Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Design Pattern Detection Framework for TOSCA-Topologies

Marvin Wohlfarth

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. h. c. Frank Leymann
Supervisor:	Jasmin Guth Michael Falkenthal
Commenced:	4. Januar 2017
Completed:	4. Juli 2017
CR-Classification:	D2.11, D2.3, G2.2, I.5.0

Abstract

Cloud Computing Patterns are Design Patterns especially for cloud applications and provide abstract solution concepts for often reoccurring problems during the implementation of cloud applications. These concepts are mainly used by developers and modelers. To learn about implemented patterns in a completed application, one has to manually analyze the code and the architecture. To improve this time-consuming method, the possibility of automating this process is investigated. This bachelor's thesis proposes an approach for a Design Pattern Detection Framework, to perform an automatic pattern detection. TOSCA, provided by OASIS, is a standardized description for the development of cloud applications. Their architectures can be described by TOSCA topologies, to model components and relationships among each other. The framework, which is developed in the context of this bachelor's thesis, is written in Java and integrated in Winery, a graphical modeling tool for TOSCA topologies, which is a part of the OpenTOSCA ecosystem. The underlying concept of this work follows an approach to detect which *Cloud Computing Patterns* are used in TOSCA topologies. The concept defines the modeling of *Cloud Computing Patterns* with TOSCA topologies and how TOSCA topologies are abstracted, to be comparable with pattern topologies. Further, the use of pattern taxonomies is explained to include the interrelations of *Cloud Computing Patterns*. Basically, patterns and TOSCA topologies are handled as graphs. Consequential, probabilities for possible patterns can be set. For the detection of pattern graphs in a topology graph, an algorithm for subgraph isomorphism is used.

Contents

List of Figures

List of Algorithms

1 Introduction

The idea of cloud computing was mentioned for the first time in 1961 by Paul McCarthy when he talked about utility computing [QLDG09]. The breakthrough of cloud computing started between 2003 and 2006, when big companies like Google[1], Amazon[2], and Microsoft[3] started to offer public cloud services [QLDG09]. Enterprises started to outsource their IT resources, because of economical reasons. Today, cloud computing is present in nearly every section of the business community as well as in the private life of many people who are users of cloud applications. With an increasing use of applications hosted in the cloud, the need of standardization increased, too, to enable automatic deployment and portability of applications [BBKL14a]. The architecture of cloud applications describes the basic components and their interaction. To model such architectures, topologies are used. A topology represents the structure of an application by modeling the different components and their relations. Therefore, the Topology and Orchestration Specification for Cloud Applications (TOSCA) standard is developed to provide a uniform standard for the deployment and the administration of cloud applications. Furthermore, concepts are designed to provide possible solutions to frequently occurring problems. These concepts are called *Cloud Computing Patterns* and can be used independently from used technologies.

Motivation

Cloud Computing Patterns provide good solution concepts for reoccurring problems concerning the building and managing of cloud applications to fit the requirements efficiently. Further, these concepts are abstract, independent of concrete vendor products [FLR+14], and reduce the complexity of the architectures. Retrieving patterns in the architecture of an application provides a sufficient overview about the underlying concept. A manual search for realized patterns in an application is very time-consuming because one has to understand the architecture and behavior of an application in detail.

[1] https://www.google.com

[2] https://www.amazon.com

[3] https://www.microsoft.com

By performing a manual search, the advantage of patterns is lost because nevertheless one needs to know the implementation details. The aim of this bachelor's thesis is to develop a concept for an automatic detection of *Cloud Computing Patterns* in TOSCA topologies to provide a quick overview on the used architectural structures of a cloud application and to present an implementation of the introduced concept. Therefore, an approach to model *Cloud Computing Patterns* using TOSCA topologies is developed. Also, a procedure to recognize topology fragments in a TOSCA topology is designed.

Structure

The work is structured as follows:

Chapter 2 – Fundamentals
The basic knowledge for this work is described. This includes essential parts of the TOSCA-Specification, the OpenTOSCA-Ecosystem, and Design Patterns, i.e., *Cloud Computing Patterns*.

Chapter 3 – Related Work
In this chapter works related to the presented approach are described. Therefore, it is investigated if OpenTOSCA already offers approaches for pattern detection. Afterwards, an overview of general pattern detection approaches related to this work is given. Beyond this, algorithms for subgraphisomorphism, needed for the detection of patterns, are named and described.

Chapter 4 – Concept
The third chapter describes the main concepts of the framework and explains the framework's architecture. This includes how the *Cloud Computing Patterns* are modeled and hierarchically arranged.

Chapter 5 – Implementation
This chapter describes details of the implementation. This includes the used data structure needed for the pattern detection, the topologies, and the implemented subgraph isomorphism. With the help of two exemplary topologies, the function of the algorithm is explained in detail. Also, issues that occurred during the development are faced. Furthermore, the restrictions of the framework are mentioned.

Chapter 6 – Conclusion & Outlook
The final chapter summarizes the results of this work and gives a short outlook to possible future work.

2 Fundamentals

In this chapter terms and technologies are described which are essential for this bachelor's thesis. First of all, basics of the TOSCA-Specification are explained, followed by the OpenTOSCA-Ecosystem and the concept of design patterns.

2.1 Topology and Orchestration Specification for Cloud Applications

The TOSCA [OAS13] standard developed by the Organization for the Advancement of Structured Information Standards (OASIS)[1], provides a standard for the description of cloud applications and their management. The aim of TOSCA is to automate the deployment and the management of cloud applications and to improve their portability and interoperability. The overall goal is to offer a vendor-neutral and standardized ecosystem for cloud applications, that does not require any more external software.

Basically, TOSCA consists of two main parts [BBKL16]: the service template, which is graphically modeled by a topology, for the service structure of cloud applications, exemplary shown in Figure 2.1, and the service orchestration for deployment and management. A topology is a graph of typed nodes and directed, typed edges. For each node in a topology exists a Node Template, which is representing and holding information about a single component in a software system (e.g., a web server). Further, the Node Type defines the type of a Node Template (e.g., Apache Tomcat for the Node Template web server). The directed, typed edges are instances of Relationship Templates which are representing the relation or dependencies between two Node Templates. The Relationship Type defines the type of a Relationship Template and holds the information about this relation (e.g., a HostedOn relation for an operating system hosted on some kind of hardware). In addition, each Node Template can hold more information:

- Properties holding specific information like an ip address of a server or login credentials

[1] https://www.oasis-open.org/

- Deployment Artifacts describing how this Node Template is deployed (e.g., as a .war file)
- Management Operations defining input and output parameters plus their data types and Implementation Artifacts which are implementing those (e.g., a REST-service)
- Capabilities and Requirements, which define the need of an operating system for a web server for example

2.1.1 Cloud Service Archive

A TOSCA-based application is packaged in a Cloud Service Archive (CSAR) [BBKL16], which is basically a zip-archive with the ending ".csar". It contains the topology templates, the types with all properties and the management plans and is the standardized packaging format for TOSCA service templates.

2.2 OpenTOSCA-Ecosystem

OpenTOSCA[2] gets developed at the University of Stuttgart and provides an open source ecosystem for the OASIS TOSCA standard. It consists of three main parts, the OpenTOSCA Container, Winery, and Vinothek. Winery is a graphical modeling tool and provides a topology modeler. Vinothek is a web-based self-service portal for end users to instantiate cloud applications [BBKL14b]. The OpenTOSCA Container provides a TOSCA runtime environment.

