Falazi

Course of Study: M.Sc. Computer Science

Examiner: Prof. Dr. Dr. h. c. Frank Leymann

Supervisor: Michael Falkenthal, M.Sc.

Commenced: April 4, 2017

Completed: October 4, 2017

CR-Classification: D.2.2,D.2.11,D.2.13,D.3.1,D.3.4

Abstract

Patterns and pattern languages have been used to abstractly describe solutions of recurring problems in various domains. They are general enough to be applicable to many technologies and use-cases. However, with the generalness of patterns comes the problem of how to turn them into concrete solutions suitable for a specific environment. To this end, it was suggested to link patterns with sets of technology-specific concrete solutions that implement them, and make them accessible via repositories. Nonetheless, users still lack the support to choose the suitable combination of concrete solutions that implement a pre-selected sequence of patterns. In our work, we aim at solving this problem by elaborating a conceptual design to describe it along with the various entities related to it, which is then used as a basis for a 2-phase algorithm that automatically selects concrete solutions based on a given sequence of patterns and certain environment- and user-specific conditions. We finally evaluate our approach by studying the complexity of the algorithms and implementing a web-based prototype for them.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

List of Abbreviations

AMI Amazon Machine Image. 49

ANTLR4 ANother Tool for Language Recognition version 4. 77

AWS Amazon Web Services. 22

CMP Composite Management Planlet. 39

CS Concrete Solution. 29

CSAD Concrete Solution Aggregation Descriptor. 23

EBT Elastic Beanstalk. 49

EC2 Amazon Elastic Compute Cloud. 48

HTML HyperText Markup Language. 80

HTTP Hyper-Text Transfer Protocol. 50

JSON JavaScript Object Notation. 11

MUSE Muster Suchen und Erkennen-Pattern Search and Recognition. 82

PHP Hypertext Preprocessor. 22

REST Representational State Transfer. 50

S3 Amazon Simple Storage Service. 48

SESE Single-Entry, Single-Exit. 39

TOSCA Topology and Orchestration Specification for Cloud Applications. 36

UI User Interface. 83

URI Uniform Resource Identifier. 33

WAR Web Archive. 49

XQuery XML Query. 75

1 Introduction

General Background

Patterns are well-known concepts used to document the essence of proven solutions for reoccurring problems so that they can be adapted and reused in similar situations. The idea to abstract concrete solutions and formulate reusable patterns out of them was first introduced by C. Alexander et al. [Ale79] [AIS+77] for the field of building architecture, but soon were adopted by many other fields as well such as computer science with its various sub-domains, educational styles, costumes, music, and many others.

Patterns target specific problems in a specific context, and due to the fact that problems are usually interrelated in a real-world setup, patterns are also connected together, forming what is called a Pattern Language which includes, alongside patterns, semantic links that help to navigate from one pattern to other patterns that solve related problems.

Given a problem and a pattern language for the same domain, one would try to solve it by formulating a sequence of patterns each of which solves a specific aspect of it. However, an issue arises when this sequence of patterns is to be implemented as a concrete solution, because the process of generating patterns, in the first place, strips them from any technology-specific details so that patterns become more generic and widely applicable, but this forces the user to re-implement concrete solutions out of patterns which entails a lot of effort.

Falkenthal et al. [FBB+14] [FBB+16] [FL17] proposed that patterns should be linked with the concrete solutions that they originated from, that is, the process of abstracting concrete solutions of recurring problems to come up with patterns should maintain a link between the generated patterns and the original concrete solutions. Furthermore, new concrete solutions should be connected to existing patterns especially if they represent a new technology in the domain that was not present when the patterns were authored. Concrete solutions that are linked to patterns can be stored with them in the same repository, and with this, users are presented with concrete solutions when they select patterns, and thus moving from the abstract layer of patterns to the concrete layer of solutions is made easier.

However, much like patterns in a pattern language, concrete solutions also have certain kinds of relationships among themselves. For instance, some concrete solutions can be

aggregated together to form a composite solution, while others cannot. This means that concrete solutions are not isolated from each other but they rather form a network connected through semantic links just like patterns do. Falkenthal et al. [FL17] call this network a *Solution Language*.

Motivation

Solution languages provide navigational capabilities at the level of concrete solutions which helps users to navigate from one concrete solution to the next. However, if we look at the whole problem, we still notice a gap between pattern languages and solution languages; both provide navigational capabilities at their own levels, and although individual concrete solutions are connected to the patterns they implement, a user with a sequence of patterns at hand still lacks the support needed to map it to a sequence of concrete solutions, because each pattern is potentially connected to many concrete solutions resulting, e.g., from multiple technologies, multiple versions of the same technology or multiple trade-offs between cost and quality. Furthermore, moving from one concrete solution to the next, we need to choose a single semantic link out of many while making sure that an aggregation descriptor exists to help combining the two concrete solutions.

If the size of the solution language is big enough, performing this mapping task manually becomes infeasible. Thus, projecting a pre-selected sequence of patterns from a pattern language onto a solution language should be done automatically or semi-automatically while still allowing the user to control the process. This problem has not been thoroughly studied yet, and the work we present in this thesis aims at finding a solution to it.

Approach Outline

To this end, we first try to provide a concept that describes the mapping problem by defining a meta-model for the various entities involved. The cornerstone of this meta-model is the concrete solution itself which primarily includes requirements and capabilities that are used to represent the semantic links of the solution language dynamically. The design also includes definitions for aggregators to combine concrete solutions, initial properties to describe the environment, and user queries that affect the selection process as well as definitions for other necessary concepts.

Next, we present a 2-phased algorithm that utilizes the meta-model to map a given sequence of patterns into a sequence of concrete solutions while making sure requirements are met and the user query is fulfilled. The first phase uses a variant of depth-first traversal to generate candidate sequences of concrete solutions which are then checked for validity in

the second phase. This phase filters-out sequences that have unmet requirements or that do not fulfill the user query, which results only in sequences which are valid.

The algorithm is first analyzed statically by studying the worst-case time complexity of both of its phases. Results show that the worst-case is prohibitively too complex when the number of concrete solution in a language is too high and the given pattern sequence is too long. Nonetheless, we expect the performance to be acceptable for most real-world scenarios.

Finally, we further prove the feasibility of the algorithm and the conceptual design by creating a web-based prototypical implementation, and applying multiple real-world use-cases to it. Tests show that using our approach, one can define a wide range of selection criteria, even complex ones, but that it needs heavy annotation efforts to enrich concrete solutions with sufficient meta-data.

Document Structure

The rest of the document is structured as follows: In Chapter 2, we present the fundamentals needed to understand the background of the mapping problem, then we motivate our research goals and discuss the related work. In Chapter 3 we present a set of requirements that we try to fulfill while designing the meta-model, the mapping algorithms, and the prototypical implementation. Moreover, we discuss the set of assumption we made while working on this thesis and making design choices. In Chapter 4 we start with the body of our work by introducing the conceptual design that describes the mapping problem, and in Chapter 5 we provide the algorithms that solve it. Moreover, we analyze the worst-case time complexity of the presented algorithms. Chapter 6 provides details about the prototypical implementation developed to test the feasibility of the conceptual design and the algorithms. Finally, we list the findings and conclusions we have made in Chapter 7, and we present our vision of the future work relevant for this topic.

2 Fundamentals and Related Work

In this chapter we introduce the fundamentals of patterns and pattern languages. Furthermore, we show the importance of mapping sequences of patterns to paths of concrete solutions that implement them and how it is a single step in a larger pipeline. Finally, we present an overview of the related-work.

2.1 Patterns and Pattern Languages

Patterns are human readable artifacts that document proven solutions to recurring problems. After their successful introduction by Christoph Alexander in 1977 [AIS+77] [Ale79] for the field of building architecture, patterns were also create for many other fields, such as enterprise application architecture [Fow02], enterprise application integration [HW04], cloud application architecture [FLR+13], real-time system architecture [Dou03], reliable telecommunication systems [ACG+96], pedagogy [BEM+12], and costumes [BL15].

A pattern abstractly encapsulates the knowledge related to a single solution for some recurring problem. It is usually formulated into a well-structured document [MD97] with some parts describing various aspects of the problem that the pattern solves, and other parts describing an overview of the suggested solution which is suitable for the problem in a specific context. This document often includes justifications for the proposed solution based on the failure or shortcomings of alternative approaches, as well as some examples of how to apply the generic knowledge of the pattern in real-world scenarios.

Even though single patterns are useful when solving isolated problems, their real power is achieved when they are combined together to form a pattern language. In a pattern language, each pattern is connected to other patterns that are used in the context of each other. This kind of formation allows us to solve a complex problem by consecutively selecting patterns that solve smaller parts of it. This makes a set of interrelated patterns more beneficial than the sum of benefits of the constituting patterns considered individually [Zdu07].

Patterns and pattern languages are usually formulated as a result of an iterative process that starts from real-world scenarios[FLR+14]. This process aims at classifying problems and their solutions and extracting the essence of the two making patterns and pattern languages applicable beyond the specific use-cases that contributed to their creation. However, the

abstraction performed to create them strips away implementation details that are needed when the user selects a set of patterns to solve a complex problem and then wants to build a concrete solution based on them. In this case the user might need to re-implement technology-specific details that were already known at the pattern creation time. To face this issue Falkenthal et al. [FBB+14] propose to maintain references from patterns to the concrete solutions that participated in the authoring process as well as additional concrete solutions added later on. We further discuss concrete solutions and their relation to patterns in the following section.

2.2 The Space of Concrete Solutions

A concrete solution is a domain-specific artifact or a set of inter-related artifacts that implement one or more patterns from a pattern-language. For example, a concrete solution could be a Java class that realizes the *Singleton* design pattern [JGHV95], or a snippet from an Amazon Web Services (AWS) CloudFormation Template [Ama17a] that realizes the *Elastic Load Balancer* cloud-computing pattern [FBFL15], or even the costume of a character that realizes the super-hero pattern [BL15]. Although a concrete solution could be a set of artifacts such as a collection of Java classes and interfaces that implement the *Composite* design pattern [JGHV95], and for the sake of simplicity, we will consider that the whole set of artifacts is treated as a single concrete solution.

The relationship between patterns and concrete solutions is usually not simply one-to-one; a single pattern could have many implementing concrete solutions each of which specializes in a specific field of application. For example, an *Elastic Load Balancer* cloud-computing pattern could be realized in many cloud platforms, such as AWS [1], Windows Azure [2], or Google Cloud Platform [3], and for each one of these platforms, the implementing artifact could have a different nature or format. Furthermore, other patterns of the cloud-computing domain can also have many implementing concrete solutions. Thus, in general, we have a bigger space of concrete solutions that are associated to the space of a pattern-language.

Falkenthal et al.[FBB+14] show that these concrete solutions can be stored in a solution repository that is linked to the pattern language. However, using such a repository only allows to navigate from a pattern to its potential implementations, but not to navigate through the set of concrete solutions as a navigation structure is missing at this level. We can see this further depicted in Figure 2.1.

Next, we explain why having links at the concrete solution level is important, and how this issue is addressed.

[1] https://aws.amazon.com/
[2] https://azure.microsoft.com/
[3] https://cloud.google.com/

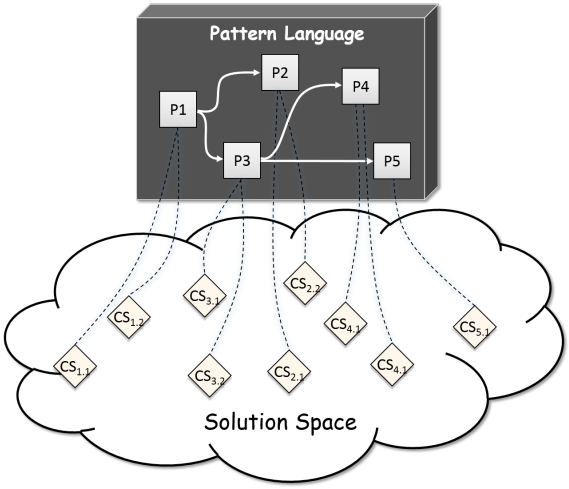


Figure 2.1: Missing navigation support through the space of concrete solutions connected to a pattern language [FL17]

2.3 Solution Languages

The navigational capabilities at the level of concrete solutions are needed because when using a pattern language to solve a problem at a conceptual level, patterns are usually selected as a sequence rather than isolated. This sequence is built with the help of semantic links that exist between patterns and constitute, along with the patterns themselves, the pattern language at hand. This pattern sequence has to be then mapped onto the solution space to get the collection of concrete solutions that can concretely solve the targeted problem. However, choosing those concrete solutions does not only depend on the patterns they implement (the vertical links in Figure 2.1) but also on their nature and the specific context of the problem. For example, a concrete solution that uses Java code to implement the *Singleton* design pattern cannot be combined with a concrete solution that uses C# code to implement the *Builder* design pattern [JGHV95] in order to form a final solution of the problem, and both cannot be used when the user needs a solution which is written in Hypertext Preprocessor (PHP). Thus, navigable semantic links between concrete solutions that reside in the same solution space are needed. Falkenthal et al. [FL17] have identified this problem and introduced the concept of Solution Languages to address it.

A solution language tries to organize concrete solutions in a way similar to the patterns of a pattern language. It does that by adding semantic links between concrete solutions that help in traversing them by letting the user know, for example, if specific concrete solutions *can be aggregated* together or not, or that a set of concrete solutions are *exchangeable variants* or *exclusive options*. Furthermore, a solution language supports documenting how to aggregate concrete solutions that share a semantic link of the type *can be aggregated* between them by introducing the concept of Concrete Solution Aggregation Descriptor (CSAD), which is a domain-specific means to describe how two concrete solutions can be

aggregated together. Such a CSAD could be an automated script that modifies the code of concrete solutions allowing them to interact in a meaningful way, or even a human-readable description of how to combine two pieces of cloths that represent two concrete solutions in the film-clothing domain.

Next, we show how these semantic links and CSADs can be utilized in helping the user pick the right collection of concrete solutions for their specific problem.

2.4 Concrete Solution Paths

Finding a conceptual solution to a problem by building a path of patterns that belong to a pattern language is often studied in research. Specifically, there exists many approaches, such as [PCW05] and [Zdu07], to facilitate navigation from one pattern to the next based on the specific situation, so starting from a pattern one can, with the help of these approaches, build the whole desired sequence of patterns. However, research lacks a thorough study of how to map the resulting pattern sequence [Zdu07], which is also known as the *solution path* [FBB+14][4], to the corresponding solution language. Such a mapping, which can make use of the aforementioned semantic links and the CSADs that annotate them, is a very important step towards having the complete pattern-based problem solving pipeline (which is demonstrated in Figure 2.2).

This pipeline consists of 5 major phases:

Problem Analysis in which the problem at hand is studied and the criteria that will be used for the selection of patterns is derived.

Pattern Selection in which we apply a pattern selection algorithm and get a solution path consisting of patterns that solves the problem.

Mapping to a Solution Language in which we apply the work we introduce in this thesis to map the solution path to a concrete solution path that resides in a solution language.

Aggregation in which we aggregate the various concrete solutions constituting a concrete solution path into a single composite concrete solution.

Refinement in which users manually refine the composite concrete solution to make it fit exactly their needs. This steps results in the final ready-to-use solution that solves the original problem.

[4]The term "solution" in a "solution path" is intended to mean an abstract solution to a given problem, much like the "solution" section of a pattern document.

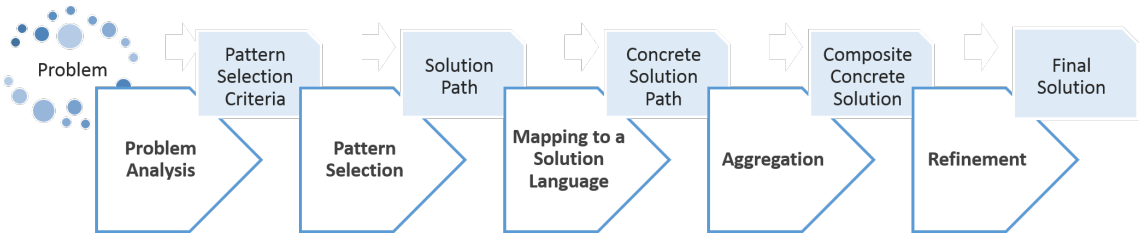


Figure 2.2: The complete pattern-based problem solving pipeline.

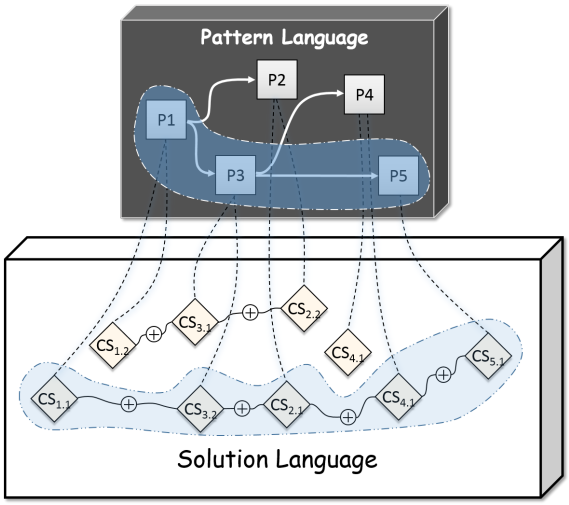


Figure 2.3: Solution path from a pattern language projected onto a solution language [FL17]

In their work [FL17], Falkenthal et al. give an example of the mapping step similar to what is shown in Figure 2.3. The figure depicts a solution path consisting of the patterns P2, P4, and P5, which was built in a previous step, and the collection of concrete steps (S4, S7, and S9) that is mapped to it.

The process with which we can arrive at this result, however, is abstract and lacks concrete details; it is intended to show what navigational capabilities a solution language can provide, but not how exactly we can use these capabilities. Furthermore, if one would build a system that manages a solution language and provides a mapping capability from the pattern language, then the nature, the structure and the format of semantic links have to be concretely defined. Moreover, it has to be exactly defined what a valid mapping from a solution path to a concrete solution path is.

The work we present here is a continuation of the work conducted by Falkenthal et al. [FBB+14] [FBB+16] [FL17] and aims at enhancing how we address the aforementioned issues. To this end, we first introduce a concept that is able to describe concrete solutions, and is helpful in identifying their compatibility by using the their requirements and capabilities as a realization of the concept of semantic links.

Second, we define algorithms that are able of mapping a pre-selected sequence of patterns onto a solution language to produce a concrete solution path consisting of concrete solutions as well as the CSADs that aggregate them. Our work does not go further into the step of actually using CSADs to aggregate the concrete solutions. This issue has its own complications and we consider it suitable for a separate research.

Finally, we provide an implementation of a solution repository that is able of hosting a solution language and can prove the feasibility of the mapping algorithms we formulated.

2.5 Related Work

Falkenthal et al. [FBB+14] discuss the problem of creating solutions based on patterns. They argue that this problem results from the fact that patterns describe solutions only at a conceptual level, so they suggest that a pattern should be linked with both the original concrete solutions that were used to author it in the first place, as well as new implementations of it. To this end, they introduce the concept of Solution Implementations that allow users using existing patterns to reuse the connected implementation artifacts. This concept facilitates the composition of the implementing artifacts of related patterns too. Furthermore, they introduce Selection Criteria as a means to select the appropriate concrete solution implementing a given pattern. However, these criteria only help in vertical selection of concrete solutions (i.e., from patterns to implementing concrete solutions) but not horizontally (from one concrete solution to the next). Moreover, they only define the rules that allow two concrete solutions to be combined but not how to select a suitable set of concrete solutions matching a given sequence of patterns.

In [FL17], Falkenthal et al. introduce the concept of solution languages as a means to allow users to navigate from one concrete solution to the next, thus creating capabilities similar to those of a pattern language. Using their approach, navigation occurs through semantic links. However, we argue in Section 4.12 that hard-coded semantic links do not scale well when the number of concrete solutions in a solution language is high enough. Furthermore, their approach does not show how the mapping problem could be resolved.

Pattern refinement was studied by Falkenthal et al. in [FBB+16]. They point out that although the abstraction of patterns makes them more reusable, it creates a gap between the abstract knowledge of patterns and the concrete steps needed to solve a specific problem. Thus, they introduce pattern refinement as a new kind of pattern relationships. It describes a pattern that refines another pattern towards a concrete solution. Patterns and pattern languages become "layered" based on their approach, and instead of getting a solution path solving a given problem abstractly, we get a solution graph. Nevertheless, their approach does not discuss concrete solutions but rather focuses on patterns with various levels of abstraction.

Fehling et al. [FBFL15] introduce PatternPedia, a wiki-based repository for pattern authoring and management. PatternPedia supports a pattern research methodology that covers many steps of the life-cycle of patterns. What we find interesting for our work in this methodology is how it associates concrete solutions with patterns that were authored based on them, and how it involves the creation or reuse of concrete solutions after a user selects a set of patterns to solve a given problem. To support this methodology, PatternPedia facilitates not only storing and managing patterns, but also concrete solutions, and the semantic links that it uses to connects patterns together, can also be used to connect concrete solutions together, as well as writing semantic queries against them. However, as we mentioned earlier, semantic links are not scalable when the number of concrete solutions is too high. Furthermore, PatternPedia does not support automatic selection of concrete solutions based on a given solution path.

Zdun [Zdu07] addresses the issue of pattern selection by first identifying the set of quality goals relevant for a given pattern language and recognizing the effects of patterns on these goals by analyzing their "forces" and "consequences" sections. Afterwards, Zdun's approach creates a grammar for the given pattern language based on the semantic links between its constituent patterns and annotates it with the various effects on quality goals identified earlier. Sequences of patterns, which Zdun identifies as a consecutive pattern-selection decisions, are then similar to sentences of a language which can be validated against the grammar of the same language. These sequences can be automatically created based on a given production rule. Although this approach is designed for pattern languages, the concept of automatic selection based on a given rule influenced our work on concrete solution selection.

Porter et al. [PCW05] argue that the selection of patterns is a question of temporal ordering of their application rather than a spatial one resulting from the pattern language structure. They conclude from this that a sequence is the best structure to express the selection of patterns for the purpose of composing them. They even argue that a pattern language is not an adequate description means by itself as it is not clear about in what order patterns should be applied, and thus, they state that "patterns must still appear on the page in some linear order. One of the best orders in which to present the patterns is the main sequence order, that is, the order that would occur if all the patterns in the language were applied." We have adopted their view, and considered that the input for the mapping problem that we try to solve is a sequence of patterns.

In [Wet17] Wettinger discusses supporting continuous delivery by automatically gathering concrete solutions. He defines a pipeline for this process that includes crawling existing solution repositories and grabbing concrete solution artifacts from them, and then annotating these artifacts and inter-linking them to form a knowledge-base of concrete solutions that can be used in the continuous delivery pipeline by identifying environment-specific requirements and applying them to select suitable concrete solutions. Wettinger's approach uses requirements and capabilities for the purposes of interlinking concrete solutions

within the knowledge base “without an explosion of direct links between them”. Moreover, Wettinger suggests to use label taxonomies in order to make the matching process of requirements and capabilities more declarative. The meta-model described by Wettinger was a starting point for the design of the meta-model we introduce in this thesis. However, Wettinger’s approach does not take patterns or their relationship to concrete solutions into consideration which a main difference from our approach.

2.6 Summary

In this chapter we have shown the fundamentals behind the work we have done by demonstrating the place where previous research has reached and outlining the final goal of having a complete pattern-based problem solving pipeline and indicating where our research and the previous research fit within it.

3 Requirements and Assumptions

In this chapter we present the set of requirements that we have considered while designing the conceptual meta-model that describes the problem and its related entities, as well as when defining the algorithms that solve the mapping problem and when implementing a prototype that validated our approach. Furthermore, we present the set of assumptions we have made while doing the aforementioned tasks.

3.1 Requirements

These requirements can be seen as the goals we try to achieve, rather than the software-engineering sense of requirements. Some of them were identified after discussions with the supervisor of the thesis based on his experience and research in the field.

3.1.1 Requirements on Conceptual Design

These requirements apply to the meta-model of concrete solutions and the semantic links between them.

We can see an overview of these requirements in Table 3.1, and in the following we are going to discuss them in more detail:

#	Requirement
R1	A Concrete Solution (CS) should be able to uniquely identify its associated artifacts
R2	A CS should be able to handle an arbitrary format of the artifact
R3	Semantic links should be expressive

Table 3.1: An Overview of the Requirements at the Conceptual Design Level

R1- A Concrete Solution Should Be Able to Uniquely Identify Its Associated Artifacts

A concrete solution and the artifact(s) with which it is associated can be decoupled, yet a concrete solution should have a mechanism to locate and allow access to its artifact(s). This is important because eventually the artifacts will be used to generate the final solution, and concrete solutions are just means to associate meta-data to the artifacts without having to modify them.

R2- A Concrete Solution Should Be Able to Handle an Arbitrary Format of the Artifact

Pattern languages and eventually solution languages cover a broad range of domains (and sub-domains) like architecture, computer science, teaching, costumes, visual designing, etc. And for each domain, and sometimes within the same domain, artifact implementing patterns can have a broad variety of types and formats, and if our approach is to be applicable in many fields, then the meta-model of a concrete solution should be able handle this heterogeneity.

R3- Semantic Links Should Be Expressive

Semantic links should be realized in a way that allows covering many potential cases; they should be able to express complex relationships between concrete solutions that can go beyond *can be aggregated with*, *variants*, and *exclusive options*. For example, considering that the process of building a repository for a solution language can be gradual and concrete solutions can be added to or removed from it at any time [WBFL17], a concrete solution should be able to connect to other concrete solutions that are not yet included in the solution language, so a link between them should be flexible enough to support that.

3.1.2 Requirements on Algorithms

These requirements apply to the algorithms responsible for mapping a preexisting solution path to a concrete solution path derived from the solution language. In Table 3.2 we can find an overview of these requirement. We further describe them in the following:

#	Requirement
R4	The algorithm should allow the user to influence its result
R5	The algorithm should be applicable to a wide range of solution languages
R6	The algorithm should not assume any order of the concrete solutions

Table 3.2: An Overview of the Requirements at the Algorithmic Level

R4- The Algorithm Should Allow the User to Influence Its Result

The algorithm should accept inputs beyond the solution path. For example, the user could require that a certain alternative of a concrete solution should be included in the final concrete solution path or they could have any other condition that cannot be expressed within the meta-data of the concrete solutions or the semantic links between them, and the algorithm should be able to handle such cases.

R5- The Algorithm Should Be Applicable to a Wide Range of Solution Languages

The algorithm should be generally applicable to all Solution Languages that fulfill the conceptual model it supports. It should not depend on the specificities of some of the Solution Languages.

R6- The Algorithm Should not Assume Any Order of the Concrete Solutions

The algorithm should not assume that the application of concrete solutions within the result will be ordered. Although in some domains it could be the case that the application of concrete solutions is ordered, which can make a specialized algorithm more efficient, the algorithm we are trying to formulate aims at covering a wide range of domains, thus such an assumption is not accepted.

3.1.3 Requirements on Implementation

These requirements apply to the design and the implementation of the solution repository that will be used to prove the feasibility of the concept and algorithms developed in this work. They were chosen while keeping in mind that this repository might be used within a larger implementation that might cover the whole pipeline rather than the mapping step only. In Table 3.3 you can find an overview of these requirements. We further describe them in the following:

#	Requirement
R7	The implementation should be realized using the serverless architectural style

Table 3.3: An Overview of the Requirements at the Implementation Level

R7- The Implementation Should Be Realized Using the Serverless Architectural Style

Serverless architecture [Rob16] can significantly reduces operational cost and complexity by relying on third-party services rather than managing one's own server to host the application. Experience shows that operational complexity and cost can badly affect the continuation of projects manged by certain classes of users, such as researchers, as they have limited time to spare on management tasks.

Throughout this thesis, we will refer to the requirements mentioned here, and show how we try to fulfill them.

3.2 Assumptions

To make our approach applicable to as many domains of pattern languages as possible we tried to minimize the number of assumptions we make. Nevertheless, some assumptions still had to be made, and in this section we summarize them and discuss why we had to make them:

1. The input patterns are formulated as a sequence. Zdun [Zdu07] describes the process of pattern selection as a sequence of events. This makes the result a linear sequence of patterns. Furthermore, Porter et al. [PCW05] argue that the structure of a pattern languages only imposes weak order of patterns, and that selecting patterns is rather a temporal ordering of them which results in a sequence of patterns. In our work, we adopt this point-of-view and in Chapter 7 we note that it can be further generalized.

2. We assume that all considered patterns can be associated with artifacts that represent real-world concrete solutions. We base this on the fact that the processes of deriving a pattern, in the first place, entails abstracting existing concrete solutions to reach to their essence [FLR+14]. So patterns are originally derived from concrete solutions and thus, they can be associated to concrete solutions.

3. A concrete solution implements exactly one pattern. We recognize that some concrete artifacts only partially implement a pattern, we show how we address this in Section 4.1. Furthermore, we also recognize that some concrete solutions can implement more than one pattern. We discuss this in Chapter 7.

4. A concrete solution can be annotated with requirement- and capability-related meta-data. These annotations are used to realize the semantic links between concrete solutions so that they can form a solution language similar to a pattern language. Here, we adopt the approach taken by Wettinger et al. [WBFL17] as described earlier. We further discuss this issue in Chapter 7.

5. An aggregator or a CSAD is responsible for aggregating exactly two concrete solutions. This greatly makes the meta-model and the algorithms easier. Here, we make an implicit assumption that an aggregator is an operator in the space of concrete solutions and note that this issue is not yet fully studied (see Chapter 7).

6. An artifact representing a (part of a) concrete solution or an aggregator can be uniquely addressed. We assume that artifacts are real-world objects, such as script files, documents, blue-print designs, pieces of clothing, music notes, etc... and that any real-world object can be uniquely addressed. Such artifacts can be addressed using, e.g., Uniform Resource Identifiers (URIs).

3.3 Summary

In this section, we have listed the requirements that our work tries to fulfill at three levels: the conceptual design, the algorithms, and the implementation and described them in details. Furthermore, we have shown the set of assumptions we have made while working on theses three levels and tried to argue why we had to make those assumptions and why they make sense.

In the following chapter we present the concept we have designed to describe concrete solutions and our realization of the semantic links between them.

4 Conceptual Design

In this chapter we introduce a conceptual design that describes concrete solutions and the semantic links between them (see Section 2.3 on page 23). To this end, we try to come up with an abstract design that can handle most solution languages. This design is then used in the next chapter by an automatic algorithm that maps an input solution path which resides in a pattern language to a concrete solution path at the level of the solution language. As this algorithm uses an abstract data model (resulting from the abstract design) it too has abstract aspects. Then, we move to a suggested concrete design and algorithm that can be realized afterwards with a concrete implementation with a minor loss of generality.

The conceptual design we introduce in the following sections focuses on the task of mapping a solution path to a concrete solution path. A real-world repository that stores concrete solutions could have extra tasks other than this [WBFL17], and thus might also focus on further aspects that are not covered here.

4.1 Concrete Solutions

First of all, we make a distinction between a concrete solution, and the artifacts that implement a pattern. As we mentioned in Section 2.2 on page 22, a pattern is realized, for a specific application, with one or more artifacts which can differ in their nature from one domain to another. This format heterogeneity makes it difficult to use artifacts directly in a design that is supposed to be able to handle solution languages from many domains. Furthermore, the selection of implementations of patterns during the process of mapping a solution path to a concrete solution path only makes use of the semantics of the artifacts, not the artifacts themselves. Thus, we provide a separation layer between artifacts and our conceptual design by using concrete solutions that only reference the actual artifacts[1].

Thus, a concrete solution serves two purposes: (i) It provides access to the artifacts that implement, as a group, the associated pattern. (ii) It describes the collective semantics of the associated artifacts by listing their capabilities and requirements

Definition 4.1.1 (Concrete Solution)
A concrete solution ***cs*** *is a tuple (id, A_ID, Cap, Req) in which:*

[1] Here, we assume that an artifact can be uniquely identified.

id *is an identifier that is able of uniquely identifying the concrete solution.*

A_ID *is a set of artifact identifiers. These artifacts collectively provide an implementation of a pattern* ***p*** *from a pattern language.*

Cap *is a set of capabilities.*

Req *is a set of requirements.*

For convenience, when the artifacts identified by the set A implement a pattern p we say that the concrete solution cs also implements p, and we denote that by **cs::p**.

Next, we describe the concepts of requirements and capabilities that were used in the previous definition.

4.2 Requirements and Capabilities

In this context, requirements and capabilities are means to express the semantics of concrete solutions in terms of what they can do or achieve (i.e., *capabilities*), or what conditions they require to exist in order for them to be able to fulfill their roles (i.e., *requirements*). They could be derived from the individual artifacts that constitute a concrete solution or from these artifacts as a whole.

For example, a concrete solution that implements the *Elastic Load Balancer* cloud-computing pattern using a CloudFormation template snippet as its artifact could have a requirement that states "access to AWS is needed", and can have a capability that states "Implements Elastic Load Balancer".

Definition 4.2.1 (Requirement)
A requirement ***r*** *of a concrete solution* ***cs*** *is a conditional statement that has to be fulfilled in order for* ***cs*** *to function properly and thus be included in a concrete solution path.*

Definition 4.2.2 (Capability)
A capability ***c*** *of a concrete solution* ***cs*** *is any functional or non-functional property of* ***cs*** *that can affect the correct operation of other concrete solution, or is "interesting" to the user.*

The previous definitions are rather abstract but they suffice for the purpose of providing the meta-model for the mapping algorithm. Later in this thesis, we present more specific definitions that are more suitable for a concrete implementation.

The notion of requirements and capabilities in association with solutions is not new. It is found, for example, in the Topology and Orchestration Specification for Cloud Applications (TOSCA) standard [OAS13] to describe the dependencies of service components, or as part of the application environment meta-model proposed by Wettinger et al.[WBFL17] which

describes a solution repository that facilitates the continuous delivery of DevOps solutions. Furthermore, Falkenthal et al. [FBB+14] introduced a similar notion [2] to detect whether two concrete solutions in a solution language can be aggregated together or not.

4.3 Initial Properties

When designing a concrete solution repository, one would try to cover as many use cases as possible. For example, if we are to design a solution repository for the domain of cloud application architecture, we would include various concrete solutions that are compatible with the wide range of cloud providers, such as AWS, Microsoft Azure, IBM Bluemix [3], etc. This makes some concrete solutions require (in the form of requirements Section 4.2) specific context-dependent conditions to be true in order for them to function properly.

For example, a concrete solution that has an artifact in the form of an AWS CloudFormation Template snippet would have a requirement that states "access to AWS is needed". Other concrete solutions in the same solution language cannot fulfill this requirement as it depends solely on the context in which the solution language repository is used. Moreover, when a user, with a specific problem at hand, wants to utilize such a repository by, e.g., using the mapping functionality we introduce here, they would know exactly which of these conditions are true. The user, then, would want to provide concrete solutions with this information to fulfill their corresponding requirements. To this end we introduce ***Initial Properties***.

Definition 4.3.1 (Initial Property)
An initial property is a statement provided by the user to describe the context in which the solution language is being used. It is relevant to a specific use-case and thus is not stored within the solution language repository, but rather considered as one of the inputs of the mapping algorithm.

It is worth mentioning that initial properties are similar in nature to capabilities (see Section 4.2) in that they both can be addressed by requirements. However, a capability is associated to a specific concrete solution, whereas an initial property is associated to the context of the specific problem at hand.

[2] They used the terms *preconditions* and *postconditions*
[3] https://www.ibm.com/cloud-computing/bluemix/

4.4 Aggregators

The purpose of an aggregator is documenting how two concrete solutions can be aggregated together as a step in building a comprehensive composite concrete solution that solves the input problem.

The nature of an aggregator (or a Concrete Solution Aggregation Descriptor (CSAD) [FL17]) is highly domain-dependent as it is related to the nature of the artifacts that are associated to the concrete solutions of the domain. For example, when the artifacts represent programming code snippets, an aggregator could, e.g., be an automated program that changes the code files in order for them to see each other and work together while also adding any necessary links to external libraries. In other domains, such as the film-clothing domain, an aggregator could be a manual description of the steps that should be performed to make two pieces of cloths that represent two different concrete solutions match together properly. In any case, an aggregator is also an artifact, and here we assume that it can be uniquely identified.

The possible heterogeneity of aggregators in various domains makes it especially difficult to include them directly in a domain-agnostic algorithm. This made us follow an approach similar to the one we have taken in Section 4.1 on page 35 which dealt with concrete solutions, by differentiating between the artifact and its meta-data. Here, the meta-data that we care about is an identifier that uniquely identifies the artifact, as well as the pairs of concrete solutions that the aggregator can handle. Other pieces of meta-data could be necessary when performing the actual aggregation, especially those that describe the nature of the resulting composite concrete solution which would be helpful for further aggregations. However, for the purpose of the mapping step, the aforementioned meta-data suffices.

Definition 4.4.1 (Aggregator)
*Let $\mathcal{CS}$ be the set of all possible concrete solutions, then an aggregator **a** is a tuple (id, artifact_id, f) in which:*

id *is an identifier that is capable of uniquely identifying the aggregator.*

artifact_id *is an identifier that is capable of uniquely identifying the artifact that represents the core functionality of the aggregator.*

f *is a boolean function defined as follows:*

$$f : \mathcal{CS} \times \mathcal{CS} \rightarrow \{true, false\}$$

*It accepts two concrete solutions **cs1** and **cs2** as inputs, and returns true if the artifact defined by **artifact_id** is capable of aggregating **cs1** and **cs2**; otherwise, it returns false.*

Definition 4.4.2 (Compatibility of Concrete Solutions)
Let $\mathcal{A}$ be the set of all aggregators, then we say that two concrete solutions $cs1, cs2 \in \mathcal{CS}$ are compatible if and only if

$$\exists\ a = (id_a,\ artifact\_id_a,\ f_a) \in \mathcal{A}\ such\ that\ f_a(cs1,\ cs2) = true$$

*That is, there exists an aggregator **a** that is capable of aggregating **cs1** and **cs2**. For convenience, we denote this as **a(cs1, cs2)***

The compatibility of two concrete solutions is a necessary condition for them to be selected consecutively within any concrete solution path. However, the existence of an aggregator that can aggregate two concrete solutions does not necessarily mean that they are semantically compatible. The reason behind this is that the nature of concrete solutions, in some domains, makes it relatively simple for them to be aggregated.

An example of this comes from the domain of cloud application management [FLR+14] [FLR+13]. For this domain, it was proposed by Breitenbücher et al. [BBKL13] that concrete solutions should be in the form of management workflows that are called *Management Planlets* which 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."[FBB+14].

One interesting aspect of *Management Planlets* is that they are modeled as a Single-Entry, Single-Exit (SESE) workflow fragments making them straightforward to combine. In fact, a single aggregator is enough for the whole solution language. Such an aggregator would just put the two input fragments in sequence one after the other forming a larger workflow fragment that Breitenbücher et al. [BBKL13] call a Composite Management Planlet (CMP). This, however, does not mean that any two arbitrary *Management Planlets* would make sense to aggregate, i.e., that they are semantically compatible. Other aspects, such as the user needs, requirements and capabilities, as well as the context of the problem need to be considered too.

4.5 Concrete Solution Path

A concrete solution path is the mapping of a solution path (a.k.a. pattern sequence) from a pattern language to a solution language. It is the final result of the mapping phase of the pattern-based problem solving pipeline we introduced earlier (see Figure 2.2 on page 25). It consists of concrete solutions that map one-to-one to the patterns of a given solution path, as well as the aggregators needed later to merge these concrete solutions into a composite solution.

Definition 4.5.1 (Concrete Solution Path)
A concrete solution path ***csp*** *is a tuple that is defined as follows:*

$$csp \;=\; (cs_1,\; A_1,\; ...,\; cs_k,\; A_k ...,\; A_{n-1}, cs_n)$$

$$\begin{aligned} \textit{s.t. } & cs_1 ... cs_n \;\in\; \mathcal{CS}, \\ & A_1 ... A_{n-1} \;\subset\; \mathcal{A}, \\ & \forall k \in \{1, ..., n-1\},\; \forall a \in A_k \Rightarrow a(cs_k, cs_{k+1}), \\ & n \geq\; 1 \end{aligned}$$

That is, any two consecutive concrete solutions within the ***csp*** *are compatible and they can be aggregated using one of the aggregators stated in-between.*

In extreme cases, a concrete solution path can have only a single concrete solution with no aggregators. A concrete solution path can also have the same concrete solution present at different positions of the tuple. We see in the definition that between every two concrete solutions $cs_a,\; cs_b$ within a path there exists a set of aggregators instead of a single one. The reason is that there might exist more than one aggregator that can aggregate $cs_a,\; and\; cs_b$.

Note that the definition we present here does not give any guarantees about the validity of the concrete solution path, i.e., the requirements of the concrete solutions included may not all be fulfilled. We will define the notion of a **valid** concrete solution path later on in this thesis. Nevertheless, having the definition as it is here helps us to model some of the intermediate results of the mapping algorithm.