2.2.1 OpenTOSCA Container

The OpenTOSCA Container provides a TOSCA runtime environment supporting the imperative and declarative processing of TOSCA topologies [BBH+13]. Therefore, the deployment and management logic is provided by plans. Imperative processing means the implementation of management plans can be executed fully automatic, e.g., to start and to terminate an application. These plans can be realized by Implementation Artifacts of Node Templates and Relationship Templates. On the other side declarative processing means the interpretation of the deployment and management logic from plans by the runtime.

[2] http://www.opentosca.org/

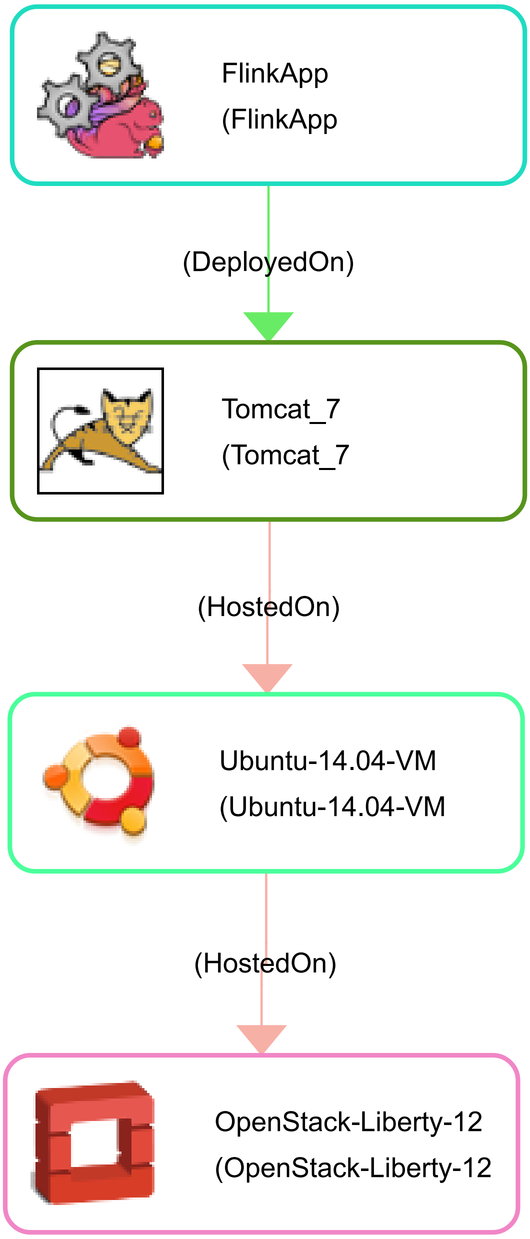

Figure 2.1: Topology of a hosted application

2.2.2 Winery

Winery is an Eclipse project[3] providing a web-based environment to graphically model and create TOSCA applications [KBBL13]. Winery consists of three parts. The first part is

[3] https://projects.eclipse.org/projects/soa.winery

a management graphical user interface (GUI) for Node Types and Node Templates where new types and templates can be created, or existing ones can be edited or deleted, called Element Manager. Secondly, the topology modeler which is providing a GUI to create service templates for a service structure. The last part is the repository, holding the CSAR files and support the import and export of new files. All elements of Winery, such as Node Types or Node Templates, are uniquely identifiable and accessible by URLs to enable a simple way to share topologies. Seven of the 45 elements which are defined in the TOSCA meta model can be used directly for visual topology modeling. These seven elements are Relationship Template, Node Template, Relationship Constraint, Deployment Artifact, Requirement, Capability, and Policy. All remaining elements can be managed in the Element Manager GUI. Doing this, Winery is separating the simple visual modeling of application topologies in the Topology Modeler and the more detailed configuration with more technical insight for experts using the Element Manager. Figure 2.1 shows an example of an application topology, modeled with the Winery. A simple application hosted on a tomcat server which is running on an ubuntu virtual machine. This virtual machine is provided by OpenStack. For this thesis the topology modeler of Winery builds the entry point. All functionality is implemented in the backend of Winery, more details will follow in Chapter 5.

2.2.3 Vinothek

The Vinothek provides a GUI for end users and displays the available and deployed CSARs for easy provisioning. This is an approach to tackle the problem of different management APIs of TOSCA runtimes, which are not standardized by the specification yet [BBKL14b]. Vinothek hides all technical details and only shows a simple, graphical user interface based on web technologies such as HTML5 and JavaScript.

2.3 Design Patterns

As explained in [Gam15] Design Patterns are used to provide a good solution for specific, but returning problems concerning during the development of object-oriented software in an abstract form. To build reusable object-oriented software, it is necessary to solve occurring problems specific for the particular case. Afterwards, the solution has to be abstracted to provide general concepts for future and similar problems. Design Patterns provide possible solutions to design problems and make it easier to reuse successful designs and architectures. In addition, they improve the documentation and maintenance of existing software systems. A single Design Pattern is a conceptual solution, which describes a problem that occurs and then describes a possible solution concept to it. In general, a pattern consist of four essential elements:

- the pattern name
- the problem on which the pattern can be applied
- the solution to the named problem, which doesn't give a particular instruction but an abstract description on how it can be implemented
- the consequences describe the costs and benefits of applying a pattern regarding the whole implementation

As per the design pattern reference book [Gam15], there are 23 design patterns, which can be classified in three categories: Creational Patterns, Structural Patterns and Behavioral Patterns. Creational Patterns abstract the instantiation process and help to make a system independent of how its objects are created, composed and represented. Therefore, these patterns provide a way to create objects while hiding the creation logic. A common known representative is the Singleton pattern, which ensures that there exists only one instance of a class. Structural Patterns concern the class and object composition. They use inheritance to compose interfaces or implementations. An example is the Composite pattern, which enables composite objects. The Behavioral Patterns concentrate on the communication and the assignment of responsibilities between objects. A popular pattern is the Observer pattern, that offers the possibility to add an observer to an object's state. Initially, these three categories of patterns existed. Today, there exists much more patterns, which cannot be classified to one of these categories. For example the Model-View-Controller design pattern, which is very popular. Martin Fowler defines in his book [Fow02] a new category of patterns for object-relational mapping. He defines patterns for the development of enterprise application architectures with object-oriented languages like Java or C#. This work will concentrate on a subgroup of Design Patterns, the *Cloud Computing Patterns*.

2.3.1 Cloud Computing Patterns

With growing amount of applications running in the cloud, several provider of cloud platforms developed Design Patterns for the implementation of applications. To introduce an example, Homer et al. released a book [HSB+14], where they describe *Cloud Design Patterns* especially for the use with Microsoft Azure[4], the cloud platform offered by Microsoft[5]. Those patterns are mainly for the implementation of cloud applications,

[4] https://azure.microsoft.com
[5] https://www.microsoft.com

that will be hosted on the Azure platform. But provider-specific Design Patterns are not suitable for a general use on different platforms. They differ in the naming of services and basic architectural aspects. Therefore, Cloud Computing Patterns on a more abstract level are needed. Fehling et al. deliver this abstraction layer in [FLR+14]. These patterns can be split in five subcategories. The *Cloud Computing Fundamentals* describe cloud service models such as *Infrastructure as a Service* and cloud deployment types like *Public Cloud* or *Private Cloud*. These patterns extend the National Institute of Standards and Technology (NIST) cloud definition [MG11]. The cloud offerings describe how a cloud provider offers resources and functionality for an application. There are cloud environments like an elastic platform, processing offerings, storage offerings, and communication offerings. Cloud application architectures specify how applications should be designed to be best suited adapted for cloud environments. The cloud application management patterns are used for concepts to manage applications in the cloud automatically. Finally, the composite cloud applications cover frequent combinations of patterns from the former described categories. This work will concentrate on an excerption of the *Cloud Computing Fundamentals*, the *Cloud Offering Patterns* and the *Cloud Application Management Patterns*. A more detailed description of each pattern used in this work is given in Chapter 4 .