4.6 The Context of a Concrete Solution

When a concrete solution is selected to be part of a concrete solution path, it is no longer isolated, but rather "lives" with many other concrete solutions that are also selected in the same path. What each concrete solution expects from the others is to fulfill its set of requirements, and what it can do to the others is "offering" them with its own set of capabilities (see Section 4.2 on page 36).

The context of a concrete solution **cs** is intended to gather all the properties that **cs** can make use of to fulfill its requirements. It is derived from the capabilities of other concrete solutions within the same concrete solution path, as well as from the initial properties given by the user (see Section 4.3).

Definition 4.6.1 (Context of a Concrete Solution Path)
Let $\mathcal{CSP}$ be the set of all concrete solution paths, and $\mathcal{IP}$ be the set of all initial properties, and given a concrete solution path $csp \;=\; (cs_1, A_1, ..., A_{n-1}, cs_n) \;\in\; \mathcal{CSP}$ and a set of initial

properties $IP = \{ip_1, ..., ip_m\} \subset \mathcal{IP}$*, the context of csp, which is the context of each of the concrete solutions that constitute csp, is defined as follows:*

$$Context_{csp} = (\ (id_{ip},\ IP),\ (id_{cs_1}, Cap_{cs_1}),\ \dots\ ,\ (id_{cs_n}, Cap_{cs_n})\ )$$

The first entry of the context is the user-provided set of initial properties identified by a unique identifier id_{ip}*, and each of the remaining n entries represents an identified set of capabilities*[4]*.*

It is seen in the definition that we order the sets of capabilities in the same way the corresponding concrete solutions are ordered within the concrete solution path **csp**. The reason behind this is that the requirements of a specific solution, say cs_k, from **csp** might depend on the capabilities of the concrete solutions adjacent to it, specifically $cs_{k-1}\ and\ cs_{k+1}$. Thus, the order of concrete solutions must be preserved in the context. Furthermore, we see that the elements of the context are all identified with the same unique identifier of the corresponding concrete solution. This, again, results from the fact that a requirement of a concrete solution might reference the capabilities of another specific concrete solution by using its identifier, and to make the context capable of serving such a requirement, we keep the capabilities identified.

4.7 User Queries

The user might want to enforce certain conditions on the mapping process which are not directly enforced by the patterns that constitute the input solution path, nor by the requirements associated to individual concrete solutions. Such criteria could be, e.g., a condition regarding the total cost of the aggregate solution, a restriction, resulting from performance or legal concerns, on the regions of the servers that would host the solution, or a specification of some concrete solutions that the user requires to be included or excluded from the final concrete solution path. To this end, we introduce ***User Queries***.

Definition 4.7.1 (User Query)
A user query is a conditional statement that any concrete solution path ***csp*** *needs to fulfill in order for it to be* ***valid****. It is provided as an input by the user making it relevant to a specific use-case only. Hence, it is not stored within the solution language repository.*

User queries are very similar in nature to the requirements of a concrete solution in that they are conditional statements regarding concrete solutions. However, each requirement is associated to a specific solution, but user queries are associated to the whole selection process.

[4]Identified by the concrete solution identifier that the capabilities belong to

4.8 Fulfillment of Conditional Entities

In the previous sections, we have identified two types of conditional entities: (i) requirements of concrete solutions and, (ii) user queries. In this section, we explain what it means for a requirement or a user query to be fulfilled, and how to check their fulfillment.

4.8.1 Fulfillment of Concrete Solution Requirements

A requirement (Section 4.2) is fulfilled when its conditional statement is met. This can happen when concrete solutions with certain capabilities exist within the same concrete solution path that contains the solution owning the requirement being considered. This means that the conditional statement that constitutes a requirement usually addresses capabilities. A requirement can also address other aspects of concrete solutions such as their relative position within the concrete solution path under consideration, or even some global properties such as the initial properties provided by the user. In fact, the context **ctx** (see Section 4.6) of a concrete solution **cs** that resides within a concrete solution path **csp** serves as the **scope** which any of the requirements that belong to **cs** can "see"[5], and thus be built upon.

To describe the fulfillment of requirements we introduce the $fulfills_{req}$ function.

Definition 4.8.1 (Function: $fulfills_{req}$)
Let $\mathcal{CTX}$ be the set of all possible concrete solution contexts, $\mathcal{R}$ be the set of all requirements, and $\mathcal{ID}_{cs}$ be the set of all possible identifiers of concrete solutions, then $fulfills_{req}$ is a boolean function which is defined as follows:

$$fulfills_{req} : \mathcal{R} \times \mathcal{ID}_{cs} \times \mathcal{CTX} \rightarrow \{true, false\}$$

*$fulfills_{req}$ takes a requirement **r**, the identifier id_{cs} of the concrete solution **cs** that **r** belongs to, and the context **ctx** of **cs**, and returns true if the **r**'s condition is fulfilled by the values of **ctx** and id_{cs}. Otherwise, it returns false.*

How $fulfills_{req}$ is actually implemented depends mainly on the concrete definition of requirements of which we will introduce an example later in this thesis.

[5] In the sense of variable visibility.

4.8.2 Fulfillment of a User Query

Like a requirement, a user query (Section 4.7) is fulfilled when its conditional statement is met. This, too, can happen when the capabilities of the concrete solutions constituting the concrete solution path under consideration, as well as the initial properties provided by the user have suitable values. However, unlike requirements, the position of concrete solutions within the path is not important to user queries. Nonetheless, the context that all concrete solutions of a path share can also serve here as the scope that a user query "sees".

Similar to what we did in the case of requirements, in the following we define the function $fulfills_{uq}$ to describe the fulfillment of a user query:

Definition 4.8.2 (Function: $fulfills_{uq}$)
Let $\mathcal{UQ}$ be the set of all possible user queries, then $fulfills_{uq}$ is a boolean function which is defined as follows:

$$fulfills_{uq} : \mathcal{UQ} \times \mathcal{CTX} \rightarrow \{true, false\}$$

*$fulfills_{uq}$ takes a user query **uq**, and the context **ctx** resulting from a concrete solution path **csp**, and returns true if the **uq**'s condition is fulfilled by **ctx**. Otherwise, it returns false.*

Again, the actual implementation of $fulfills_{uq}$ depends on the concrete definition of user queries of which we will introduce an example later in this thesis.

As we have just seen, fulfillment of both conditional entities can only be considered when we have a context of a concrete solution path at hand. This means that the fulfillment of requirements and user queries is not static but rather situational in that it depends on the specific concrete solution path being considered, as well as the initial properties given by the user.

4.9 Validity of Concrete Solution Paths

In Section 4.5, we have introduced the notion of a concrete solution path, and we have pointed out that having such a path does not guarantee that the requirements of the included concrete solutions can be fulfilled (see Section 4.8.1). Here, we explain what it means for a concrete solution path to be valid and what checks we need to perform to make certain of validity.

The validity of a concrete solution path is evaluated contextually, i.e., it depends not only on the path itself, but also on the user query (Section 4.7) and the initial properties (Section 4.3) given by the user, and it depends on two conditions: (i) the fulfillment of the requirements of all concrete solutions included within the path (see Section 4.8.1), and (ii) the fulfillment of the user query (see Section 4.8.2).

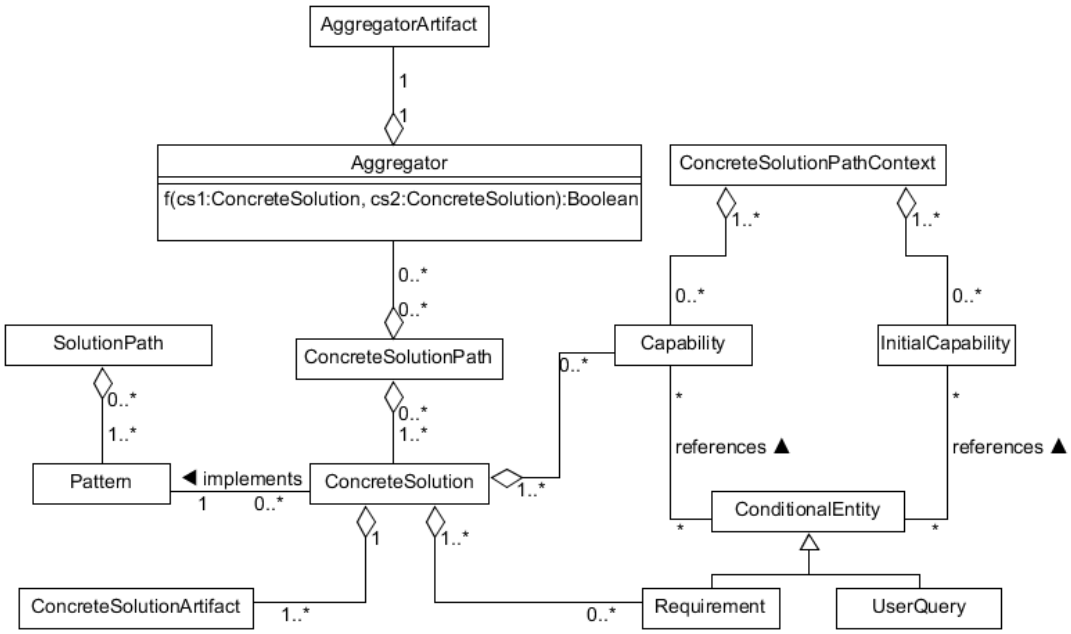


Figure 4.1: The complete conceptual design meta-model supporting the mapping phase of the pattern-based problem solving pipeline.

Definition 4.9.1 (Validity of a Concrete Solution Path)
Given a set of initial properties $IP = \{ip_1, ..., ip_m\} \subset \mathcal{IP}$*, a user query* $uq \in \mathcal{UQ}$*, and a concrete solution path* $csp = (cs_1, A_1, ..., A_{n-1}, cs_n) \in \mathcal{CSP}$*.*

Let ***ctx*** *be the context of* ***csp****, and* ***CS*** *be the set of all concrete solutions within* ***csp****, then we say that* ***csp*** *is a valid concrete solution path if the following conditions are met:*

(1) $\forall cs = (id_{cs}, A\_ID_{cs}, Req_{cs}, Cap_{cs}) \in CS \ \wedge \ \forall r \in Req_{cs} \Rightarrow fulfills_{req}(r,\ id_{cs},\ ctx) \ = \ true$

(2) $$fulfills_{uq}(uq,\ ctx) \ = \ true$$

By introducing the concept of concrete solution path validity we complete the meta-model (see Figure 4.1 for a complete overview of the meta-model) that allows us to propose algorithms to solve the mapping problem (Section 2.4 on page 24), but before that, we provide a formal definition of the problem itself and a concrete example to demonstrate it.

4.10 Problem Definition

The problem definition we introduce here summarizes the conceptual meta-model we have discussed so far, and identifies the task that needs to be accomplished by the mapping algorithm we will present later on.

Definition 4.10.1 (Problem Definition)
Given:

*(1) A **solution language** identified by a set of concrete solutions $CS \in \mathcal{CS}$ and a set of aggregators $A \in \mathcal{A}$.*

(2) A solution path represented by a tuple of patterns $sp = (p_1, ..., p_n)$ where $n \geq 1$ resulting from the pattern selection phase. (see Section 2.4 on page 24)

(3) A set of initial properties $IP \subset \mathcal{IP}$ given by the user.

(4) A user query $uq \in \mathcal{UQ}$ given by the user.

We need to find all concrete solution paths $csp = (cs_1, A_1, ..., A_{n-1}, cs_n) \in \mathcal{CSP}$ that fulfill the following conditions:

*(a) The concrete solution path **csp** maps in a one-to-one fashion to the solution path **sp**, i.e.:*

$$\forall\, p \in sp,\ \exists!\, cs \in csp \text{ such that } cs :: p,\ and$$
$$\forall\, cs \in csp,\ \exists!\, p \in sp \text{ such that } cs :: p$$

*(b) The concrete solution path **csp** is valid (see Definition 4.9.1).*

4.11 Practical Use Case

We demonstrate the conceptual design we have discussed so far by introducing a sample problem from the domain of cloud application architecture [FLR+14]. The example, which builds upon a similar use case first presented by Falkenthal et al. [FBB+14], is demonstrated in Figure 4.2 on the next page.

The general problem presented here is that the business logic of some application is implemented in an application component, and the expected workload of it is not static making it necessary for instances of this component to be provisioned and decommissioned dynamically. Determining the appropriate number of business-logic components as well as actually provisioning/decommissioning them is done by a different component. Finally, the various instances of the business-logic components share a central state which is stored externally in a storage offering.

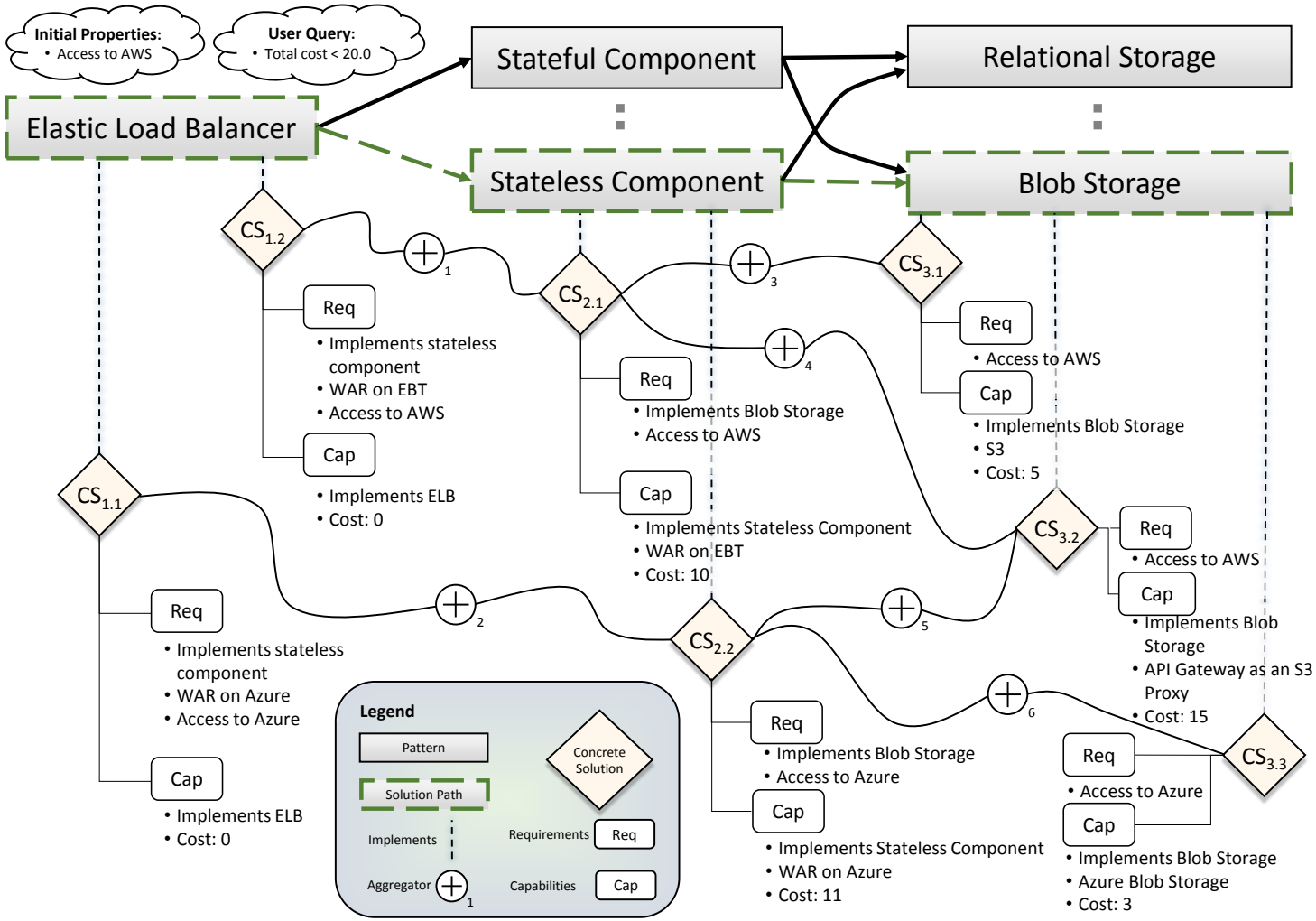


Figure 4.2: The problem of mapping a solution path to a concrete solution path in the domain of cloud applications architecture.

4.11.1 The Patterns

The patterns we show here are from the catalog of Fehling et al.[FLR+14]. *Stateless Component* and *Stateful Component* are both cloud application components but they differ in the way they treat the state. In this context, the "state" can mean both (i) the session state, i.e., the state resulting from the interaction between the client and the component, such as the items inserted so far by the client in a shopping cart, and (ii) the application state, i.e., the data that the application handles, such as the personal information of customers..

Stateful Component stores the state locally, and when scaled-out synchronizes it with other instances of the same component to provide a unified behavior. The developer, however, faces the decision of ensuring strict consistency or eventual consistency with a consistency-availability trade-off.

Stateless Component, on the other hand, does not store any internal state. Instead, it either stores the state in a storage offering, or expects it to be stored and (re-)delivered by the client with each request. When using *Stateless Components*, we benefit from increased robustness as the failure of a stateless component will not result of the loss of data as well as an increased capability to scaling out because provisioning and decommissioning operations of the component are simplified.

If the *Stateless Component* stores the state in an external storage offering, it can do so, e.g., in a *Relational Database* which is suitable for structured data, or in a *Key-value Storage* if the structure of the data needs to be more flexible. *Blob Storage* could also be used when dealing with larger objects like files.

Elastic Load Balancer is a management pattern that uses the number of synchronous accesses to an application component and possibly some other utilization parameters to determine the number of component instances that need to be active at the same time. And to enforce this number it initiates commission operations or decommission operations to the *Elastic Infrastructure* or the *Elastic Platform* that hosts the component.

The connections depicted between patterns here reflect some of the navigation links of the corresponding pattern language. In this example, we assume that three patterns were selected to solve the problem at a conceptual level, namely, *Elastic Load Balancer*, *Stateless Component* and *Blob Storage*, and these altogether constitute the solution path that is provided to the mapping phase (see Figure 2.2 on page 25) as input.

4.11.2 The Concrete Solutions

Below the patterns depicted in Figure 4.2 we see the concrete solutions defined within the solution language under consideration that are connected to the patterns of the solution

Listing 4.1 A JSON-formatted template fragment that shows the anatomy of a CloudFormation template [6].

```
{
  "AWSTemplateFormatVersion" :"version date",

  "Description" :"JSON string",

  "Metadata" :{
    //template metadata
  },
  "Parameters" :{
    //set of parameters
  },
  "Mappings" :{
    //set of mappings
  },
  "Conditions" :{
    //set of conditions
  },
  "Transform" :{
    //set of transforms
  },
  "Resources" :{
    //set of resources
  },
  "Outputs" :{
    //set of outputs
  }
}
```

path. These concrete solutions focus on implementations supporting the AWS cloud and the Azure cloud.

The artifacts associated with solutions $CS_{1.2}, CS_{2.1},\ and\ CS_{3.1}$ are code snippets of an AWS CloudFormation Template [Ama17a]. A CloudFormation Template has the general anatomy shown in Listing 4.1 in which the *Resources* section is the only required section and it is used to define cloud resources and their properties such as an Amazon Elastic Compute Cloud (EC2) instance or an Amazon Simple Storage Service (S3).

The aforementioned concrete solutions define specific resources that implement the associated patterns:

$CS_{1.2}$ defines two resources: (i) a regular load balancer ("AWS::ElasticLoadBalancing::LoadBalancer") responsible for spreading out synchronous requests to the application component

[6] http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/template-anatomy.html

Listing 4.2 A JSON-formatted template fragment that shows the anatomy of an Azure Resource Manager template.[8]

```
1 {
2    "$schema":
         "http://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
3    "contentVersion": "",
4    "parameters": {},
5    "variables": {},
6    "resources": [],
7    "outputs": {}
8 }
```

over its instances (see Listing A.1 on page 93), and (ii) an auto-scaling group ("AWS::AutoScaling::AutoScalingGroup") that gets the current load information from the regular load balancer, defines the scaling parameters and specifies the launch configuration of the component to scale (see Listing A.2 on page 93).

This concrete solution (as well as all others in this category) requires the user to have access to AWS. Furthermore, it expects a stateless component to be the target of scaling and works specifically for deploying Web Archive (WAR) on Elastic Beanstalk (EBT). To formalize all of this, a set of concrete solution requirements is attached to it. Moreover, $CS_{1.2}$ has two properties that might be relevant to other concrete solutions or to the user; with its two artifacts, it implements the *Elastic Load Balancer* management pattern (which is a functional property), and it costs 5 [7](which is a non-functional property). These two properties are depicted as a set of capabilities associated with the concrete solution.

$CS_{2.1}$ defines the launch configuration ("AWS::AutoScaling::LaunchConfiguration") of an application component bundled as an Amazon Machine Image (AMI) that will run on an EC2 instance (see Listing A.3 on page 93). Like $CS_{1.2}$, the capabilities and requirements of this concrete solutions are depicted in the figure.

$CS_{3.1}$ defines an S3 bucket ("AWS::S3::Bucket") with replication capabilities for prefixes associated with the application state (see Listing A.4 on page 94). Like before, the relevant requirements and capabilities are depicted in the figure too.

Microsoft Azure defines a template structure with similar capabilities to the CloudFormation Template that is called Azure Resource Manager Template [Mic17]. Listing 4.2 shows the general anatomy of the template, and similar to the previous case, the *resources* entry is used to define cloud resources and their properties.

[7]The unit of cost, as well as the meaning of cost are irrelevant to this example, we just assume consistency of the unit and the meaning whenever cost is mentioned

[8]https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-group-authoring-templates

Concrete Solution	Resource Name	Resource Type
$CS_{1.1}$	Load Balancer	"Microsoft.Network/loadBalancers"
$CS_{2.2}$	Virtual Machine	"Microsoft.Compute/virtualMachines"
$CS_{3.3}$	Azure Blob Storage	"Microsoft.Storage/storageAccounts"

Table 4.1: An Overview of the Resources Referenced by the Concrete Solutions Specific to the Microsoft Azure Cloud

Concrete solutions $CS_{1.1},\ CS_{2.2},\ and\ CS_{3.3}$ are all snippets of this template that define various Azure resources. They define equivalent resources to concrete solutions $CS_{1.2}, CS_{2.1},\ and\ CS_{3.1}$ respectively, and for this reason we will not go into details about them. We only show an overview of these concrete solutions as well as the name and formal type of the resources they reference in Table 4.1.

Like those specific to AWS, these concrete solutions are annotated with sets of requirements and capabilities that would help in their selection.

Concrete Solution $CS_{3.2}$ is different from the rest in that its usage is not restricted to one cloud provider or another. The reason is that, although it is associated to an S3 resource, it also defines an API Gateway [Ama17c] that acts as a proxy to the S3 service allowing it to be reached through the Internet using Hyper-Text Transfer Protocol (HTTP) as it declares a Representational State Transfer (REST) API for the service using a Swagger template [Sof17]. Amazon provides a fully explained step-by-step example on how this template can be formulated [Ama17b]. The example also shows the final resulting Swagger template. As usual, this concrete solution comes with its own sets of requirements and capabilities that are depicted on the figure. We notice here that although it is reachable from outside of AWS, it still requires access to AWS as it defines an AWS-specific resource, S3.

4.11.3 The Aggregators

As we have seen in Section 4.4 on page 38, the nature of the artifacts that aggregate concrete solutions depends mainly on the nature of the concrete solution artifacts themselves. In this scenario, concrete solution artifacts are code snippets with some variation points where references to other resources can be inserted, making the task of an aggregator is altering the code and "filling-in the blanks" inside so that concrete solution artifacts can "see" each other and interact together. Aggregators in this case would be small programs that achieve these tasks. Figure 4.2 shows 7 different aggregators connected to the pairs of concrete solutions that they can aggregate which is determined by the f function of each aggregator (see Definition 4.4.1).

To demonstrate the functionality of aggregators, we take the aggregator a_4 as an example. This aggregator can combine a concrete solution of type $CS_{2.1}$ which is a CloudFormation Template snippet with a concrete solution of type $CS_{3.2}$ which is a full Swagger Template. To do so, a_4 takes advantage of a special kind of CloudFormation resources that has the type "AWS::ApiGateway::RestApi" and is capable of embedding a Swagger (OpenAPI) template within its body. The aggregator's role in this case is adding this resource (see Listing A.5 on page 94) to the list of CloudFormation Template resources and fill the body with a copy of the Swagger snippet defined by the concrete solution $CS_{3.2}$. A complete description of the functionality of such an aggregation artifact can be found in this online article be Severson [Sev16].

4.11.4 User-Defined Entities

In this scenario the user, besides providing the sequence of patterns, has defined two entities that will affect the selection of concrete solutions, one of which is an initial property (see Section 4.3 on page 37) and the other is a user query (see Section 4.7 on page 41):

Initial Properties : the user has access to only AWS, which means that if such an access is required by some concrete solutions then this requirement should be fulfilled. The user provides this piece of information in the form of an initial property which we abstractly refer to as "*Access to AWS*". The absence of a similar initial property for Azure will automatically exclude all concrete solutions with the requirement "*Access to Azure*" from being considered while building a concrete solution path.

User Query : the user, in this case, is not interested in expensive solutions. For this reason, he/she provides a user query which is abstractly defined as "*total cost < 20.0*" which would force the selection algorithm to exclude concrete solution paths that have a total cost larger than or equal to 20.0.

4.12 Interpretation of Semantic Links

Semantic links between concrete solutions in a solution language as described by Falkenthal et al. [FBB+14][FL17] do not scale well when the number of concrete solutions is high or when they are complicated in terms of dependencies. The reason is that semantic links would be represented as direct links between concrete solution risking a possible worst-case connection complexity of $O(n^2)$.

Wettinger states that "Decoupling concrete solutions from requirements and capabilities is common practice to interlink diverse solutions without an explosion of direct links between them. Moreover, this enables the systematic classification of solutions regarding their requirements and capabilities."[Wet17]. In our work we have followed a similar approach:

by using requirements and capabilities to express the dependencies of concrete solutions, we decouple them from one another. This, however, comes at the cost of having to annotate concrete solutions with requirements and capabilities. What is different in our case from what was used by Wettinger in his work “Gathering Solutions and Providing APIs for their Orchestration to Implement Continuous Software Delivery” is that the requirements of a single concrete solution can reference more than one capability found in multiple other concrete solutions, which can be seen as several virtual semantic links all at once, or a single virtual hyper-edge targeting more than one concrete solution (see the notion of a Hypergraph [Sap11]). Furthermore, even if all or some of the dependencies of a concrete solution are fulfilled by another concrete solution there has to be an aggregator that can combine them in a meaningful way too.

4.13 Fulfillment of Conceptual Design Requirements

In Section 3.1.1 on page 29 we have introduced a set of requirements that we try to fulfill when designing the conceptual meta-model that describes the problem. In this section we show how the abstract conceptual design we provide here fulfills those requirements:

1. **Requirement R1** (A concrete solution should be able to uniquely identify its associated artifacts): According to our meta-model, each concrete solution has a set of unique identifiers for the artifacts it represents which are used to locate those artifacts and fetch them if necessary.

2. **Requirement R2** (A concrete solution should be able to handle an arbitrary format of the artifact): The conceptual design does not force any assumptions about the nature or the format of the artifacts associated to concrete solutions. The only assumption about them is that they can be addressed and uniquely identified by an identifier.

3. **Requirement R3** (Semantic links should be expressive): As it is explained in Section 4.12, the usage of requirements and capabilities to represent semantic links makes those links very dynamic as they are no longer "hard-coded" within concrete solutions. Furthermore, requirements provide a high degree of flexibility and expressiveness as they are represented as declarative conditional statements rather than a choice of a pre-defined set of semantic link types. We demonstrate this flexibility when we introduce a concrete definition of requirements and capabilities later in this thesis.

4.14 Summary

In this chapter we have introduced a conceptual model to describe the problem of mapping a solution path to a corresponding set of concrete solution paths. Furthermore, we have provided a concrete use case to demonstrate the various entities of the conceptual model. Finally, we showed how our model interprets semantic links between concrete solutions and how it fulfills the requirement we have introduced before.

In the next chapter, we introduce an algorithm that uses this conceptual design and performs the mapping task.

5 Concrete Solution Selection Algorithm

In the previous chapter we have introduced a conceptual model to describe concrete solutions and their related entities which included an abstract representation of capabilities and requirements. We have also defined the problem of mapping a solution path consisting of patterns from the layer of pattern languages to the layer of solution languages to form a resulting set of matching valid concrete solution paths.

In this chapter we introduce an algorithm that is capable of solving the aforementioned mapping problem [1]. We break down the algorithm into smaller procedures and introduce helping data structures to facilitate comprehension. After describing the algorithm, we discuss its complexity and demonstrate its functionality by running it against the concrete use-case of the previous chapter (see Section 4.11 on page 45).

5.1 The Algorithm

As mentioned before, the goal of the algorithm is solving the mapping problem presented in Section 4.10 on page 45 which requires finding the set of all *valid concrete solution paths* that map to a given *solution path* while taking the user input (*initial capabilities* and *user query*) into consideration.

5.1.1 The Overall Algorithm

The overall algorithm is divided into 2 phases; a path generation phase and a path filtering phase, and the reason behind that is explained in the following:

According to Section 4.9 on page 43 the validity of a concrete solution path entails the fulfillment of the requirements of all concrete solutions within the path, as well as the fulfillment of the user query. On the other hand, the conditional statements that constitute requirement and user queries can contain a reference to the capabilities of any concrete solution of the same path. In fact, we have introduced the concept of the concrete solution path context (see Section 4.6 on page 40) just to describe exactly what can be referenced

[1] In this work, we interchangeably refer to the same problem as "the mapping problem", "the selection problem", or "the matching problem" without any difference in the intended interpretation.

Algorithm 5.1 The general mapping algorithm

Input:

sp: a sequence of pattern names
ip: a set of initial properties
uq: the user query

Output:

$valid$: the resulting set of valid concrete solution paths

```
1: function MappingAlgorithm(sp,ip,uq)
2:     potential ← phase1(sp)
3:     valid ← phase2(potential, ip, uq)

4:     return valid
5: end function
```

within those conditional statements. However, we cannot build the context for a concrete solution path until this path is fully created which in turn means that we cannot guarantee the validity of a concrete solution path **while** it is being constructed but only **after** it is built.

For this reason, we have chosen to divide the algorithm (which is depicted in Algorithm 5.1) into two phases. The first phase produces a set of *potentially valid concrete solution paths* by only considering the connectivity of concrete solutions [2] and without taking the conditional statements into consideration. These statements are then checked in the second phase which filters out invalid concrete solution paths and produces a set with only the valid ones.

5.1.2 Phase 1 - Path Generation

The first phase in the mapping algorithm is depicted in Algorithm 5.2. The general idea of this phase is producing concrete solution paths which include concrete solutions that map in a one-to-one manner to the patterns of the solution path given as input. Furthermore, any two consecutive concrete solutions residing within the same path should have, at least, one aggregator that can aggregate them. We refer to these paths as **potentially valid concrete solution paths**.

To facilitate creating an algorithm that solves this task, we use a representation of the concrete solution path that is equivalent to the one introduced in Definition 4.5.1 on

[2] This notion will be explained later when discussing the 1st phase.

Algorithm 5.2 The algorithm for the 1st phase

Input:
sp: a sequence of pattern names // sp_i denotes the ith pattern in sp
Output:
$potential$: a set of potentially valid concrete solution paths

```
1: function PHASE1(sp)
2:     potential ← ∅                                  // initialize the resulting set
3:     startingNodes ← {cs ∈ CS | cs :: sp_1}         // all concrete solutions implementing the
                                                        first pattern
4:     for all cs_start ∈ startingNodes do
5:         path_current ← ( )                         // we start with an empty path
6:         step_start ← (∅, cs_start)                 // starting steps don't have aggregators
7:         DFTRAV(step_start, 0, sp, path_current, potential)  // traverse solutions to get all
                                                                   paths starting at cs_start and
                                                                   implementing sp
8:     end for

9:     return potential
10: end function
```

page 40 but more suitable for this purpose. The new representation considers the path as a sequence of path steps, and each step is simply a pair:

$$step_i \;=\; (A_i,\; cs_i)\; s.t\; A_i \;\subset\; \mathcal{A} \;\wedge\; cs_i \;\in\; \mathcal{CS}$$

Here, A_i is a set of aggregators and cs_i is a concrete solution, and A_i is expected to be a non-empty set unless the step represents the first step in the path. In this case $A_0 = \emptyset$.

The algorithm of this phase only cares for the connectivity of patterns (with each other, through aggregators, and with patterns that constitute the solution path). For this reason given a solution path $sp = (p_1, ..., p_n)$, we look at the concrete solutions that implement patterns of sp as an **acyclic edge-labeled directed graph** G_{sp}**(**V**,**E**,**w**)** (which is defined later in this section).

Having this representation in mind, the problem of this phase is then reduced to finding all paths from $CS_{start} = \{cs \in V \mid cs :: p_1\}$, the set of concrete solutions that implement the first pattern in the solution path, to $CS_{end} = \{cs \in V \mid cs :: p_n\}$, the set of concrete solutions implementing the last pattern in the solution path.

To this end, we use multiple invocations of depth-first traversal (see Algorithm 5.2) each of which starts from one of the nodes of CS_{start} to fill the set of paths called $potential$ with all potentially valid concrete solution paths.

The concrete algorithm for the depth-first traversal is explained later in this section. But first, let us look closely at the graph representation of the solution language.

Graph-Representation of the Solution Language

As mentioned before, the first phase of the mapping algorithm considers the solution language as an acyclic edge-labeled directed graph $G_{sp}(V, E, w)$. To understand how this graph looks like let us consider the example shown in Figure 5.1. Note how the concrete solution cs_a was virtually duplicated in the graph representation. In principle, let p be a concrete solution that occurs $n > 1$ of times within the solution path sp, and let CS^p be the set of concrete solutions implementing p. Then, in the graph representation, the set CS^P is (virtually)[3] duplicated n times. To formalize this, we first introduce the function $AnnotateNode$:

Definition 5.1.1 (Function: AnnotateNode)
Let $\mathcal{CS}$ be the set of all possible concrete solutions, then the function $AnnotateNode$ is defined as:

$$AnnotateNode : \mathcal{CS} \times \mathbb{N} \rightarrow \mathcal{CS}$$

$$AnnotateNode(cs,\ i) = cs^i$$

where cs and cs^i have the same components but are considered different concrete solutions.[4]

And now we introduce the function $GenerateNodes$ that is responsible of generating the set of nodes for the graph representation we mentioned earlier:

Definition 5.1.2 (Function: GenerateNodes)
Let $\mathcal{SP}$ be the set of all possible solution paths then $GenerateNodes$ is defined as:

$$GenerateNodes : 2^{\mathcal{CS}} \times \mathcal{SP} \rightarrow 2^{\mathcal{CS}}$$

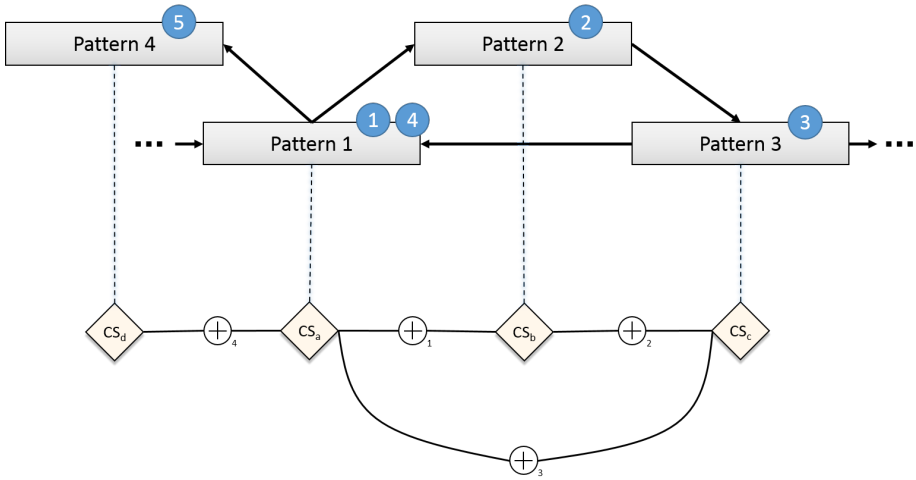
This function takes a set of concrete solutions CS that represents the concrete solutions of a certain solution language, as well as a solution path $sp = (p_1, ..., p_n)$ with possible duplicate patterns within it. Applying the function results in a new set of concrete solutions:

$$GenerateNodes(CS,\ sp) = \bigcup_{i=1}^{n} \{AnnotateNode(cs,\ i) \in \mathcal{CS} \mid \exists\ cs\ \in\ CS \wedge cs :: p_i\}$$

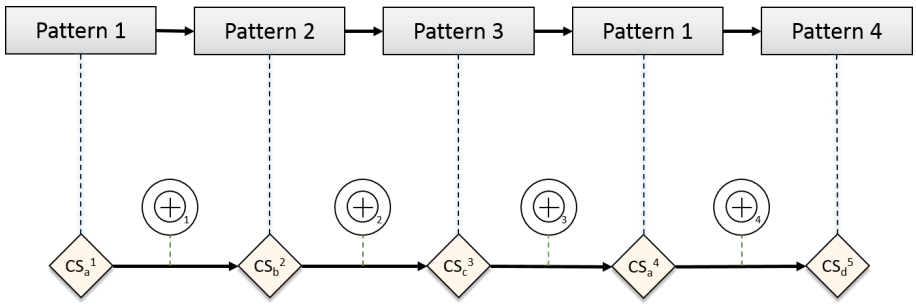
Now, we can concretely define the graph of concrete solutions as seen by the 1st phase of the mapping algorithm:

[3] We use the phrase "virtually duplicated" here because no concrete solutions are actually duplicated but rather we are describing how this phase of the algorithm considers the solution language.

[4] cs and cs^i can reside within the same set.



(a) The original solution language. The numbers in blue circles represent the order of occurrence of patterns within the solution path.



(b) The solution language after turning it into an acyclic edge-labeled directed graph. Notice how the same concrete solution cs_a occurs twice as a node.

Figure 5.1: Converting a Solution Language into an acyclic edge-labeled directed graph based on a given solution path sp as "seen" by the 1st phase of the mapping algorithm.

Definition 5.1.3 (Graph Representation of a Solution Language)
Given a solution language with a set of concrete solutions CS and a set of aggregators A. And given a solution path sp = $(p_1, ...p_n)$ with possible duplicate patterns inside. The graph representation of the solution language is an acyclic edge-labeled directed graph G_{sp} defined as:

$$G_{sp} = (V,\ E,\ w)\ \texttt{such that:}$$

$$V = GenerateNodes(CS,\ sp)$$
$$E = \{(cs_a,\ cs_b) \in V \times V \mid \exists\ a \in A \wedge a(cs_a,\ cs_b)\}$$
$$w : E\ \rightarrow\ 2^{\mathcal{A}}$$

w is a mapping function that is used to label edges: Given an edge e=(cs_a, cs_b) w returns a set of aggregators $A_e = \{a\ \in\ A \mid a(cs_a, cs_b)\}$ that can aggregate the two concrete solutions associated to e.

Depth-First Traversal of Concrete Solutions

The algorithm (depicted in Algorithm 5.3) is recursive, and it begins with pushing the last reached path step $step_{curr}$at the end of current path $path_{curr}$, then it checks for the recursion-breaking condition, i.e., it checks whether we have reached a concrete solution that implements the last pattern in the solution path sp. If this was the case, a copy of the current path $path_{curr}$ is pushed at the end of the set of all potentially valid concrete solutions $potential$. Otherwise, the algorithm searches for the "next" path steps, which is done in line 8.

To show how this works, we present an example snippet of a solution language limited to the concrete solutions implementing the solution path ('Pattern 1', 'Pattern 2', 'Pattern 3') (see Figure 5.2a). As we mentioned before the algorithm treats this as an **acyclic edge-labeled, directed graph** which is depicted in Figure 5.2b. As a result, if we "run" line 8 against the example of Figure 5.2, while the current concrete solution is cs_1, $nextSteps$ would then hold the following values:

$$nextSteps = \{(\{a_1, a_2, a_3\}, cs_2),\ (\{a_3, a_4\}, cs_3),\ (\{a_4\}, cs_4)\}$$

In fact, this specific line is responsible for the whole graph representation defined in Definition 5.1.3.

Then, the algorithm issues a new depth-first traversal starting from each path step in $nextSteps$.