2.4 Amazon Web Services

Amazon Web Services (AWS)[6] offers reliable, scalable, and inexpensive cloud computing services. Very popular products are Amazon EC2[7] and Amazon Elastic Beanstalk[8]. EC2 provides an *Infrastructure as a Service*, while Beanstalk offers a *Platform as a Service*. Both are very easy to use and an implementation of the relating *Cloud Computing Patterns*. The Amazon Relational Database Service (RDS)[9] offers a selection of the most popular database systems for an easy instantiation. During the development of the Design Pattern Detection Framework, topologies with AWS components where investigated, on how *Cloud Computing Patterns*, like an *Elastic Platform*, are realized in a practical use-case. EC2 is a direct use-case of the *Elastic Infrastructure* pattern and provided as *IaaS*. Amazon Elastic Beanstalk is a *PaaS* implementing the *Elastic Platform* pattern as well as the *Elastic Load Balancer* pattern. Amazon RDS implements the *Relational Database* pattern.

[6] https://aws.amazon.com

[7] https://aws.amazon.com/de/ec2/

[8] https://aws.amazon.com/de/elasticbeanstalk/

[9] https://aws.amazon.com/de/rds/

3 Related Work

As an entry point to the Design Pattern Detection Framework, some other projects and related topics will be presented in this chapter. First of all, two approaches concerning OpenTOSCA are investigated. Then, works related to pattern detection in general are covered. In the third part, an algorithm for subgraph isomorphism is illustrated and various related algorithms are depicted.

3.1 OpenTOSCA

This section gives attention to two approaches in the context of the development of OpenTOSCA.

3.1.1 Manual Pattern Detection

Fehling et al. introduce a research process in [FBBL15] on how patterns in various domains can be identified and organized involving multiple industry partners. This process is splitted in three parts. First of all, the collection of information, where patterns could be detected, is explained. Secondly, the extraction of patterns themselves, and thirdly the application in a concrete use case are declared. The research considered several different domains of patterns. Cloud Computing Patterns, Cloud Data Patterns, Application Management Patterns, Green Business Process Patterns and Costume Patterns. Cloud Computing Patterns as well as the Application Management Patterns are described more detailed in Section 2.3.1. The pattern identification and pattern authoring are performed in iterations and repeated as long as new patters are found in the considered domain. During the pattern identification phase, relevant information of a domain, in which patterns shall be found, is collected and structured. As a result, a set of existing solutions is created. During the pattern authoring phase, the patterns are written. Therefore, the right design for the pattern language based on the considered domain is chosen and the pattern is written. Subsequently, it is getting reviewed during further iterations by other pattern authors. The third phase can be independent from the previous two: the pattern application. In this phase, the patterns are used and implemented during the development and the design of applications.

Relevance

This works demonstrates a process, how to manually detect patterns in various domains. In this case, new patterns are searched, different from the topic of this work, to detect known patterns in a structure. Therefore, it is not relevant for the current work. But it could be an improvement for the *Design Pattern Detection Framework*, if the process can be automated and included in the framework. The detection algorithm would also be able, to detect new, not yet existing patterns.

3.1.2 Architectural Pattern Language

Another approach to reduce the complexity of cloud application architectures is an architectural pattern language, also developed by Fehling et al. in [FLR+11]. This language describes the principals of cloud computing, available cloud offerings, and cloud application architectures. The aim is to guide developers during the process of locating the best cloud environment and the most applicable architecture for their problems. This process enables a pattern-based application development through interrelation of patterns. To ensure, that the recommendations for patterns which could be used are meaningful, a decision recommendation table was designed. This table is showing well-arranged the interrelations of cloud computing patterns and contains three different types of pattern interrelations:

- strong cohesion relation (+)
- exclusion cohesion relation (-)
- undetermined relation (o)

The strong cohesion means that one pattern is strongly combined with the related pattern. The exclusion cohesion relation mentions that those two patterns cannot be combined. That means, if one pattern appears in an application, it is impossible to realize the other one. The third one is the undetermined relation, which declares that there is neither a strong cohesion nor a exclusion between those two patterns. This is an indication for patterns that are used for different tasks. The table was formed based on undetermined relations between all patterns.

Relevance
This approach deals with the opposite of the main topic of this thesis. Instead of detect patterns in a completed application, an application should be implemented based on the selection of needed patterns. For this thesis the decision recommendation table was useful to learn about the interrelations of the *Cloud Computing Patterns*. Based on this table, a pattern taxonomy was build for defining the probability for the detection of a specific pattern. This will be declared more detailed in Chapter 4.

3.2 General Pattern Detection

Pattern detection is not just a software engineering related subject, but a very large field in various domains. In this section, pattern detection approaches in different domains are introduced.

3.2.1 Pattern Recognition and Machine Learning

Yuichiro Anzai describes in his book [Anz12] general patterns and their recognition by a computer. He splits up the recognition in two parts: the pattern recognition itself and machine learning. Anzai uses patterns of real objects, such as pictures. To recognize the pattern of an object, the computer has to find the boundary lines between the object and the environment or possible other objects. But when the computer is able to distinguish between the object and the rest, he still does not know, what kind of object it is. For the recognition of the object, conceptually knowledge about the object, i.e, its pattern, is needed. Anzai distinguishes between two algorithms that are necessary: an algorithm for the detection of an object and an algorithm for recognizing it based on patterns. During pattern recognition, it is often not possible to determine only one solution based on given information. So the computer has to deal with possible different sizes of objects that have the same pattern. Therefore, the computer has to generate new data from the given data he already recognizes. This process is called machine learning.

Relevance
The work of Anzai describes a general way to pattern recognition. This concept confirms the approach of this thesis having also two algorithms. One for the detection of the subgraph isomorphism and one for the recognition of pattern graphs.

3.2.2 Statistical Pattern Recognition

Webb [Web03] focuses on basic pattern recognition procedures with practical applications on real-world problems. In his approach a pattern denotes a p-dimensional data vector of measurements. Those measurements are features of an object that is described by this pattern. Webb distinguishes between supervised and unsupervised classification of patterns. Supervised classification has a set of data samples with class types, which are used as exemplars. Unsupervised classification means, the data samples have no class types and to distinguish one group from another, clustering techniques are used. The main problem faced in this work concerns the classifier design, the creation of classes for patterns. Given a set of measurements represented as a pattern vector, this pattern should be assigned to possible classes. On one side, there is an approach to assume a knowledge of the underlying class-conditional probability density functions. This means, for each pattern vector the probability, to be in a specific class, is known. But in most cases these information will be unknown and must be estimated from a set of correctly classified samples. The second approach develops decision rules which separate the measurements of a pattern vector into regions. These regions get classified and can belong to one or more classes.

Relevance

This approach of Webb is based on statistics and probability density function. As a part of the Design Pattern Detection Framework, probabilities for possible patterns based on already detected patterns are set. Although, the approach in the current thesis is very elementary, this could be an inspiration for a more precise prediction of possible patterns.

3.3 Algorithm for Subgraph Isomorphism

The subgraph isomorphism problem is a task, in which an input consists of two graphs G and H and one must determine whether G contains a subgraph that is isomorphic to H. A subgraph is defined as a graph, that contains a subset of vertices and edges of the main graph. Two graphs are isomorphic, if they have the same structure, but it is not forced that their vertices have the same identifier. The subgraph isomorphism is NP-complete. But there are cases where it may be solved in polynomial time. In the use case of this thesis it is assumed, that the graphs are finite and small. In the following, an algorithm is explained as well as some related algorithms.