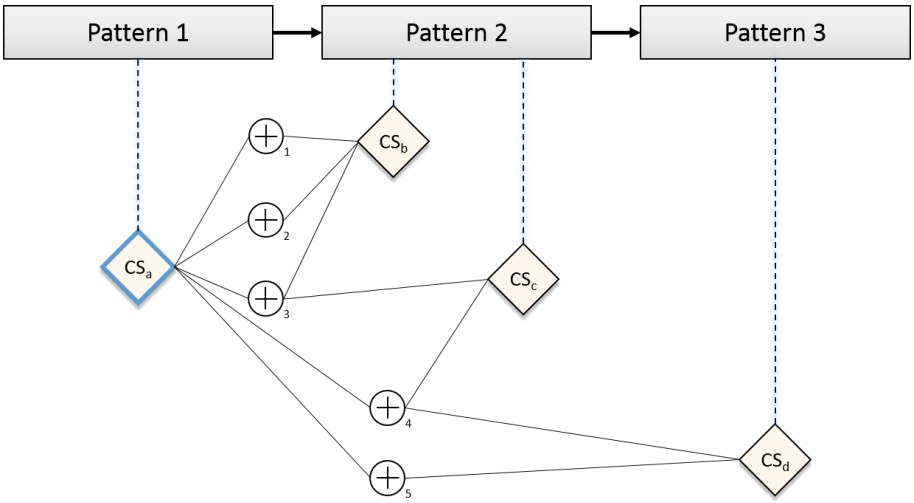
Finally, the algorithm finishes with popping out the current step from the current path as all possible paths going through it have been followed. This makes the algorithm "go back" one step in order to try other paths.

5.1.3 Phase 2 - Path Filtering

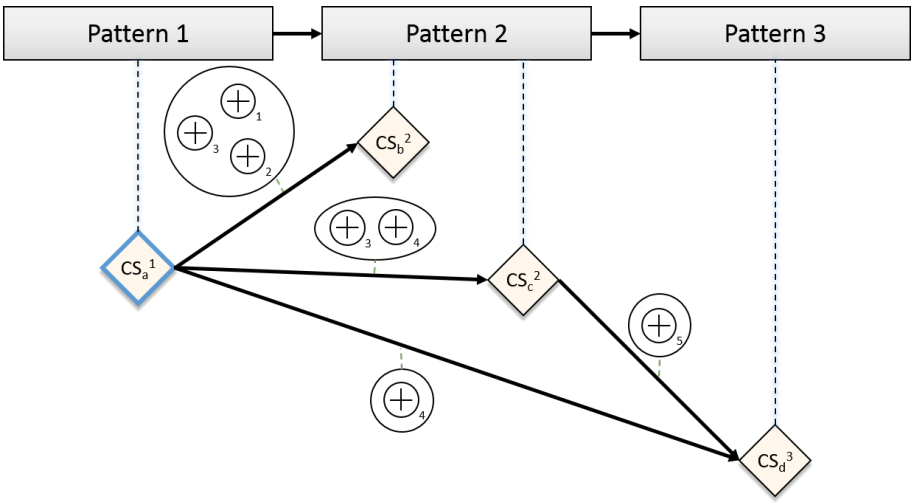
The purpose of this phase is filtering the set of **potentially valid concrete solution paths** in order to get only the valid ones (according to Definition 4.9.1 on page 44).

To this end, the algorithm utilizes the two boolean functions we introduced in Section 4.8 on page 42, namely, $fulfills_{req}$and $fulfills_{uq}$to check the fulfillment of concrete solution requirements, and the user query respectively.

The algorithm of this phase (shown in Algorithm 5.4) iterates through all potentially valid concrete solution paths stored in the variable $potential$, and for each path it creates the corresponding context (according to Definition 4.6.1 on page 40). This context is then used by the $fulfills_{uq}$function to check the fulfillment of the user query. If the user query turns out to be valid, all requirements attached to all concrete solutions of the current path are also checked for fulfillment using the $fulfills_{req}$function. The current path is only



(a) The original solution language snippet.



(b) The snippet after turning it into an Edge-Labeled, Directed Graph.

Figure 5.2: The way how the depth-first traversal algorithm knows which are the concrete solutions that come "after" cs_1. Figure 5.2a shows an original snippet of the solution language and Figure 5.2b shows the same snippet after converting it to an acyclic edge-labeled directed graph.

Algorithm 5.3 An algorithm to recursively traverse concrete solutions and get all concrete solution paths that start at a specific concrete solution and that implement a sequence of patterns.

Input:

$step_{curr}$: the current path step
$position$: the 0-based position reached in the current path
sp: a sequence of pattern names
$path_{curr}$: the current concrete solution path being constructed
$potential$: the set of all potentially valid concrete solution paths.

1: **procedure** DFTRAV($step_{curr}$, $position$, sp, $path_{curr}$, $potential$)
2: $path_{curr}$.PUSH($step_{curr}$)

3: **if** $position \geq sp.length - 1$ **then** // $path_{curr}$ is fully constructed
4: $potential$.PUSH($path_{curr}$) // add a copy of the current path to the final result
5: **else**
6: $cs_{curr} \leftarrow step_{curr}.cs$ // get the last concrete solution of current path
7: $position \leftarrow position + 1$
8: $nextSteps \leftarrow \{(A, cs) \in 2^{\mathcal{A}} \times \mathcal{CS} \mid cs :: sp_{position} \wedge A \neq \emptyset \wedge \forall a \in A \rightarrow a(cs_{curr}, cs)\}$

9: **for all** $step \in nextSteps$ **do**
10: DFTRAV($step$, $position$, sp, $path_{curr}$, $potential$) // traverse starting from next step
11: **end for**
12: **end if**
13: $path_{curr}$.POP( ) // done traversing through cs_{curr}, so go back
14: **end procedure**

considered valid when all of these requirements are fulfilled, and if so, the path is added to the final resulting set $valid$.

Note that the functions $fulfills_{req}$and $fulfills_{uq}$are only defined abstractly as their actual behavior mainly depends on the grammar that defines conditional statements and the way we choose to evaluate them. Later in this thesis, we concretely define one possible grammar for conditional statements and then we present a corresponding implementation for $fulfills_{req}$and $fulfills_{uq}$.

Algorithm 5.4 The algorithm for the 2nd phase

Input:

$potential$: a set of potentially valid concrete solution paths
ip: a set of initial properties
uq: the user query

Output:

$valid$: a set of only the valid concrete solution paths within $potential$

```
1:  function PHASE2(potential, ip, uq)
2:      valid ← ∅                                   // initialize the resulting set

3:      for all path ∈ potential do                  // iterate paths and filter-out the invalid
4:          allCS ← {cs ∈ CS | ∃(A, cs) ∈ path}
5:          ctx ← CREATECONTEXT(path)               // create the context of the current concrete
                                                       solution path
6:          isValid ← true                          // assume path is valid until otherwise proven
7:          if fulfills_uq(uq, ctx) then            // user query is fulfilled
8:              for all cs ∈ allCS do
9:                  for all req ∈ cs.Req do
10:                     if ¬ fulfills_req(req, cs.id, ctx) then      // req is not fulfilled
11:                         isValid ← false
12:                         break
13:                     end if
14:                 end for
15:                 if ¬ isValid then               // a req is not fulfilled, so no need to check
                                                       more concrete solutions in this path
16:                     break
17:                 end if
18:             end for
19:         else                                    // user query is not fulfilled
20:             isValid ← false
21:         end if
22:         if isValid then                         // path is valid, so add it to the result
23:             valid.PUSH(path)
24:         end if
25:     end for

26:     return valid
27: end function
```

5.2 Analysis of Time Complexity

In this section, we now analyze the worst-case time complexity of Algorithm 5.1. As we saw there, the algorithm is divided into two phases and thus the analysis here is also divided into two sections. But first we start with some notations. To measure the size of the problem, we assume the following:

n is the number of patterns in the pattern language.

m is the number of concrete solutions in the solution language.

K is the number of patterns within the solution path.

$\frac{m}{n}$ is the average number of concrete solutions that implement a pattern.

r is the average number of requirements per concrete solution.

c is the average number of capabilities per concrete solution.

$T_{fulfills_{uq}}$ is the worst-case time complexity of running the $fulfills_{uq}$function.

$T_{fulfills_{req}}$ is the worst-case time complexity of running the $fulfills_{req}$function.

T_{agg} is the worst-case time complexity of querying the aggregator repository[5] for all aggregators that can aggregate two specific concrete solutions.

For each of the two phases, we will gradually build the time complexity in the Big-O notation and then infer the total worst-case time complexity.

5.2.1 Time Complexity of Phase - 1

Let us assume that the input solution path is $sp = (p_1, ..., p_K)$. The time complexity[6] of finding out the starting nodes is $O(m)$ (we assume checking all concrete solutions). Next the algorithm uses depth-first traversal to visit all possible paths through graph G_{sp} defined in Definition 5.1.3 starting from concrete solutions implementing the first pattern in the solution path (CS^{p_1}) to concrete solutions implementing the last pattern (CS^{P_k}). In Figure 5.3, we show a graph that corresponds to the maximum possible number of potentially valid concrete solution paths. Each pattern in this graph is implemented by $\frac{m}{n}$ distinct concrete solutions, and each concrete solution of set CS^{p_i} is connected (can be aggregated with) each concrete solution of the next set $CS^{p_{i+1}}$, i.e., the two sets form a bipartite digraph. This makes the total number of edges between the two sets equal to:

[5] We assume that aggregators are stored in a separate repository called the aggregator repository.

[6] In the following, we mean by *complexity* the worst-case time complexity even if this is not clearly mentioned.

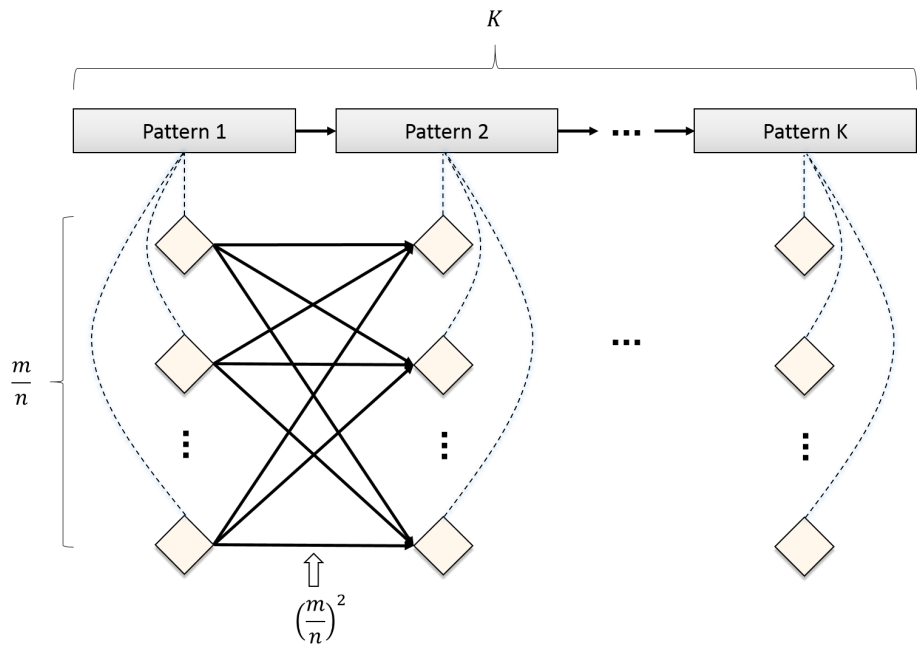


Figure 5.3: The graph that produces the worst-case number of **potentially valid concrete solution paths**. The labels of edges (sets of aggregators), as well as concrete solution id's are omitted for clarity.

$(\frac{m}{n})^2$. Moreover, this is repeated for K-1 times making the total number of possible paths from start to end (the total number of potentially valid concrete solution paths) equal to:

$$((\frac{m}{n})^2)^{K-1} = (\frac{m}{n})^{2(K-1)}$$

Having in mind the fact that this version of depth-first traversal aims at discovering all possible *paths* from start to end rather than traversing all nodes and edges, as is suggested by Corment et al.[CLRS09], the time complexity of executing this traversal is proportional to the number of paths instead of being $O(|V| + |E|)$. Moreover, whenever the algorithm 'jumps' from one concrete solution to the next using an edge, it queries the aggregator repository for aggregators that can combine the two ends of the edge (which happens while "running" the line 8). Assuming that the implementation for the algorithm remembers the result for this query (we only need to issue the query once for each edge), then the time

complexity for depth-first traversal is proportional to $O((K-1).(\frac{m}{n})^2.T_{agg} + (\frac{m}{n})^{2(K-1)})$, and the total complexity of this phase would be:

$$
\begin{aligned}
O(m) + O((K-1).(\frac{m}{n})^2.T_{agg} + (\frac{m}{n})^{2(K-1)}) = \\
O(T_{agg}.K.(\frac{m}{n})^2 + (\frac{m}{n})^K) = \\
O((\frac{m}{n})^K) = \ \texttt{\{if K > 2 and } T_{agg} \leq O(m)\} \\
O(m^K) \ \texttt{\{ if we consider } m \gg n \texttt{ \}}
\end{aligned}
$$

The time complexity of the 1st phase is polynomial in the number of concrete solutions constituting the solution language if the length of the solution path, i.e., K, is relatively small, but it can become very large when K is large.

5.2.2 Time Complexity of Phase - 2

As it was pointed out in Section 5.2.1, the worst-case number of potentially valid concrete solution paths is:

$$(\frac{m}{n})^{2(K-1)}$$

We need this number here as the 2nd phase of the mapping algorithm iterates through all of these paths and filters-out the non-valid ones. In order to do so, the algorithm, first, creates the context of the current path which his $O(K)$. Next, the algorithm checks the fulfillment of the user query with a worst-case execution time of $T_{fulfills_{uq}}$. Finally, the algorithm checks the fulfillment of each requirement of each concrete solution within the current path with a worst-case execution time of $O(r.K.T_{fulfills_{req}})$.

The overall complexity of phase - 2 would then be:

$$
\begin{aligned}
O((\frac{m}{n})^{2(K-1)}.(K + T_{fulfills_{uq}}).r.K.T_{fulfills_{req}}) = \\
O(m^K.(K + T_{fulfills_{uq}}).r.K.T_{fulfills_{req}}) \ \texttt{\{ if we consider } m \gg n \texttt{ \}}
\end{aligned}
$$

To further simplify things, it is expected that $T_{fulfills_{req}} \approx T_{fulfills_{uq}}$ because the nature of boolean expressions for concrete solution requirements as well as the user query is expected to be very similar. Let us call this common value "the worst-case execution time to evaluate a boolean expression" or T_{eval} in short. Then, the total complexity of this phase would be:

$$
\begin{aligned}
O(m^K.(K + T_{eval}).r.K.T_{eval}) = \\
O(m^K(r.K^2.T_{eval} + r.K.T_{eval}^2))
\end{aligned}
$$

Moreover, in order to evaluate a boolean expression it first needs to be parsed and then interpreted. Putting the execution time of parsing aside (as it is related to the specific

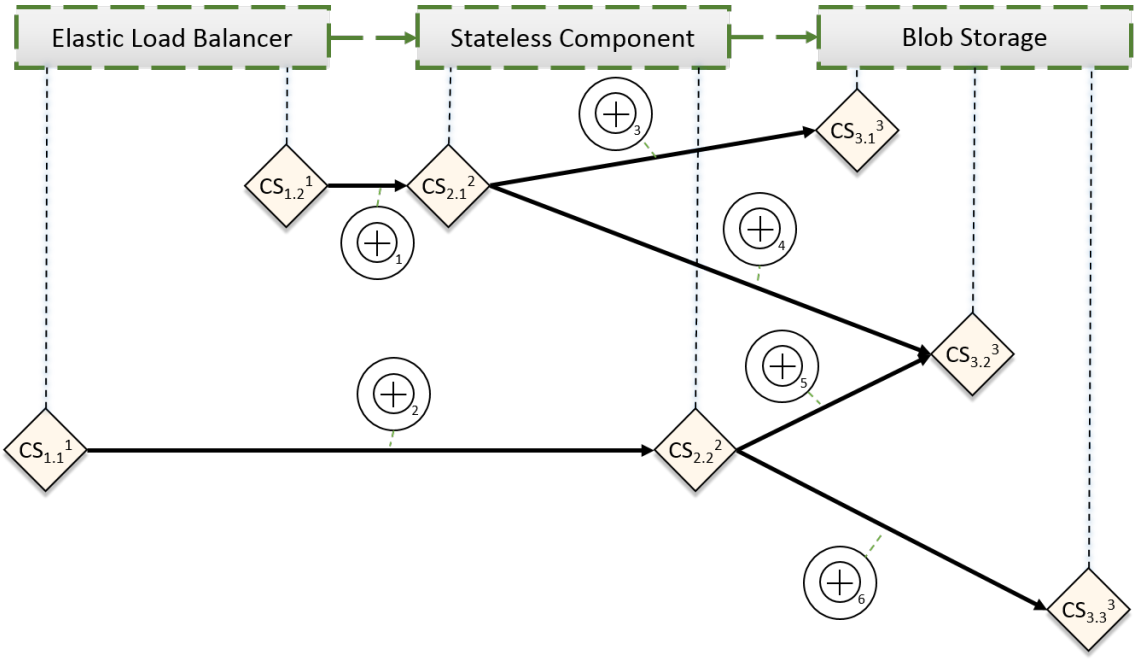


Figure 5.4: How the 1st phase of the mapping algorithm "sees" the solution language associated with the given use case.

algorithm used to perform the parsing), we know that the interpretation of the expression (in the worst-case) is $\geq O(c.K) > O(K)$ as we would need to iterate through all capabilities within the context of the current concrete solution path (which is of size K). This makes the overall complexity of the 2nd phase be:

$$O(r.K.T^2_{eval}.m^K)$$

Again, we see that the time complexity of the 2nd phase is also polynomial in the number of concrete solutions constituting the solution language if the length of the solution path, i.e., K, is relatively small, and that it can become very large when K is large. Moreover, the time to execute a boolean expression as well as the average number of requirements per concrete solution both contribute to the time complexity.

5.3 Practical Use Case

To demonstrate how the mapping algorithm works, we apply it to the same use case we introduced earlier in Section 4.11 on page 45.

5.3.1 Application of the 1st Phase

First, we show the effect of applying the 1st phase of the algorithm. To this end we introduce Figure 5.4. In this figure we see how this phase looks at the solution language, i.e., how the acyclic edge-labeled directed graph representing the problem of phase - 1 looks like. According to this figure, the task of phase - 1 would be finding all possible paths from the set of concrete solutions implementing the 1st pattern in the solution path, i.e., 'Elastic Load Balancer' (which is the set $\{cs^1_{1.1}, cs^1_{1.2}\}$) to the set of concrete solutions implementing the last pattern in the solution path, i.e., 'Blob Storage' (which is the set $\{cs^3_{3.1}, cs^3_{3.2}, cs^3_{3.3}\}$). This results in the following set of **potentially valid concrete solution paths**:

$$
\begin{aligned}
potential \; = & \\
& \{ \\
& \quad (\,(\emptyset,\ cs_{1.2}),\ (\{a_1\},\ cs_{2.1}),\ (\{a_3\},\ cs_{3.1})\,), \\
& \quad (\,(\emptyset,\ cs_{1.2}),\ (\{a_1\},\ cs_{2.1}),\ (\{a_4\},\ cs_{3.2})\,), \\
& \quad (\,(\emptyset,\ cs_{1.1}),\ (\{a_2\},\ cs_{2.2}),\ (\{a_5\},\ cs_{3.2})\,), \\
& \quad (\,(\emptyset,\ cs_{1.1}),\ (\{a_2\},\ cs_{2.2}),\ (\{a_6\},\ cs_{3.3})\,) \\
& \}
\end{aligned}
$$

Which constitutes, alongside the user query and the set of initial properties, the inputs of the second phase of the algorithm.

5.3.2 Application of the 2nd Phase

In this phase, the set of potentially valid concrete solution paths resulting from the previous phase is filtered out of the paths that either do not fulfill the user query "*the total cost is less than 20.0*" or has one ore more concrete solutions with invalid requirements.

Figure 5.5 shows how this phase processes the first concrete solution path of the set $potential$. Figure 5.5a shows how the user query is evaluated and what capabilities where used for this evaluation. Figure 5.5b show how the algorithm checks the validity of concrete solution $cs_{1.2}$ by checking all of its requirements. The figures show that these requirements are all fulfilled by the either the capabilities of other concrete solutions in the same path, or by the user-provided initial property. This also holds true for the requirements of the concrete solutions $cs_{2.1}$ and $cs_{3.1}$ (see figures 5.5c and 5.5d) making the first path, i.e., ($cs_{1.2}$, a_1, $cs_{2.1}$, a_3, $cs_{3.1}$)[7], a valid one.

Other paths of the $potential$ set are not valid; the path ($cs_{1.2}$, a_1, $cs_{2.1}$, a_4, $cs_{3.1}$) is not valid because it does not fulfill the user query, whereas paths ($cs_{1.1}$, a_2, $cs_{2.2}$, a_5, $cs_{3.2}$) and

[7]For clarity, instead of writing a set with a single aggregator, e.g.,$\{a_1\}$, we write the aggregator alone, i.e., a_1.

($cs_{1.1}$, a_2, $cs_{2.2}$, a_6, $cs_{3.3}$) are not valid because some of their concrete solutions require "*access to Azure*" which is not provided by their contexts. This makes the only valid concrete solution path be:

$$(cs_{1.2},\ a_1,\ cs_{2.1},\ a_3,\ cs_{3.1})$$

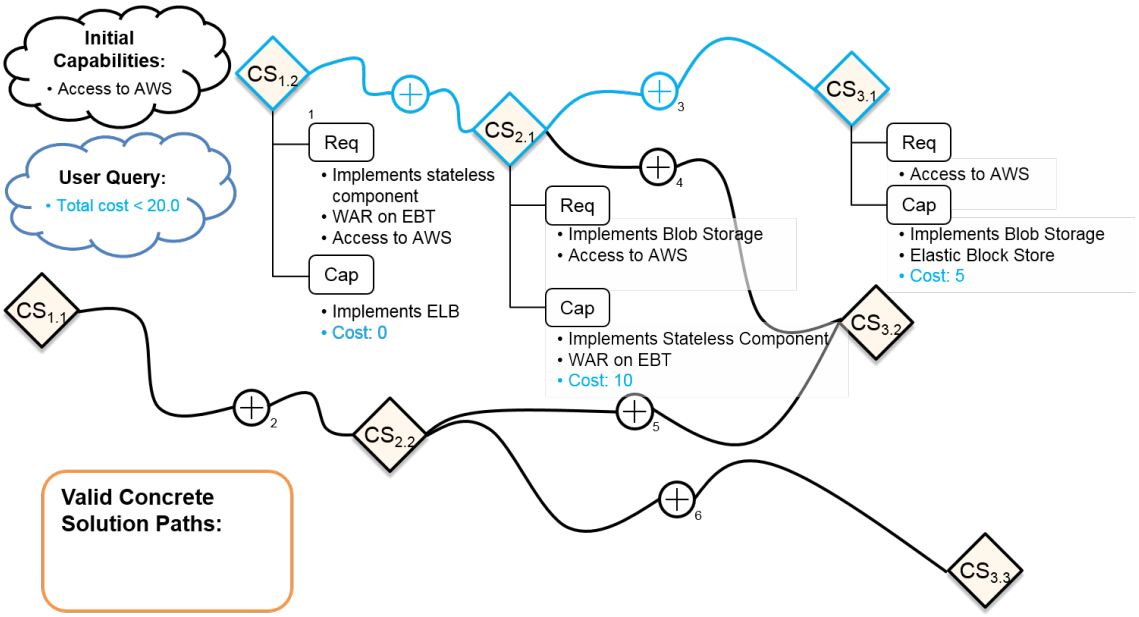
5.4 Fulfillment of Requirements on Algorithms

In Section 3.1.1 on page 29 we have introduced a set of requirements that we try to fulfill when designing the algorithms that solve the mapping problem. In this section we show how the algorithms we provide here fulfill those requirements:

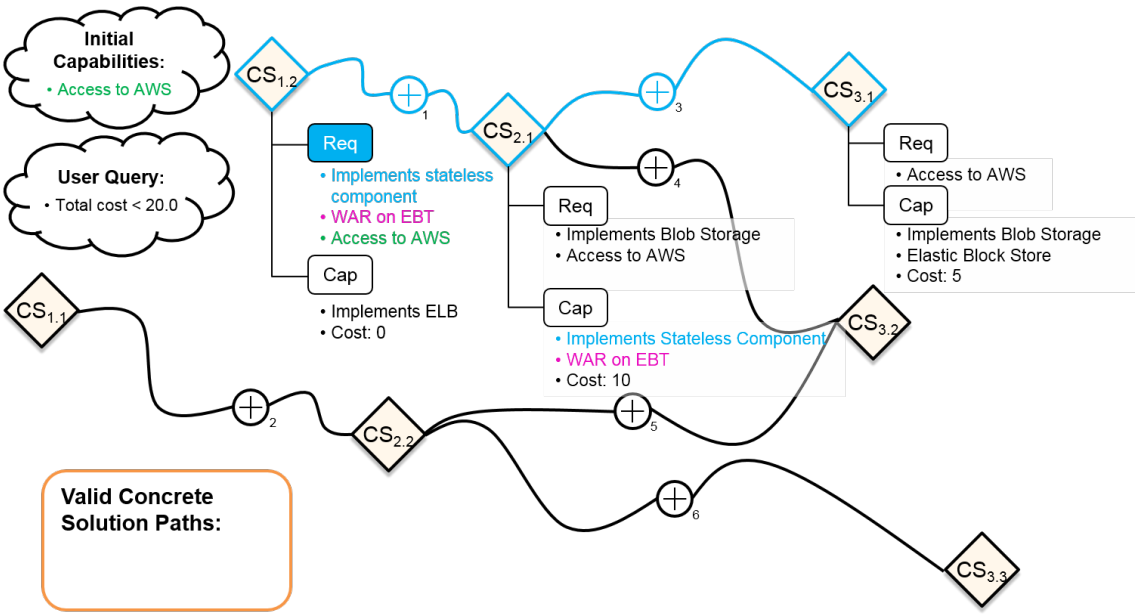
1. **Requirement R4** (the algorithm should allow the user to influence its result): The algorithm expects three kinds of inputs from the user: (i) the solution path consisting of a sequence of patterns, (ii) a collection of initial properties that describe the context in which the solution language is being used, and (iii) a user query used to provide constraints on the concrete solutions that are allowed to be in the resulting concrete solution paths. User queries, and to a smaller extent, initial properties are intended to **allow the user affect the output of the algorithm**. Looking at them from this point of view, user queries and initial properties have opposite effects: user queries are used to limit the result by providing conditions that describe the allowed concrete solutions, or the allowed paths, whereas initial properties are used to expand the set of paths by including more properties in the context which allows more requirements to be fulfilled and thus more valid concrete solution paths to be generated.

2. **Requirement R5** (the algorithm should be applicable to a wide range of solution languages): the algorithm strictly uses the meta-model described in Chapter 4, and does not introduce new restrictions on the supported pattern languages or solution languages. So, the applicability of the algorithm is derived from the applicability of the conceptual design introduced earlier.

3. **Requirement R6** (the algorithm should not assume any order of concrete solutions): the order of patterns in the input solution path affects the selection of concrete solutions only in terms of the existence of aggregators between concrete solutions implementing adjacent patterns. However, this order does not affect the fulfillment of concrete solution requirements. Actually, the introduction of the context that holds selection-relevant properties derived from all concrete solutions of a path (see Section 4.6 on page 40) is intended to achieve this independence from the order of concrete solutions. The reason why we can achieve this independence is that evaluating the validity of requirements using the context is done in 2-phases; the first builds the context (by building the path) and the second checks the validity with the help of the properties within the context. This allows a requirement of a

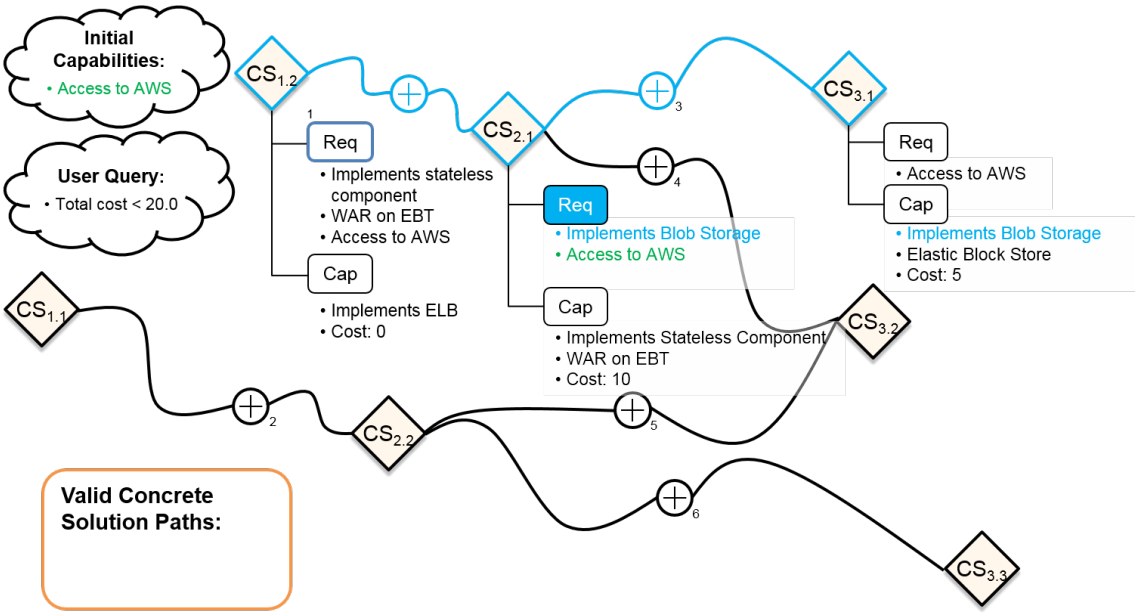


(a) Checking the fulfillment of the user query.

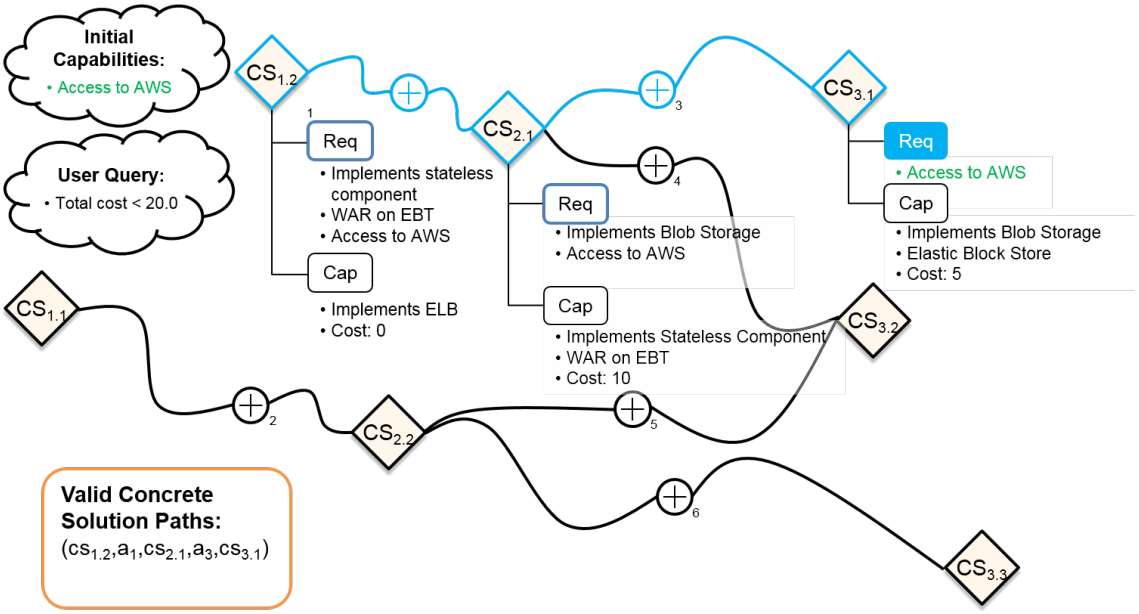


(b) Checking the fulfillment of requirements of $cs_{1.2}$.

Figure 5.5: The way the 2nd phase of the mapping algorithm processes the concrete solution path ($cs_{1.2}$, a_1, $cs_{2.1}$, a_3, $cs_{3.1}$). Requirements of the current concrete solution are color-coded.



(c) Checking the fulfillment of requirements of $cs_{2.1}$.



(d) Checking the fulfillment of requirements of $cs_{3.1}$.

Figure 5.5: The way the 2nd phase of the mapping algorithm processes the concrete solution path ($cs_{1.2}$, a_1, $cs_{2.1}$, a_3, $cs_{3.1}$). Requirements of the current concrete solution are color-coded. (continued)

concrete solution positioned anywhere in the concrete solution path to be fulfilled by a capability of another concrete solution of the same path **no matter where it is positioned**. So a requirement of a concrete solution cs **does not have to** use capabilities only defined by concrete solutions positioned before cs.

5.5 Summary

In this chapter we have introduced a 2-phase algorithm that maps a given solution path to a concrete solution path while taking the user input (initial properties and user query) into consideration. We enriched the description of the algorithm with concrete examples. Furthermore, we provided a worst-case time analysis of the two phases of the algorithm and concluded that it runs in polynomial time to the number of concrete solutions in the solution language, and that it can become exponential to the length of the input solution path when this length is large. This conclusion raises the need for enhancing the algorithm as it will not scale well when applied to large solution repositories. Finally, we showed how this algorithm fulfills the requirements we defined earlier for it.

In the next chapter, we present a prototypical implementation for the conceptual model and the mapping algorithm to demonstrate their feasibility. We also concretely define some abstract entities and functions of the meta-model.

6 Prototypical Implementation

In this chapter we prove the feasibility of the conceptual design and the algorithms we introduced in previous chapters by providing a prototypical implementation for them: First, we provide a concrete realization of the abstract aspects of the meta-model, then we describe the architecture of the implementation, and finally we discuss the implementation itself and show how it fulfills the requirements we have put for it.

6.1 Realization of Abstract Entities of the Conceptual Design

In Chapter 4 we introduced the conceptual design that describes concrete solutions and the entities related to them. One of the major requirements (see Section 3.1.2 on page 31 for details) that we tried to fulfill while building the conceptual model and the algorithms that operate on it was trying to support a wide range of solution languages. For this reason we tried to enforce as few assumptions as possible, and this resulted in some entities to be rather abstract. In this section we show how we have chosen to realize these entities in our prototypical implementation. Note that we do not claim that this realization is the only one possible nor that it is suitable for all domains.

6.1.1 Concrete Implementation of Capabilities and Initial Properties

Capabilities (as introduced in Section 4.2 on page 36) are abstractly defined as functional and non-functional properties of concrete solutions.

In the prototypical implementation, we represent capabilities with key-value pairs, and in order to enable describing various aspects of the same capability, we allow a single capability to be associated with multiple key-value pairs.

To be more precise, we provide a concrete definition of capabilities below:

Definition 6.1.1 (Capability)
A capability **c** *is defined as a pair* **c = (name, Properties)** *in which:*

name *is a name used to programmatically refer to the capability. It should be unique within the same concrete solution.*

Listing 6.1 JSON representation of the capabilities of some concrete solution. Notice that one capability can have multiple properties.

```
{
      "capabilities": [{
      "name": "deployedOn",
      "properties": {
        "value": "Azure"
      }
    },{
      "name": "deploymentFormat",
      "properties": {
        "value": "WAR"
      }
    },{
      "name":"location",
      "properties": {
        "value": "US"
      }
    },{
      "name": "implements",
      "properties": {
        "value": "Stateless Component"
      }
    },{
      "name": "cost",
      "properties": {
        "value": "11.0",
              "unit": "$"
      }
    }
   ]
}
```

Properties *is a none-empty set of key-value pairs that are used to describe various aspects of the capability.*

In order to store capabilities, we formulate them as JSON objects. Listing 6.1 shows a set of capabilities belonging to a concrete solution.

Initial properties have a similar nature to capabilities and thus we choose to give them the very same representation. The user interface of our prototype allows users to formulate initial properties themselves. Alternatively, the user can load a set of initial properties saved in a JSON file using a dedicated button.

6.1.2 Concrete Implementation of Conditional Entities

Concrete solution requirements (Definition 4.2.1 on page 36) as well as user queries (Definition 4.7.1 on page 41) are both conditional statements built against a concrete solution path context (Definition 4.6.1 on page 40).

We defined the context itself to contain the capabilities of concrete solutions constituting a path, as well as the initial properties provided by the user. Thus, conditional statements can address these capabilities and initial properties. A question remains though; what language should be used to express these conditional statements? In the following we try to justify our choices regarding this matter.

Choice of Language for Conditional Statements

In order to facilitate the choice of language for conditional statements we have put a list of minimum requirements that the language should support. These requirements are based on concrete use cases we envisioned for the domain of cloud applications architecture, and we expect them to be also applicable to other domains:

1. The language should allow us to address the capabilities of specific[1] concrete solutions.
2. The language should allow us to address capabilities of any/all concrete solutions that fulfill certain sub-conditions.
3. The language should allow us to express that specific concrete solutions should be included in/excluded from any **valid concrete solution path**.
4. The language should allow us to express that concrete solutions that fulfill specific sub-conditions should be included in/excluded from any **valid concrete solution path**.
5. The language should allow us to apply aggregate functions (e.g., SUM, COUNT) to capabilities.

Providing that capabilities are represented as JSON objects, using a variant of XML Query (XQuery) language that is suitable for JSON would be a viable option. In fact, many existing query languages that are designed for JSON and that cover the above-mentioned list of minimal requirements already exist. These languages include Jaql[2], JSONPath[3], JMESPath

[1] by specific we mean specified using its identifier.
[2] https://www.ibm.com/support/knowledgecenter/SSPT3X_1.1.0/com.ibm.swg.im.infosphere.biginsights.doc/doc/c0057474.html
[3] http://goessner.net/articles/JsonPath/

[4], and JSONiq[5]. Some of these languages do even have query engines that support them. However, there is a simple problem that prevents us from using one of these out-of-the-box query languages that is, they are simply too expressive for our purposes. For example, some of them have statements to alter the data/data structure, or to produce multiple outputs, or at least to give an output that is not boolean.

For this reason, we decided to create our own custom boolean expression language that results only in a single boolean value (as a conditional statement is expected to be). Of course, this language is still going to be executed against JSON structure and thus it borrows some concepts and notations from the languages listed above.

Next, we describe the features of this language and afterwards we explain how we interpret expressions built using it.

6.1.3 Specification of the Custom Boolean Expression Language

The language uses familiar notation and operation priority orders to express boolean and arithmetic expressions. For this reason, the following specification does not go deep into details unless needed:

- The whole expression should evaluate to a single boolean value.
- The keywords of the language are case insensitive.
- The language supports boolean expressions and arithmetic expressions.
- The language supports three data types (boolean, numeric, and string).
- Boolean expressions support comparisons of similar types.
- Expressions can be arbitrarily deeply nested.
- The simplest (atomic) element in the language (we call it a *value*) can be either a type-specific constant (e.g., TRUE, 1234, 'abc'), an arithmetic function, a boolean function, a wildcard, or a property of a capability of a specific concrete solution.
- In order to access a property of a capability of a specific concrete solution, we use the following notation:[6]

  ```
  CS["<CS-ID>"].<capabilityName>.<propertyName>
  ```

- Table 6.1 shows the list of boolean functions supported by the language.
- Table 6.2 shows the list of arithmetic functions supported by the language.

[4] http://jmespath.org/tutorial.html
[5] http://www.jsoniq.org/
[6] <..> indicates a place holder.

Function Syntax	Description
`EXISTS_CS(CS["<CS-ID>"])`	returns $true$ when the specified concrete solution exists within the current context (the current concrete solution path).
`EXISTS_CAP(CS["<CS-ID>"], <capabilityName>)`	returns $true$ when a capability with the specified name exists within the specified concrete solution (which should also exist).
`EXISTS_CAP(ANY, <capabilityName>)`	returns $true$ when a capability with the specified names exists at least once in any of the concrete solutions within the path.
`EXISTS_VAL(CS["<CS-ID>"], <capabilityName>, <propertyName>, <value>)`	returns $true$ if the specified property exists in the specified capability and concrete solution AND has the specified value.
`EXISTS_VAL(ANY, <capabilityName>, <propertyName>. <value>)`	returns $true$ if the specified property exists in the specified capability in any concrete solution AND has the specified value.

Table 6.1: The boolean functions supported by the custom boolean expression language.

- The wildcards (defined in Table 6.3) can only appear on one side of a comparison, and they cannot be part of an expression on that side (i.e., they should be "alone" on that side).
- Sub-condition are similar to conditional expressions in their syntax except for their *values*: they only support constants and properties of capabilities as *values*, and in the latter case capabilities refer to the *current* concrete solution being considered.

Appendix B.2 on page 96 shows the full definition of the grammar using ANother Tool for Language Recognition version 4 (ANTLR4).

An example of concrete solution requirements can be seen in Listing 6.2. As an example, let us look at the first requirement in the listing. In this requirement, we request that one of the neighbors of the concrete solution holding this requirement should have a capability which indicates that it implements the pattern called 'Stateless Component'.

Function Syntax	Description
`SUM(<capabilityName>.<propertyName>)`	returns the sum of the values defined by the specified property present within the specified capability among all concrete solutions of the path. The function tolerates missing properties/capabilities.
`COUNT(<capabilityName>.<propertyName>)`	returns the number of occurrences of the specified property present within the specified capability among all concrete solutions of the path.
`AVG(<capabilityName>.<propertyName>)`	returns the average of the values of a property. Effectively it is equivalent to: `SUM(<capabilityName>.<propertyName>)/ COUNT(<capabilityName>.<propertyName>)`

Table 6.2: The arithmetic functions supported by the custom boolean expression language.