3.3.1 Ullman

Ullman proposed a backtracking algorithm for the subgraph isomorphism problem in 1976 and established a basic knowledge for further algorithms facing this problem [Ull76]. The algorithm basically eliminates inferentially successor nodes in the tree search. Ullman uses matrices, to model all possible matches between the vertices of a graph and a possible subgraph. Possible mappings are marked with a 1. By systematically enumerating all possible matrices, the algorithm checks whether they encode an isomorphism. To reduce the computation time, a pruning method is used. This method compares the neighbors of the vertices of a mapping pair, if they can be mapped to each other too. If no neighbors can be mapped, the mapping pair is wrong and gets eliminated. This is repeated recursively until no more changes are possible.

Problem definition

Given two graphs G and H, the subgraph isomorphism problem is to determine, if a data graph G contains a subgraph, that is isomorphic to a query graph H. Subgraph is hereby called a graph, that contains a subset of vertices and edges of the main graph.

The basic algorithm

First of all, a $|V_P| \times |V_G|$ matrix M^0 is set up, where P and G are graphs. V represents the set of vertices of each graph. For every entry m^0_{ij} in M^0, the degree of the vertices is used as criterion. If $deg(v_i) \leq deg(v_j)$, the entry at this point is set to 1. This proofs, that v_i can be mapped to v_j, because the latter needs equal or more neighbors. Now all matrices M that can be obtained from M^0 by removing all but one 1 from each row while having at most one 1 in each column must be tested for the possibility to be a subgraph isomorphic to P. Removing all but one 1 from each row is needed, because in one possible isomorphism every vertex of the subgraph can only be mapped to one vertex in the target graph. Having at most one 1 in each column is necessary, because it is not possible to map multiple vertices to the same vertex in the target graph. In Algorithm 3.1, which represents the algorithm in pseudo-code, this is realized in the loop. The recursive function checks, if the current row is equal to the total amount of rows. If yes, every vertex of the subgraph is mapped to a vertex in the target graph. Then, it is checked if this is an isomorphism.

To reduce the computation time, a pruning procedure is used. The pseudo-code for this procedure is shown in Algorithm 3.2. This offers a simple observation, to check for the neighbors of an entry in M. An entry m_{ij} in M with a 1 means that $v_i \in P$ can

Algorithmus 3.1 Ullman's Algorithm

```
function RECURSE(used_columns, cur_row, G, P, M)
    if cur_row = num_rows(M) then
        if M is an isomorphism then
            output yes and end the algorithm
        end if
    end if
    M' = M
    prune(M)
    for all unused columns c do
        set column c in M' to 1 and other columns to 0
        mark c as used
        recurse(used_column, cur_row+1, G, P, M')
        mark c as unused
    end for
    output no
end function
```

be mapped to $v_j \in G$. But if the algorithm detects that any neighbor of $v_i \in P$ cannot be mapped to any neighbor of $v_j \in G$, the 1 set at m_{ij} is clearly wrong and can be changed to 0. If this happens, recursively all other changes that might depend on the previous modification will be indicated and changed as well. The effectiveness of the pruning procedure depends on the order of the vertices. So a reordering of vertices by descending degrees would be optimizing the computation time.

Algorithmus 3.2 Pruning Procedure for Ullman's Algorithm

```
function PRUNE(M)
    while M was changed do
        for all (i,j) where M is 1 do
            for all neighbors x of $v_i$ in P do
                if there is no neighbor y of $v_j$ s.t. M(x,y)=1 then
                    M(i,j)=0
                end if
            end for
        end for
    end while
end function
```

3.3.2 Related algorithms

Existing algorithms can be classified in two categories. The first category are algorithms for exact subgraph matching and the second one are algorithms for approximately subgraph matching. In a recent survey [LHKL13] about the current state of art in subgraph isomorphism, the authors name five recent algorithms. They represent several optimization techniques for the legacy algorithm of Ullman. Therefore, they use different join orders, pruning rules, and auxiliary information to eliminate false-positive candidates as early as possible for increasing the performance [LHKL13]. They belong to the first category and will find all possible subgraphs. As a second subcategory, there exists also indexing algorithms which can detect if there is one subgraph isomorphism. This subcategory will not be introduced, because the main goal is to detect multiple matching subgraphs.

VF2

VF2 is an optimized algorithm for graph matching and is able to efficiently solve the isomorphism and the graph subgraph isomorphism problem [CFSV04]. The used memory requirements are reduced significantly by optimizing the exploration of the search space. Supposing that one has two graphs, a data graph H and a query Graph G, VF2 selects the first vertex of G and maps it to a vertex in the data graph. Then the algorithm tries to map a neighbor of the previous vertex to a neighbor of the vertex in the data graph and repeats this recursively. In the case of no possible matching, the algorithm goes one step back and selects another neighbor. Since ten years, VF2 is the state of art algorithm to solve the subgraph isomorphism problem and commonly used in different applications. This algorithm was used for the implementation and will be described in more detail in Chapter 4.

VF2 Plus

VF2 Plus is an improved version of the VF2 algorithm especially designed for large graphs in bioinformatics applications [CFV15]. It improves two important weaknesses of VF2: the total order relationship and the structure of the terminal sets. VF2 does not support a sorting procedure and takes the original order of nodes in a graph. VF2 Plus uses a sorting procedure to find a candidate with the lowest probability to find a matching candidate on the target graph and the highest number of connections to nodes, that are already used by the algorithm. VF2 just analyzes the direct neighbors of a node for a new candidate pair. VF2 Plus improves this by exploring nodes that are mapped to a neighbor of the target node. It also uses a classification system to divide the terminal sets in different subsets. Due to the optimization for biological graphs, the VF2 Plus was not considered for the application with normal graphs in this work.

QuickSI

QuickSI was originally designed for smaller graphs. It preprocesses graphs to compute the frequencies of vertex labels and how often a triple (consisting of a source vertex label, an edge label, and a target vertex label) appears [SZLY08]. Then a minimum spanning tree is created, called QI-Sequence [SZLY08], by weighting the edges accordingly (as the first vertex the one with a higher frequency is used) and the weights are used to order them in a minimum spanning tree. QuickSI is then searching for a subgraph using this minimum spanning tree. Regarding the performance, QuickSI and VF2 are mainly equal. Both are developed as improvements of Ullman's algorithm. The decision between QuickSI and VF2 is made for VF2, because of the more common use as reference algorithm for subgraph isomorphism.

GraphQL

GraphQL is a graph query language [HS08] and as well as the following two algorithms designed to handle large graphs. It uses neighborhood signatures of data vertices to prune the initial candidate set and a pruning technique called pseudo subgraph isomorphism to globally narrow the search space. Pseudo subgraph isomorphism works by creating a bipartite graph between the query graph and its potential matches in the data graph. Then it is iteratively comparing subtrees of greater height until it reaches a specified depth. This algorithm is not used, because it is designed for the use with large graph databases and ships with an own graph algebra. For this work, small graphs with simple nodes and edges are used. An algorithm with this level of complexity is not necessary.

GADDI

GADDI indexes a data graph based on a neighborhood discriminating substructure (NDS) [ZLY09]. Between pairs of neighboring vertices, NDS is a distance and is handled as a subgraph. GADDI performs a depth first search to find the next vertex for comparison. After each run, the vertices are pruned using NDS. This algorithm struggles with a poor performance due to the expensive NDS distance calculation. Therefore, it was not considered for this work.

SPath

SPath is defined as a high performance graph indexing mechanism [ZH10]. SPath focuses on creating a set of shortest paths in the query graph and a path-based indexing technique for the data graph. Each vertex of the indexed data graph has a neighborhood signature, which holds information about the shortest paths within the vertex's vicinity. Its performance depends very hard from the path search order. Due to its large neighborhood signature overhead, the performance of SPath is poorer compared to GraphQL. GraphQL is explained to be not considered for the use in this work, so an algorithm with an even poorer performance is not be used either.

4 Concept

In this chapter the underlying concept of the Design Pattern Detection Framework is explained. Further, the realization of modeling *Cloud Computing Patterns* using TOSCA topologies is explained.

4.1 Basic Idea

The general idea is to detect *Cloud Computing Patterns* in a given TOSCA topology. Those topologies represent the structure of cloud applications and they can be very complex. Patterns are not forced to be implemented, so a cloud application may exist without realizing any patterns. Typically, most cloud applications can implement multiple *Cloud Computing Patterns*. To enable the detection programmatically, a consistent structure for TOSCA topologies and *Cloud Computing Patterns* is necessary, since the structures need to be compared accordingly. TOSCA topologies are basically represented as directed graphs, hence it is useful to model patters as directed graphs as well. Due to the possibility, that a topology can implement more than just one pattern, a pattern can be represented in a fragment of a topology. A fragment in a graph is equal to a subgraph, and therefore the algorithm has to search for matching subgraphs in the topology. Each subgraph may represent a pattern. For the detection algorithm, which needs to compare multiple subgraphs, i.e., the patterns, with the topology and its subgraphs, a subgraph isomorphism algorithm is needed. The chosen algorithm is explained in Section 4.7.

4.2 Abstract Components

Although, each cloud application uses different architectures and different components, it is possible to generalize those used components. On base of multiple topologies, the following general components of cloud applications are abstracted. These resulting components represent the superclasses for the used Node Templates in TOSCA topologies. These components are used as nodes in pattern graphs and as labels in TOSCA topologies.

- Application components represent one instance of an application.
- Service components are the superclass for resources like Java or Python.
- Storage represents all kind of databases and database management systems.
- Messaging unites message broker and topics.
- Server components stand for web servers like Apache Tomcat[1], Apache Server[2], or JBoss[3].
- Operating System component represents an instance of a virtual server with a running operating system.
- Virtual Hardware represents the abstracted, virtual hardware, where the virtual servers are hosted on. Examples therefore are OpenStack[4] or VSphere[5].

4.3 Concept to Model Patterns

Cloud Computing Patterns are described in Chapter 2.3.1 and represent abstract and general concepts to provide good solutions to reoccurring problems. Therefore, a real implementation of theses patterns is always restricted to the particular use-case and the used technologies. To be able to use patterns in a context with TOSCA topologies, it is necessary to model them in a suitable structure. As mentioned above, directed graphs are chosen for this. Every node of such a graph represents an abstract component of cloud applications. The concept is designed for the automatic detection of *Cloud Computing Patterns* in TOSCA topologies. Because of the limited information available through the topologies, e.g., there are no details about the implementation or the architectural design of a component itself, not all *Cloud Computing Patterns* can be detected by this algorithm. To form a fundament, the concept and the implementation concentrate on the following *Cloud Computing Patterns* extracted from [FLR+14]. The modeled pattern graphs do not represent full topologies, but subgraphs which can be a fragment of a respective topology.

[1] https://tomcat.apache.org/
[2] https://httpd.apache.org/
[3] https://www.jboss.org/
[4] https://www.openstack.org/
[5] https://www.vmware.com/products/vsphere.html

4.3.1 Cloud Computing Fundamentals

The *Cloud Computing Fundamentals* describe the fundamentals to understand the most of the other patterns. They describe how IT resources are provided and used. In this work, the following patterns of the subcategories *Cloud Service Models* and *Cloud Deployment Models* are faced. These patterns are not modeled as graphs. *Infrastructure as a Service* (IaaS) and *Platform as a Service* (PaaS) are detected by analyzing the structure of the TOSCA topology. The *Cloud Deployment Models* are detected regarding the use case of a cloud application. They only differ only in the size of their user groups and their accessibility.

Infrastructure as a Service

Using the *IaaS* pattern, providers offer physical and virtual hardware, such as servers, storage, and networking infrastructure [FLR+14], which can be provisioned quickly. Those IT resources can be used by customers to install individual operating systems, middleware, and application software. Many cloud providers also support automatic scaling of virtual machines. In the context of this work, just virtual hardware is used, because no physical hardware is modeled using topologies. The *IaaS* pattern is only detected, if there is virtual hardware, but no more components such as services or applications.

Platform as a Service

The *PaaS* pattern builds up on the *IaaS* pattern. It offers a new layer upon the infrastructure by providing managed operating systems and an application hosting environment, also called middleware. Using this service model, the customer only has to deploy his applications, the rest is managed by the cloud. This pattern is used for topologies, as soon as a layer above the virtual hardware or an operating system with services and servers is detected.

Public Cloud

IT resources or an application running on those, is accessible by a very large customer group and has public access.

Private Cloud

The resources, that are provided, are only accessible for an exclusive group of users, to meet a higher grade of privacy and security.

Community Cloud

Community means a group of customers, who trust each other and enable a collaborative and elastic use of IT resources.

Hybrid Cloud

A *Hybrid Cloud* combines multiple of the former described *Cloud Deployment Models* to form a homogeneous hosting environment. Different hosting environments are provided, that can be accessed by different numbers of users and the underlying IT resources can be shared between different requirements.

4.3.2 Cloud Offering Patterns

These patterns model the different offerings of functionality, which are provided to the customer by cloud offerings.

Elastic Infrastructure

An *Elastic Infrastructure* provides pre-configured virtual server images and storage. The customer is able to create individual server images for his applications based on pre-configured images and can add necessary storage resources. All the functionality is offered via a self-service interface and provides also monitoring information. The elastic infrastructure enables dynamic allocation of new virtual servers and storage. The use of an *Elastic Infrastructure* is detected with keywords, e.g. with an EC2 instance.

Elastic Platform

An *Elastic Platform* offers a middleware for the execution of custom applications, their communication and data storage. Middleware is a bundle of services and shared by different applications on the same host system to reduce management effort. Also the use of this pattern is detected with keywords, e.g. an Amazon Beanstalk.

Node-based Availability

A cloud provider guarantees the availability of individual nodes, such as virtual servers, middleware components, or hosted applications. A node is defined available if it is reachable and performing the expected functions. In the context of TOSCA topologies, the *Node-based Availability* is mainly paired with hosted applications. Therefore, this pattern requires applications to be hosted on different virtual machines. In Figure 4.1, two applications are hosted on two different virtual servers. One application is independent from the underlying infrastructure of the other application. In case of a failure of one virtual server, the other application is still running. This indicates a *Node-based Availability*.

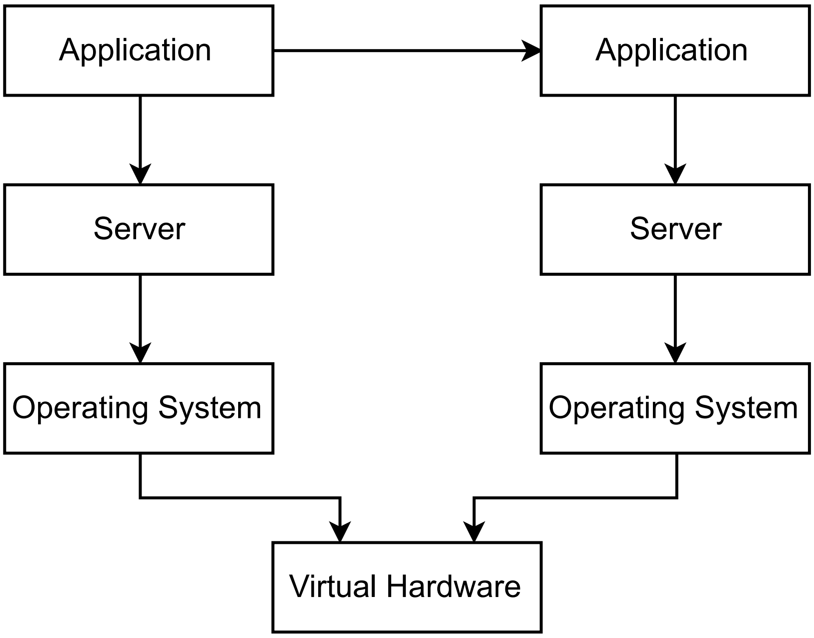

Figure 4.1: Node-based Availability Pattern

Environment-based Availability

A cloud provider guarantees the availability of the environment hosting individual nodes, such as virtual servers, middleware components, or hosted applications. For example, a provider offers an elastic platform to which customers may deploy application components. Then the provider has to ensure the availability of this environment. There is no knowledge about the availability of individual nodes in this environment, but the overall set of deployed nodes. For an *Environment-based Availability*, the only constraint is, that all server, services, applications, etc. are deployed and hosted on one virtual machine. The provider guarantess the availability of the virtual machine and therefore the availability of the whole environment. This virtual machine is represented as an operating system in Figure 4.2.

Execution Environment

The *Execution Environment* pattern describes a concept to avoid duplicate implemen-

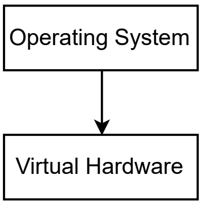

Figure 4.2: Environment-based Availability Pattern

tations of functionality. Application components, which are using similar services and functionalities, are deployed to an environment, which provides a middleware offering these services. This enables an efficient sharing of a hosting environment, because not for every application a new environment with the needed services is build. In Figure 4.3, the graph for this pattern is modeled. The execution environment itself consists of a bundle of services and servers, This environment is located as middleware between applications and the operating system. Another constraint is the existence of more than one application, otherwise there exists no sharing. All components have to be hosted on the same operating system, which is equal to the same virtual machine.

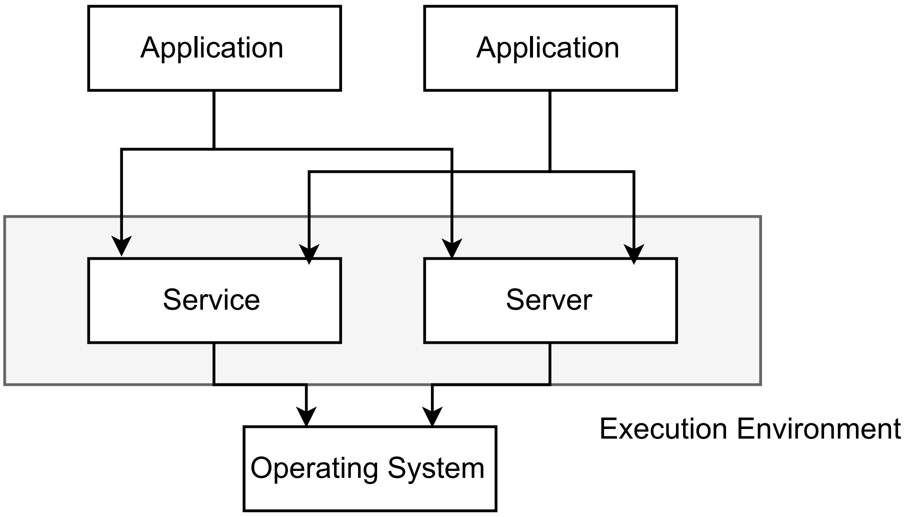

Figure 4.3: Execution Environment Pattern

Message-oriented Middleware

A message-oriented middleware provides asynchronous message-based communication. Therefore, message queues are used to exchange information asynchronously and it fits best, if the transferred amount of data is small. The structure of a message-oriented middleware requires a broker. The broker is hosted on an operating system. Upon the broker, a topic is hosted. This topic references a queue. An application can send and receive messages to and from this queue. This is modeled with the relation between the application and the messaging component as shown in Figure 4.4.

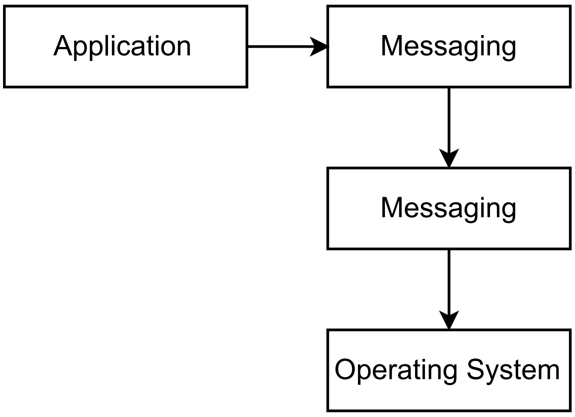

Figure 4.4: Message-oriented Middleware

Relational Database

The cloud provider offers storage in the form of relational databases for data handling, for example, a structured query language (SQL) database. For the modeling of this pattern it is assumed, that a database is set up on a database management system. Therefore, two components are used. The pattern graph is shown in Figure 4.5.

4.3.3 Cloud Application Management Patterns

These architectural patterns describe how cloud applications can be managed automatically by separate components. Such components handle the automated management of cloud applications regarding dynamic elasticity, resiliency, updates, etc.

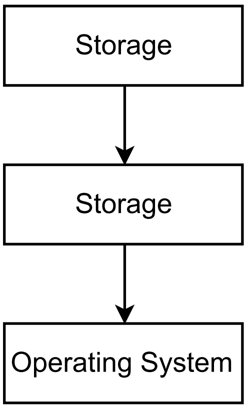

Figure 4.5: Relational Database Pattern

Elasticity Manager

The *Elasticity Manager* is a component, to elastically scale-out the number of required application component instances. The manager works as a monitoring tool and if more instances are required, for example, because of a higher demand, new instances will be deployed. In Figure 4.6, the manager component is modeled as a service on the virtual hardware layer.

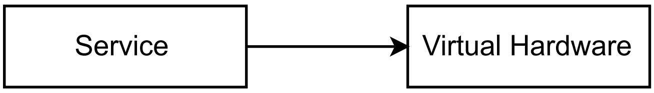

Figure 4.6: Elasticity Manager Pattern

Elastic Queue

To use an *Elastic Queue* in context with a cloud application, it is necessary to provide a *Message-oriented Middleware*. This middleware provides queues, which are distributing asynchronous requests among multiple instances of application components. Paired with an elastic queue, which monitors these queues, the required amount of instances for application components can be scaled elastically. As shown in Figure 4.7, this pattern is modeled with a messaging component above the applications. This messaging component represents the queues. To enable automatic scaling, a manager component on the layer of the virtual service is also necessary.

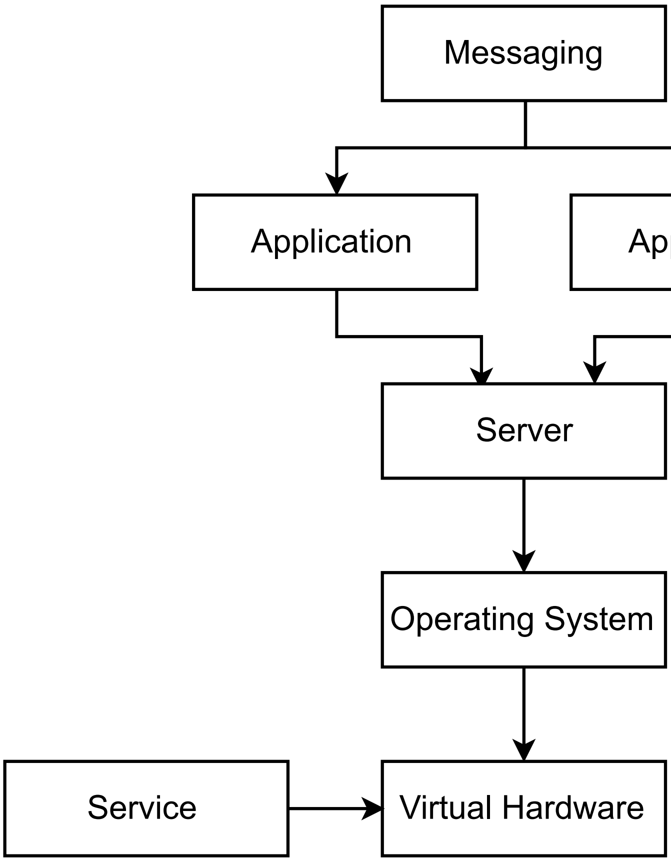

Figure 4.7: Elastic Queue Pattern

Elastic Load Balancer

An *Elastic Load Balancer* is a management component that is provided with information from a load balancer. A load balancer's task is to distribute requests among multiple instances of application components. To be elastic, this function is extended with a monitoring component, to automatically scale the required number of those instances. In a topology, this pattern is modeled as a service above the applications as shown in Figure 4.8. This service has ConnectsTo relations to the applications and can be identified as a LoadBalancer. The service on the layer of the virtual hardware is an elasticity manager, to enable the automatic scaling.

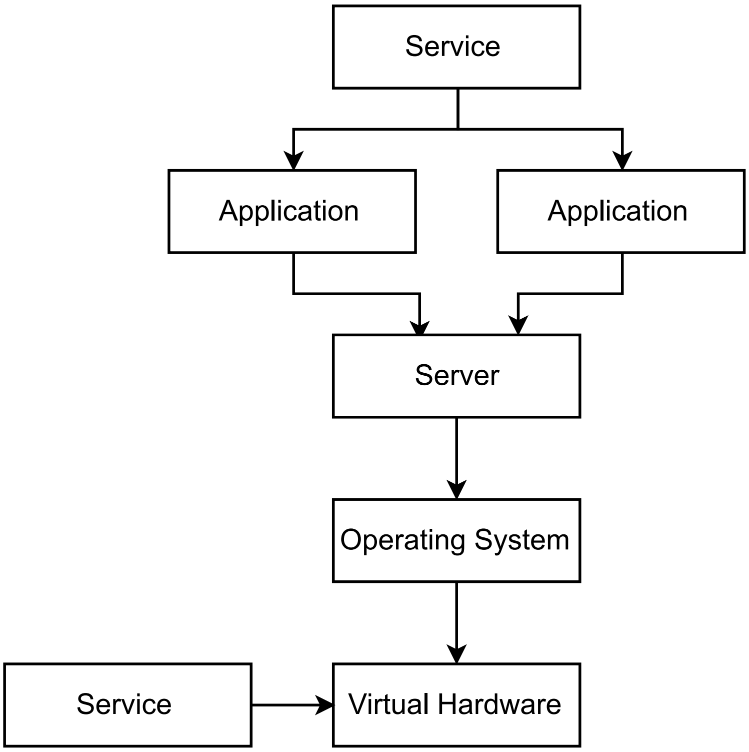

Figure 4.8: Elastic Load Balancer Pattern

4.4 Mapping of TOSCA Topologies

TOSCA-Topologies represent the input for the pattern detection algorithm. To be able to compare them to the *Cloud Computing Patterns*, it is necessary to know what each node of the topology represents. Therefore, the topology will be labeled with the same general component names as the modeled patterns. The first step of labeling a topology is made during the keyword search. If a node is detected in the keywords, it is labeled accordingly. The second step is using the present labels and the relationships between the single nodes of the topology. There are currently three relevant types for relationships:

- DependsOn: This relation is used to define a dependency between two nodes. The source node of this relationship is the dependent component. The target node represents the needed resource. For example an Apache Tomcat server (source) is dependent on a Java Runtime Environment (target).
- DeployedOn: Using this relation shows that an application will be deployed on a server.

- HostedOn: Services like Java, Python, etc. use this relation to identify their host system, for example, a virtual machine. Also database management systems that run on a database server use this relation.
- ConnectsTo: Relation for modeling the connection of applications to a service like a database or a message broker. The source node is the requesting node, the target node the offering service node.

To start the mapping, the lowest node of the topology is set as the starting point. The lowest node in this case is defined as the node with no outgoing relations. Outgoing from the this node, the topology gets labeled by backtracking all incoming relations of each node. Every node gets only one label and each label represents one of the abstract components, which are defined in Section 4.2. Therefore, constraints are defined. Figure 4.9 shows the possible incoming relations of a node. Nodes with a label are marked with a gray background. Following the incoming relation, the next node can be labeled. Some nodes have more than one incoming relation. If they cannot be distinguished by the type of the relation, all possibilities are saved and the next node is checked. This concept of mapping is strongly dependent from the collaboration with the keyword detection. In common use cases, the most types of servers, operating systems, messaging, and storage components are detected using keywords. In cooperation with this mapping concept it is ensured that all nodes get labeled. In the special case that a node could not be labeled, all possible labels according to Figure 4.9 are saved. If a pattern matches a subgraph containing this unidentified node, the two graphs will be compared using all combinations of different labels for this node. If they match, the combination is paired with the possible pattern and is added to the possible pattern list. For example, if the lowest node is detected and labeled as virtual hardware plus the incoming relation is a HostedOn relation, the source node has to be an operating system. Otherwise, if the lowest one would be identified as an *Amazon Elastic Beanstalk* component with an incoming DeployedOn relation, this indicates an application component and will be labeled with application.

4.5 Pattern Taxonomy

Based on the decision recommendation table described in Section 3.1.2 pattern taxonomies for *PaaS* and *IaaS* were created, because all cloud applications are underlying one of these business models. Working with such taxonomies reduces the runtime of the subgraph isomorphism algorithm, because impossible patterns can be detected and removed, before the subgraph isomorphism algorithm starts. In addition, probabilities can be calculated suggesting other patterns after the detection of a specific pattern, often there is no hundred percent probability for the existence of a pattern. Starting from one

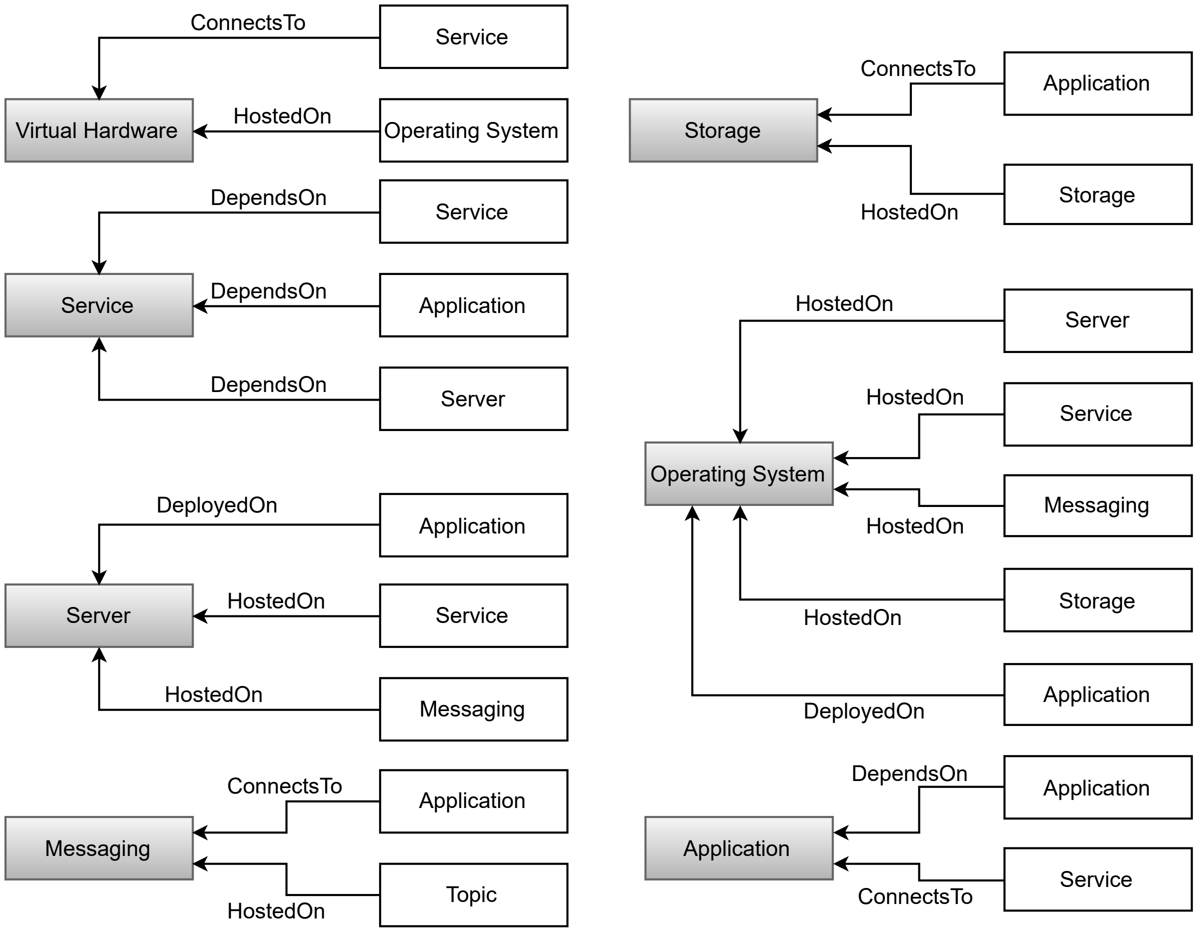

Figure 4.9: Constraints for mapping of TOSCA topology

of the *Cloud Service Models* as initial position, related patterns are arranged in a graph structure, depending on their interrelations defined in [FLR+14]. The graph cannot be interpreted as a binding statement for pattern detection, but as a hint for possible patterns based on previous detected patterns.

4.5.1 Infrastructure as a Service Taxonomy

Figure 4.10 shows the pattern taxonomy for *IaaS*. According to [FLR+14], the usage of the *Environment-based Availability Pattern* is very often found in public clouds. Therefore, the *Public Cloud* pattern was added to both taxonomies. The other *Cloud Deployment Patterns* cannot be detected from the context of other used patterns, but regarding the whole topology and analyzing the use case. All three *Cloud Management Components*, such as *Elastic Queue*, need an underlying *Elastic Infrastructure* to be realized. The

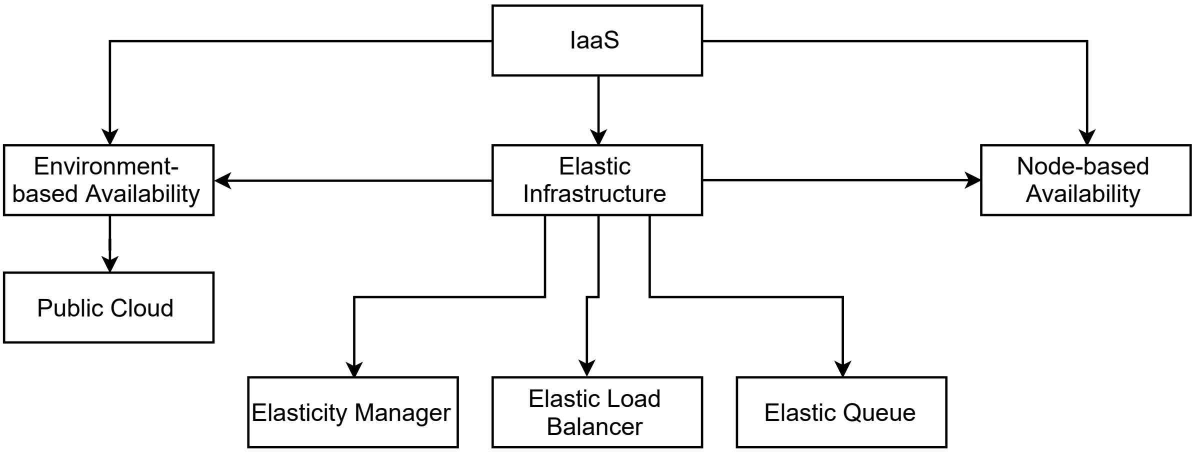

Figure 4.10: Taxonomy for IaaS model

Node-based Availability and the *Environment-based Availability* patterns can be used in a standalone application structure without an *Elastic Infrastructure*, but also together with this pattern.

4.5.2 Platform as a Service Taxonomy

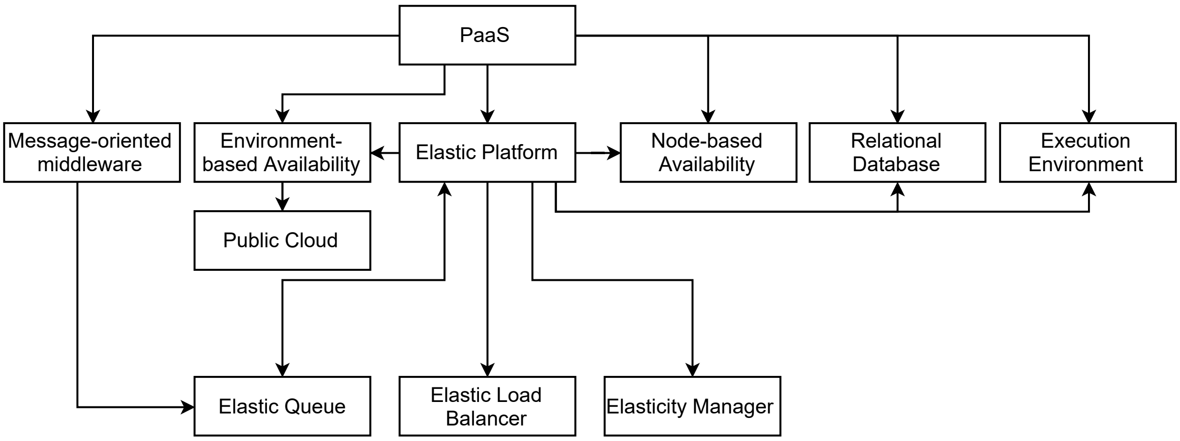

Figure 4.11: Taxonomy for PaaS model

Figure 4.11 shows the pattern taxonomy for *PaaS*. The *Node-based Availability* pattern, the *Relational Database* pattern, the