Listing 6.2 JSON representation of the requirements of some concrete solution. Notice how we accessed multiple properties of a single capability in the second requirement.

```
{"requirements": [{
      "expression": "neighbor.implements.value = 'Stateless Component'"
    },{
      "expression": "any[deploymentFormat.value='WAR'].deployedOn.value = 'Azure'"
    },{
      "expression": "any.accessTo.value = 'Azure'"
    }]
}
```

Building the Custom Boolean Expression Language

In order to interpret an expression language [7], or any computer-based language for that matter, we need to build a tokenizer that is capable of turning the plain text of the expression into a series of predefined tokens. Furthermore, we need to build a parser that turns a stream of tokens into a parse tree according to the rules of some grammar, and finally we need to build an interpreter that traverses this tree in order to evaluate the expression.

Creating a parser, and to a smaller extent, a tokenizer, can be a tedious task, especially when the intended language has many rules to describe it or if it supports nesting of

[7] Of course, very simple languages can be analyzed using regular expressions only, but for more advanced situations such as when nesting expressions is needed, using regular expressions for this purpose can be too difficult, or even not possible.

Syntax	Description
`ANY[<sub-condition>].<capabilityName>.<propertyName>`	Appears as one side of a comparison, and means that the comparison will evaluate to $true$ when it is satisfied by the value of **at least one** of the occurrences of the specified property which belongs to **any** concrete solution that fulfills the specified sub-condition.
`ANY.<capabilityName>.<propertyName>`	equivalent to `ANY[true].<capabilityName>.<propertyName>`
`ALL[<sub-condition>].<capabilityName>.<propertyName>`	Appears as one side of a comparison, and means that the comparison will evaluate to $true$ when it is satisfied by the value of **all** occurrences of the specified property which belongs to **all** concrete solution that fulfills the specified sub-condition.
`ALL.<capabilityName>.<propertyName>`	equivalent to `ALL[true].<capabilityName>.<propertyName>`
`NEIGHBOR[<sub-condition>].<capabilityName>.<propertyName>`	**(specific to concrete solution requirements)** Appears as one side of a comparison and means that the comparison will evaluate to $true$ when it is satisfied by the value of **at least one** of the occurrences of the specified property which belongs to one of the concrete solutions that (i) fulfills the specified sub-condition, and that (ii) is a direct neighbor of the concrete solution holding the requirement.
`NEIGHBOR.<capabilityName>.<propertyName>`	**(specific to concrete solution requirements)** Equivalent to: `NEIGHBOR[true]. <capabilityName>.<propertyName>`

Table 6.3: The wildcards supported by the custom boolean expression language.

expressions. To deal with this recurring problem, some frameworks were invented that support the process of creating a tokenizer and a parser based on a given grammar. One of these frameworks is ANTLR4[Par17], and it outperforms other frameworks in terms of expressiveness of the grammars, as well as the number of output programming languages it supports.

T. Parr, the creator of ANTLR4, defines it as "... a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees"[Par17]. ANTLR4 allows writing grammars that look simpler

Figure 6.1: The process of using ANTLR4 in creating an interpreter for a language.

than grammars of other tools because it uses the Adaptive LL(*) algorithm for parsing and supports direct left-recursive rules through grammar rewriting [PHF14].

Figure 6.1 shows a simplified pipeline that describes how one would use ANTLR4 in order to interpret an intended language:

First, one would build a grammar that describes the intended language. In the case of ANTLR4, the grammar is divided into a lexer grammar that describes the tokens of the language, and a parser grammar that describes the syntax of the language.Appendix B on page 95 has more details about these two grammars for our own custom expression language. Second, one would use ANTLR4 to generate classes that represent the parser and the tokenizer. ANTLR4 is an open source project available at GitHub [8]. The main repository supports the following target languages: Java, C++, C#, Python (2, and 3), Javascript, Go, and Swift. Moreover, several other forks of the repository support other languages such as Typescript (the one we are using in this work).

Finally, one would build an interpreter by either implementing a listener which is the easier and the less capable option or a visitor which the more difficult but more powerful option. The interpreter would use the parse tree generated by the parser after applying the grammar rules on some input text.

6.2 Architecture

We have implemented our prototype as a client-side application using the Angular[9] platform. The latest version of Angular uses the TypeScript[10], an object-oriented programming language which is a typed superset of Javascript making Angular a very powerful and robust platform for developing client-side, sophisticated applications that have clear structure and a high degree of modularity, and that are easier to test. Furthermore, developing the application solely as client-side greatly eases managing it at production time because only minimal server-side maintenance is needed.

[8] https://github.com/antlr/antlr4
[9] https://angular.io/
[10] https://www.typescriptlang.org/

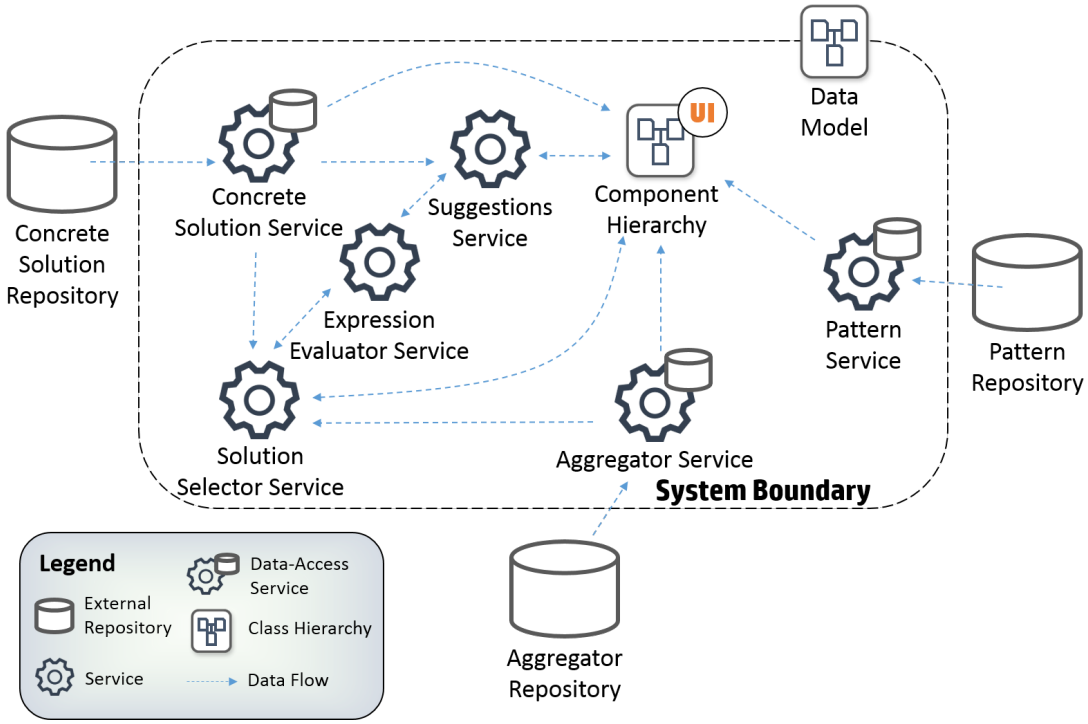


Figure 6.2: Architectural view on the prototype focusing on the flow of data within the system and across its boundaries.

At a very high level, Angular applications are built by specifying specially-annotated HyperText Markup Language (HTML) templates for visual presentation and associating them with component classes to control them and to capture user interactions. Business logic is built into services, and the whole application is divided into modules.

Figure 6.2 shows an overview of the system that focuses on the flow of data. As mentioned earlier, business-logic is realized through services and thus we will next focus on them as they contain the implementation of the algorithms introduced in Chapter 5 while briefly describing other parts.

6.2.1 External Repositories

The prototype is designed to communicate with three types of external repositories:

Concrete Solution Repository is responsible for storing concrete solutions and managing them. Remembering that the relationships between concrete solutions are represented through requirements and capabilities which are part of the concrete solution meta-data, the repository is in fact responsible of managing a solution language. It does not have to store the artifact associated with concrete solutions but rather the meta-data of each of them. We assume that the repository is capable

of fetching concrete solutions based on their identifiers and returns them as JSON objects.

Aggregator Repository is responsible for storing aggregators meta-data. Like the concrete solution repository, it is not expected to store the artifacts that actually perform the aggregation. The repository should be capable of looking up aggregators that can aggregate two given concrete solutions and returning these aggregators as JSON objects.

Pattern Repository is responsible of managing a pattern language. The repository is used only for visualization purposes and thus is only expected to report pattern names as well as their inter-links.

Existing repositories can play these roles. For example PatternPedia [FBFL15] is mainly designed as a general-purpose pattern repository, and can also manage concrete solutions. Muster Suchen und Erkennen-Pattern Search and Recognition (MUSE) [IAA17] is another repository specialized for film costumes patterns and concrete solutions. What we find as a viable option too is using an existing public and free online versioning system such as GitHub[11] to store the meta-data files of patterns, concrete solutions and aggregators. This approach serves two main purposes: (i) it removes the need to maintain the server that hosts the repositories as this will be the responsibility of GitHub, and (ii) a platform like GitHub has inherent support for collaboration which is very helpful when building and maintaining such repositories.

6.2.2 Services

In our implementation we can recognize two types of services; data-access services and business-logic services.

Data-Access Services

This special kind of services is intended to serve as an abstraction layer of external repositories, so that local components and business-services only use them when they need to access data stored in external repositories. By using these services, other parts of our implementation will not be affected if the representation of external repositories changes. Furthermore, they can serve as local caches for data fetched from repositories significantly speeding up access times. These services are designed to communicate with the external repositories through HTTP, and thus are rather easy to adapt when repositories change.

[11] https://github.com/

As we have three kinds of external repositories, we also have three kinds of data-access services; **concrete solution service**, **aggregator service**, and **pattern service**.

Business-Logic Services

These services represent the "brain" of the prototype; two of the services contribute to implementing the algorithms introduced in Chapter 5 and the third provides auto-complete suggestions for the user interface as explained below:

Solution Selector Service : This is the primary service of the application. It implements both of the phases of the mapping algorithm. The service gets its inputs (the solution path, the initial properties and the user query) from the user interface, and uses the concrete solution repository and the aggregator repository to retrieve the meta-data necessary to execute the algorithm. During the execution of the algorithm, conditional entities are evaluated with the help of another service, the **expression evaluation service**.

Expression Evaluation Service : This service is responsible of evaluating boolean expressions written in the custom expression language we have defined in Section 6.1.2. Such expressions include concrete solution requirements and the user query. This service uses the tokenizer, the parser, and the interpreter generated with the help of the ANTLR4 toolset.

Suggestions Service : This service provides auto-complete suggestions for the user interface; as an option, the user can enter initial properties by hand, and in order to decrease the possibility of errors, the user is presented with a list of dynamic suggestions for labels while typing them in. The service generates these suggestions by analyzing the requirements of concrete solutions and extracting capability names and property names from them. This makes the service use the expression evaluation service to analyze the relevant boolean expressions.

All of the services are designed to be stateless, thus the potential support for parallelism. Nonetheless, caching of intermediate and final results can enhance the performance of the mapping algorithm by decreasing its worst-case time complexity as we have seen in Section 5.2.

6.2.3 User Interface (UI) Components

Components, in Angular, are classes that controls the user interface. Each component is responsible for some part of the interface and they support nesting, i.e., a component can have child components. We designed our application to be a single page with multiple

sections. Thus, the whole user interface forms a single hierarchy of components rooted at the component that represents the whole page.

The UI provides users with the ability to provide the needed inputs for the mapping algorithm, and shows them the resulting valid concrete solution paths. It includes an interactive graph showing patterns, concrete solutions, and aggregators as well as the relations between them. This graph is used for picking patterns for the solution path, visualizing detailed information about each concrete solution and visualize the paths resulting from the algorithm.

To implement the interactive graphs we used the Swimlane NGX library[12], and for all other visual elements we used the PrimeNG library [13].

6.2.4 The Data Model

The data model is a hierarchy of classes that represent the various entities we have introduced in Chapter 4. The purpose of these classes is instantiating instances that can carry the data they represent within the system from one component to the other.

Data model objects get created by data-access services by parsing the raw data they receive from external repositories (which is usually in JSON format). These objects are then cached within these services in order not to have to retrieve them from external repositories and parse them each time they are required.

In Figure 6.2, all data flows within the system boundaries caries instances of classes from the data model hierarchy.

6.3 Practical Use-Cases

In this section we present some practical use cases built upon the concrete implementation we have chosen for capabilities and conditional entities. First, we show how having the correct combination of capabilities and requirements we can emulate the concept of semantic links between concrete solutions. Then we show some use-cases of user queries. These use-cases are applied to the example we first introduced in Section 4.11 on page 45 and then further refined in Chapter 5.

[12] https://github.com/swimlane/ngx-charts-dag
[13] https://www.primefaces.org/primeng/

6.3.1 Realization of Some Semantic Links

Should Be Aggregated Together If CS-A should be aggregated with CS-B then we can set one of A's requirements to be `EXISTS_CS(CS["B"])`, and one of B's requirements to be `EXISTS_CS(CS["A"])`.

Can Be Aggregated Together This is not determined by requirement/capabilities but rather with the existence of an aggregator in between.

Should Not Be Aggregated Together If CS-A and CS-B should not be part of the same concrete solution path we can set one of A's requirements, for example, to be: `not EXISTS_CS(CS["B"])`.

Alternatives If attached to CS-A there can be CS-B or CS-C (or both), then we can set one of A's requirements to be `EXISTS_CS(CS["B"])or EXISTS_CS(CS["C"])`.

Exclusive Alternatives If attached to CS-A there has to be either CS-B or CS-C, then we can set one of A's requirements to be `(EXISTS_CS(CS["B"])AND not EXISTS_CS(CS["C"]))OR (EXISTS_CS(CS["C"])AND not EXISTS_CS(CS["B"]))`.

Should not be Aggregated If we want CS-A to not be included in any concrete solution path (e.g., temporarily or for legal reasons), we can set one of its requirements to be: `FALSE` which can never be fulfilled.

Aggregable with Solutions of Certain Locality Only If we want to indicate that a CS such as CS-A should be aggregated with solutions with a certain locality (providing that the locality of concrete solutions is defined as a capability), we can use the following requirement: `ALL.location.value = 'DE'`. One would need such a case when enforcing data protection laws for example.

Notice how most of these examples are applied using only the function `EXISTS_CS` whereas the expression language is capable of expressing more sophisticated scenarios.

6.3.2 Sample User Queries

The Total Cost is Less than 20.0 To achieve this, we can use the following user query:
`SUM(cost.value)<20.0`

Limit the Technologies of Concrete Solutions To limit the technologies of concrete solutions to Azure for example, we can use the following user query:
`ALL.deployedOn.value = 'Azure'`

Complex Situation The following user query forces a specific concrete solution to be included in the concrete solution path while making sure its cost is less than 10.0 and that the total cost is less than 25.0, and requires that the locality of all used concrete solutions is either the USA or Europe, and makes sure that all used concrete solutions that implement the 'Elastic Load Balancer' pattern have at least '5' as the minimum number of application component instances, and finally restricts the allowed platforms on which concrete solutions are deployed to Azure and Elastic Beanstalk:

```
CS['cs1.2'].cost.value < 10 AND SUM(cost.value) < 25 AND
(ALL.location.value = 'EU' OR ALL.location.value = 'US') AND
ALL[implements.value = 'Elastic Load Balancer'].minComponents.value > 5 AND
(ALL.deployedOn.value = 'Elastic Beanstalk' OR All.deployedOn.value = 'Azure')}
```

6.4 Fulfillment of Implementation Requirements

In Section 3.1.3 on page 31 we have introduced a requirement that we try to fulfill when designing the prototypical implementation that proves the validity of the conceptual design and algorithms. In this section we show how the implementation we provide here fulfills this requirement:

R7 - The Implementation Should Be Realized Using the Serverless Architectural Style As we have mentioned earlier, our prototypical implementation is client-side only. Of course a web server needs to host the application, and the external repositories also need to be hosted, but all of this can be done in a cloud-setup where one will have minimal management responsibilities. Furthermore, single-page angular applications can be hosted easily and free-of-charge on GitHub using the GitHub Pages[14] making things even easier.

6.5 Summary

In this chapter, we have provided a concrete definition for the abstract entities of the conceptual design including concrete solution requirements and capabilities, as well as initial properties and user queries. These concrete definitions were necessary to develop a prototypical implementation intended for verifying the feasibility of the conceptual design itself as well as the algorithms we introduced earlier to solve the mapping problem.

The prototypical implementation is a client-side application that has access to external repositories managing concrete solutions, aggregators, and patterns. After describing the

[14] https://pages.github.com/

architecture of the implementation we provided concrete use-cases that show how we can solve sample problems and questions using the prototype. Finally, we have shown that the prototype follows the serverless architecture.

7 Conclusions and Future Work

In this work we have analyzed the problem of mapping a sequence of patterns, or a solution path, to a sequence of concrete solutions, or a concrete solution path. This issue arises when we solve a problem conceptually by picking a series of patterns from a pattern language, and then want to implement a concrete solution that realizes these patterns, and instead of doing that from scratch we decide to use an existing solution language formed of concrete solutions associated to the patterns of the pattern language and semantically inter-connected together. The problem would then be: How can we project the solution path onto the solution language in order to get a concrete solution path?

To solve this problem we first designed a conceptual model that describes it; it addresses concrete solutions and their requirements and capabilities, as well as the aggregators that can combine concrete solutions into composite ones. Furthermore, in this meta-model, we defined the concept of a concrete solution path, which represents a sequence of concrete solutions and aggregators, and the concept of a concrete solution path context which we considered as the set of cumulative properties of a concrete solution path. Moreover, we described two user-provided entities, namely, initial properties and user queries. After that, we were able to formally define the problem and set the exact characteristics of the concrete solution paths we want to find.

Although the meta-model we designed here captures many aspects of the problem and is capable of supporting its solution, it greatly depends on annotations; each concrete solution in a solution language has to be thoroughly analyzed to derive its capabilities and requirements, and then attach them to its meta-data. In this work, we have not addressed the problem of how to analyze concrete solution to get the needed meta-data, but we anticipate that this would be an issue that can hinder the adoption of our approach unless an automatic or a semi-automatic method was established to solve it.

Next, we developed an algorithm that solves the mapping problem, and we divided this algorithm into two phases. The first phase generates all concrete solution paths that map to the given solution path without taking the validity, i.e., the fulfillment of requirements and the user query, of the selected concrete solutions into consideration. This phase formulates the set of concrete solutions of a solution language as an acyclic edge-labeled directed graph and searches for the desired paths using depth-first traversal. The second phase of the problem applies the requirements and the user query to the paths resulting from the first phase in order to filter-out the invalid ones. After introducing the algorithm, we

analyzed its worst-case time complexity and showed that it is polynomial to the number of concrete solutions in the solution language which makes scaling it very difficult when the size of the solution language is large enough.

To prove the feasibility of the conceptual design and the algorithms, we developed a corresponding prototypical implementation as a client-side web application, and showed its internal architecture and how it can be connected to external repositories. Afterwards, we provided sample problems and ways to solve them using the prototype.

Throughout the thesis we used a running example from the domain of cloud computing architecture. We tried to make this example realistic, and we made sure it demonstrates many interesting aspects of the problem.

Future Work

During our work on this thesis, we have identified various points worth considering in the future.

1. As we mentioned earlier, the mapping problem is only one step in a longer pipeline and not all other steps are thoroughly studied. In fact, the very next step in the pipeline, i.e., aggregation of concrete solutions, is not fully shaped and analyzed yet. Nonetheless, this step is very important for the pipeline, and studying it might even change the way the mapping problem should be considered in the first place. Interesting questions that need to be answered with this regard include: Are aggregators operators in the space of concrete solutions? and what are the properties of the concrete solutions resulting from applying them? Can we find a set of finite number of aggregators sufficient to operate on all concrete solutions of some domain? and does this set differ significantly from one domain to the other?

2. We have studied the case in which we assumed that a concrete solution implements exactly one pattern; however, in reality some concrete solutions already implement a composition of patterns[1]. This is especially relevant while aggregating concrete solutions. If we want our meta-model to be relevant for the aggregation step, our concept of a concrete solution should be adapted to support concrete solutions resulting from the aggregation of smaller ones.

3. We have assumed that pattern selection process results in a sequence of patterns. However, Falkenthal et al. [FBB+16] argue that the result is a sub-graph of patterns with layers corresponding to different levels of abstraction. The meta-model and the algorithms should be adapted to support this generalized approach.

[1]This suggests that the set of patterns might be eligible to become a composite pattern

4. In the worst case, the algorithm is prohibitively complex. Thus, it has to be enhanced while focusing on reducing its time-complexity. This could be done, for example, by applying filtering in the first phase, so we limit the potential number of paths that result from it, or by having some pre-knowledge about the domain which allows us to limit the algorithm to a single phase by assuming, e.g., that all requirements should be fulfilled by concrete solutions that come "before" the current one in the path.

5. The algorithm uses simple string matching to compare labels, which is needed, for example, when evaluating requirements against capabilities. However, the use of label taxonomies, which was suggested by Wettinger [Wet17] for a similar setup, would make requirement matching more intelligent. To illustrate this idea, let us assume that we have a requirement of some concrete solution that states: `ANY.deploys.value = "Operating System"` and that some other concrete solution has the following capability: `{"name":"deploys""properties":{"value":"SUSE Linux"}}`. Then, using regular string matching, the requirement will not be fulfilled, but when using label taxonomies for comparison, the requirement will be fulfilled as the taxonomy would recognize that "SUSE Linux" $is\ a$ "Operating System". However, using such taxonomies should be thoroughly studied before adopting them as they introduce additional maintenance burden.

6. During the usage of the prototype we have noticed that many factors contribute to the final result; the input solution path, the set of initial properties, the capabilities and requirements of concrete solutions, the existence or non-existence of aggregators, and the user query. This, sometimes, makes the result difficult to understand. What can help in such situation is including the provenance of the result through Why- or How-Provenance [CCT09]. Furthermore, Why-Not-Provenance [BHT14] could also be very interesting for query debugging purposes.

7. Even though the current prototype is designed to make communication with external repositories for concrete solutions, aggregators, and patterns easy, it currently only uses local JSON files for this purpose. Connecting it to existing pattern/solution repositories can widen the domain of accessible concrete solutions and thus further prove the feasibility of our approaches.

A Template Snippets

A.1 AWS CloudFormation Template Snippets of Concrete Solutions

```
"MyLB" :{
      "Type" :"AWS::ElasticLoadBalancing::LoadBalancer",
      "Properties" :{
            "Listeners" :[{
                  "LoadBalancerPort" :"80",
                  "InstancePort" :"80",
                  "Protocol" :"HTTP"
            } ],
      }
}
```

Listing A.1: A regular *Load Balancer* that listens to external requests and routes them to back-end instances.

```
"MyAutoscalingGroup" :{
      "Type" :"AWS::AutoScaling::AutoScalingGroup",
      "Properties" :{
            "AvailabilityZones" :{["eu-west-1"] },
            "LaunchConfigurationName" :{"Ref" :<VAR>},
            "MinSize" :"1",
            "MaxSize" :"3",
            "LoadBalancerNames" :[{"Ref" :MyLB} ]
      }
}
```

Listing A.2: An *Auto-Scaling Group* resource that references a regular *Load Balancer* and the *Launch Configuration* of a *Stateless Component* (defined as a placeholder to be filled later by an aggregation operator).

```
"MyCfg" :{
      "Type" :"AWS::AutoScaling::LaunchConfiguration",
      "Properties" :{
            "ImageId" :{"ami-StatelessComponent" },
            "InstanceType" :{"m1.large" },
      }
}
```

Listing A.3: The *Launch Configuration* of a *Stateless Application Component* (defined as a placeholder to be filled later the user).

```
"MyS3Bucket": {
 "Type": "AWS::S3::Bucket",
 "Properties": {
  "VersioningConfiguration":{
   "Status":"Enabled"
  },
  "ReplicationConfiguration": {
   "Role": "arn:aws:iam::123456789012:role/replication_role",
   "Rules": [
    {
     "Id": "MyRule1",
     "Status": "Enabled",
     "Prefix": "MyApplicationStatePrefix",
     "Destination": {
      "Bucket": "arn:aws:s3:::my-replication-bucket",
      "StorageClass": "STANDARD"
     }
    }
   ]
  }
 }
}
```

Listing A.4: *S3 Bucket* with application state replication capability that serves as an implementation to the *Blob Storage* pattern.

A.2 AWS CloudFormation Template Snippets of Aggregators

```
"MyRestApi": {
 "Type": "AWS::ApiGateway::RestApi",
 "Properties": {
  "Body": {
   //OpenAPI specification
  }
  "Description": "A test API",
  "Name": "MyRestAPI"
 }
}
```

Listing A.5: Definition of an *API Gateway* resource for CloudFormation that is able to embed a Swagger template in its body.

B Grammar for Conditional Statements

B.1 Lexical Rules

In this section, we present the lexical rules common for both the grammars of concrete solution requirements and user queries. A lexical rule defines a primitive token that the lexer (or tokenizer) should recognize. Tokens are like the words of a language.

```
lexer grammar CommonTokens;

AND
   :A N D;
OR
   :O R;
NOT
   :N O T;
CS
   :
   C S '[' STRING_LITERAL ']'
   ;
INITIAL_CAPABILITY
   :I C;
ANY
   :A N Y;
ALL
   :A L L;
EXISTS_CS
   :E X I S T S '_' C S;
EXISTS_CAP
   :E X I S T S '_' C A P;
EXISTS_VAL
   :E X I S T S '_' V A L;
SUM
   :S U M;
COUNT
   :C O U N T;
AVG
   :A V G;

BOOL_CONSTANT
   :TRUE|FALSE;
TRUE
   :T R U E;
FALSE
   :F A L S E;
VARIABLE
  : VALID_VAR_START VALID_VAR_CHAR*
  ;
STRING_LITERAL
  : '\'' .*? '\'';
DOT
   :'.';
COMMA
   :',';
LPAR
   :'(';
RPAR
   :')';
LBRAC
   :'[';
RBRAC
   :']';
SUM_MINUS
   :'+'|'-';
MULT_DIV
   :'*'|'/';
SCIENTIFIC_NUMBER
  : NUMBER ( E SIGN? NUMBER )?
  ;

fragment
VALID_VAR_START
  : ('a'..'z')
  | ('A'..'Z')
  | '_'
```

```
  ;
fragment
VALID_VAR_CHAR
  : VALID_VAR_START
  | ('0'..'9')
  ;
fragment
NUMBER
  : ('0'..'9')+ ('.' ('0'..'9')+)?
  ;
fragment A:('a'|'A');
fragment B:('b'|'B');
fragment C:('c'|'C');
fragment D:('d'|'D');
fragment E:('e'|'E');
fragment F:('f'|'F');
fragment G:('g'|'G');
fragment H:('h'|'H');
fragment I:('i'|'I');
fragment J:('j'|'J');
fragment K:('k'|'K');
fragment L:('l'|'L');
fragment M:('m'|'M');
fragment N:('n'|'N');
fragment O:('o'|'O');
fragment P:('p'|'P');
fragment Q:('q'|'Q');
fragment R:('r'|'R');
fragment S:('s'|'S');
fragment T:('t'|'T');
fragment U:('u'|'U');
fragment V:('v'|'V');
fragment W:('w'|'W');
fragment X:('x'|'X');
fragment Y:('y'|'Y');
fragment Z:('z'|'Z');
fragment
SIGN
  :
  ( '+'
  | '-')
  ;
WS
  :[ \t\n\r]+ ->skip;
```

Listing B.1: ANTLR4 lexer grammar rules to be used by the parser rules for concrete solution requirements and user queries. Note that white spaces are skipped, and that all tokens are case-insensitive.

B.2 Parser Rules

In this section we introduce the ANTLR4 rules for parsing concrete solution requirements. Rules for parsing user queries are very similar to these, they only omit the NEIGHBOR accessor.

These parsing rules use the lexical rules we presented in Appendix B.1.

```
grammar RequirementsGrammar;
import CommonTokens;

booleanExpression
   : BOOL_CONSTANT
   | CS DOT VARIABLE DOT VARIABLE
   | multiValueVariable
   | EXISTS_CS LPAR CS RPAR
   | EXISTS_CAP LPAR (ANY |CS) COMMA VARIABLE RPAR
   | EXISTS_VAL LPAR (ANY |CS) COMMA VARIABLE COMMA VARIABLE COMMA
              (SCIENTIFIC_NUMBER |STRING_LITERAL |BOOL_CONSTANT) RPAR
```

```
12     | '(' booleanExpression ')'
13     | NOT booleanExpression
14     | multiValueVariable ('<>'|'>='|'>'|'<='|'<'|'=') arithmeticExpression
15     | arithmeticExpression ('<>'|'>='|'>'|'<='|'<'|'=') multiValueVariable
16     | arithmeticExpression ('<>'|'>='|'>'|'<='|'<'|'=') arithmeticExpression
17     | multiValueVariable ('<>'|'=') stringValue
18     | stringValue ('<>'|'=') multiValueVariable
19     | stringValue ('<>'|'=') stringValue
20     | booleanExpression AND booleanExpression
21     | booleanExpression OR booleanExpression
22     ;
23
24 arithmeticExpression
25     : SCIENTIFIC_NUMBER
26     | CS DOT VARIABLE DOT VARIABLE
27     | (SUM |COUNT |AVG) LPAR VARIABLE DOT VARIABLE RPAR
28     | SUM_MINUS arithmeticExpression
29     | LPAR arithmeticExpression RPAR
30     | arithmeticExpression MULT_DIV arithmeticExpression
31     | arithmeticExpression SUM_MINUS arithmeticExpression
32     ;
33
34 stringValue
35     : STRING_LITERAL
36     | CS DOT VARIABLE DOT VARIABLE
37     ;
38
39 multiValueVariable
40     :(ANY|ALL|NEIGHBOR) DOT VARIABLE DOT VARIABLE
41     |(ANY|ALL|NEIGHBOR) LBRAC fBooleanExpression RBRAC DOT VARIABLE DOT VARIABLE
42     ;
43
44 fBooleanExpression
45     : BOOL_CONSTANT
46     | VARIABLE DOT VARIABLE
47     | LPAR fBooleanExpression RPAR
48     | NOT fBooleanExpression
49     | fArithmeticExpression ('<>'|'>='|'>'|'<='|'<'|'=') fArithmeticExpression
50     | fStringValue ('<>'|'=') fStringValue
51     | fBooleanExpression AND fBooleanExpression
52     | fBooleanExpression OR fBooleanExpression
53     ;
54
55 fArithmeticExpression
56     : SCIENTIFIC_NUMBER
57     | VARIABLE DOT VARIABLE
58     | SUM_MINUS fArithmeticExpression
59     | LPAR fArithmeticExpression RPAR
60     | fArithmeticExpression MULT_DIV fArithmeticExpression
61     | fArithmeticExpression SUM_MINUS fArithmeticExpression
62     ;
```

```

fStringValue
   : STRING_LITERAL
   | VARIABLE DOT VARIABLE
   ;


NEIGHBOR
   :N E I G H B O R;
```

Listing B.2: ANTLR4 parser grammar rules used to recognizing and parse concrete solution requirements. There is a single lexer rule at the end that defines the requirements-specific key-word: NEIGHBOR.

Bibliography

[ACG+96] M. Adams, J. Coplien, R. Gamoke, R. Hanmer, F. Keeve, K. Nicodemus. "Fault-Tolerant Telecommunication System Patterns." In: *Rising [45]* (1996), pp. 81–94 (cit. on p. 21).

[AIS+77] C. Alexander, S. Ishikawa, M. Silverstein, J. R. i Ramió, M. Jacobson, I. Fiksdahl-King. *A Pattern Language: Towns, Buildings, Construction*. Gustavo Gili, 1977 (cit. on pp. 17, 21).

[Ale79] C. Alexander. *The Timeless Way of Building*. Vol. 1. New York: Oxford University Press, 1979 (cit. on pp. 17, 21).

[Ama17a] I. Amazon Web Services. *AWS CloudFormation Templates*. Amazon Web Services, Inc. 2017. URL: https://aws.amazon.com/cloudformation/aws-cloudformation-templates/ (visited on 08/20/2017) (cit. on pp. 22, 48).

[Ama17b] I. Amazon Web Services. *Create an API as an Amazon S3 Proxy*. Sept. 14, 2017. URL: http://docs.aws.amazon.com/apigateway/latest/developerguide/integrating-api-with-aws-services-s3.html (cit. on p. 50).

[Ama17c] I. Amazon Web Services. *What Is Amazon API Gateway?* Sept. 14, 2017. URL: http://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html (cit. on p. 50).

[BBKL13] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. "Pattern-based Runtime Management of Composite Cloud Applications." In: *CLOSER 2013*. 2013, pp. 475–482 (cit. on p. 39).

[BEM+12] J. Bergin, J. Eckstein, M. Manns, H. Sharp, K. Marquardt, J. Chandler, M. Sipos, M. Volter, E. Wallingford. *Pedagogical Patterns: Advice for Educators*. CreateSpace, 2012 (cit. on p. 21).

[BHT14] N. Bidoit, M. Herschel, K. Tzompanaki. "Query-Based Why-Not Provenance with NedExplain." In: *Extending Database Technology (EDBT)*. 2014. URL: https://hal.inria.fr/file/index/docid/962157/filename/ned_edbt2014.pdf (cit. on p. 91).

[BL15] J. Barzen, F. Leymann. "Costume Languages as Pattern Languages." Englisch. In: *Proceedings of PURPLSOC (Pursuit of Pattern Languages for Societal Change). The Workshop 2014*. Ed. by P. Baumgartner, R. Sickinger. Krems: PURPLSOC 2015, June 2015, pp. 88–117. ISBN: 978-3-7375-5458-9. URL: http://www2.informatik.uni-stuttgart.de/cgibin/NCSTRL/NCSTRL_view.pl?id=INPROC-2015-38&engl= (cit. on pp. 21, 22).

[CCT09] J. Cheney, L. Chiticariu, W.-C. Tan. "Provenance in Databases: Why, How, and Where." In: *Foundations and Trends® in Databases* 1.4 (2009), pp. 379–474. ISSN: 1931-7883. DOI: 10.1561/1900000006. URL: http://dx.doi.org/10.1561/1900000006 (cit. on p. 91).

[CLRS09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*. MIT press, 2009, pp. 540–543 (cit. on p. 65).

[Dou03] B. P. Douglass. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time systems*. Vol. 1. Addison-Wesley Professional, 2003 (cit. on p. 21).

[FBB+14] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann. "Efficient pattern application: validating the concept of solution implementations in different domains." In: *Int J Adv Softw* 7.3&4 (2014), pp. 710–726. ISSN: 1942-2628. URL: http://www.iaas.uni-stuttgart.de/RUS-data/ART-2014-13%20-%20Efficient%20Pattern%20Application.pdf (cit. on pp. 17, 22, 24–26, 37, 39, 45, 51).

[FBB+16] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann, A. Hadjakos, F. Hentschel, H. Schulze. "Leveraging Pattern Application via Pattern Refinement." In: *Proceedings of the International Conference on Pursuit of Pattern Languages for Societal Change (PURPLSOC)*. epubli GmbH, 2016, pp. 38–61. URL: http://www.iaas.uni-stuttgart.de/RUS-data/INPROC-2016-47%20-%20Leveraging%20Pattern%20Applications%20via%20Pattern%20Refinement.pdf (cit. on pp. 17, 25, 26, 90).

[FBFL15] C. Fehling, J. Barzen, M. Falkenthal, F. Leymann. "PatternPedia – Collabroative Pattern Identification and Authoring." In: *Proceedings of the International Conference on Pursuit of Pattern Languages for Societal Change (PURPLSOC)*. epubli GmbH, June 2015. URL: http://www.iaas.uni-stuttgart.de/RUS-data/INPROC-2015-51%20-%20PatternPedia.pdf (cit. on pp. 22, 27, 82).

[FL17] M. Falkenthal, F. Leymann. "Easing Pattern Application by Means of Solution Languages." In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services (XPS), 2017, pp. 58–64. URL: http://www.iaas.uni-stuttgart.de/RUS-data/INPROC-2017-10%20-%20Solution%20Languages.pdf (cit. on pp. 17, 18, 23, 25, 26, 38, 51).

[FLR+13] C. Fehling, F. Leymann, S. T. Ruehl, M. Rudek, S. Verclas. "Service Migration Patterns - Decision Support and Best Practices for the Migration of Existing Service-based Applications to Cloud Environments." In: *Proceedings of the 6th IEEE International Conference on Service Oriented Computing and Application (SOCA)*. IEEE Computer Society, 2013, pp. 9–16. DOI: 10.1109/SOCA.2013.41. URL: http://www.iaas.uni-stuttgart.de/RUS-data/INPROC-2013-53%20-%20Service%20Migration%20Patterns.pdf (cit. on pp. 21, 39).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014. DOI: 10.1007/978-3-7091-1568-8 (cit. on pp. 21, 32, 39, 45, 47).

[Fow02] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002 (cit. on p. 21).

[HW04] G. Hohpe, B. Woolf. *Enterprise Integration Patterns: Designing, building, and deploying messaging Solutions*. Addison-Wesley Professional, 2004 (cit. on p. 21).

[IAA17] IAAS. *MUSE*. Institut für Architektur von Anwendungssystemen. Sept. 23, 2017. URL: http://www.iaas.uni-stuttgart.de/forschung/projects/MUSE/ (cit. on p. 82).

[JGHV95] R. Johnson, E. Gamma, R. Helm, J. Vlissides. "Design patterns: Elements of reusable object-oriented software." In: *Boston, Massachusetts: Addison-Wesley* (1995) (cit. on pp. 22, 23).

[MD97] G. Meszaros, J. Doble. "A Pattern Language for Pattern Writing." In: *Proceedings of International Conference on Pattern languages of program design (1997)*. Vol. 131. 1997, p. 164 (cit. on p. 21).

[Mic17] Microsoft. *Azure Resource Manager*. Sept. 14, 2017. URL: https://docs.microsoft.com/en-us/azure/azure-resource-manager/ (cit. on p. 49).

[OAS13] OASIS. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0*. Nov. 25, 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html (visited on 08/17/2017) (cit. on p. 36).

[Par17] T. Parr. *ANTLR*. Sept. 23, 2017. URL: http://www.antlr.org/ (cit. on p. 79).

[PCW05] R. Porter, J. O. Coplien, T. Winn. "Sequences as a basis for pattern language composition." In: *Science of Computer Programming* 56.1-2 (2005), pp. 231–249. URL: http://www.sciencedirect.com/science/article/pii/S0167642304001868 (visited on 08/20/2017) (cit. on pp. 24, 27, 32).

[PHF14] T. Parr, S. Harwell, K. Fisher. "Adaptive LL (*) parsing: the power of dynamic analysis." In: *ACM SIGPLAN Notices*. Vol. 49. 10. ACM. 2014, pp. 579–598. URL: https://www.researchgate.net/profile/Terence_Parr/publication/273188534_Adaptive_LL_Parsing_The_Power_of_Dynamic_Analysis/links/55a94f9e08ae481aa7f98119/Adaptive-LL-Parsing-The-Power-of-Dynamic-Analysis.pdf (cit. on p. 80).

[Rob16] M. Roberts. *Serverless Architectures*. 2016. URL: https://martinfowler.com/articles/serverless.html (visited on 04/15/2017) (cit. on p. 32).

[Sap11] A. Sapozhenko. *Hypergraph*. Encyclopedia of Mathematics. Feb. 7, 2011. URL: https://www.encyclopediaofmath.org/index.php/Hypergraph (cit. on p. 52).

[Sev16] M. Severson. *Introduction to Swagger for CloudFormation and API Gateway*. Sept. 18, 2016. URL: https://blog.jayway.com/2016/09/18/introduction-swagger-cloudformation-api-gateway/ (cit. on p. 51).

[Sof17] S. Software. *Swagger (OpenAPI) Specification*. 2017. URL: https://swagger.io/specification/ (cit. on p. 50).

[WBFL17] J. Wettinger, U. Breitenbücher, M. Falkenthal, F. Leymann. "Collaborative gathering and continuous delivery of DevOps solutions through repositories." In: *Computer Science-Research and Development* 32.3-4 (2017), pp. 281–290 (cit. on pp. 30, 33, 35, 36).

[Wet17] J. Wettinger. "Gathering Solutions and Providing APIs for their Orchestration to Implement Continuous Software Delivery." PhD thesis. Institut für Architektur von Anwendungssystemen der Universität Stuttgart, 2017, p. 72 (cit. on pp. 27, 51, 91).

[Zdu07] U. Zdun. "Systematic pattern selection using pattern language grammars and design space analysis." In: *Software: Practice and Experience* 37.9 (2007), pp. 983–1016 (cit. on pp. 21, 24, 27, 32).

All links were last followed on September 26, 2017.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature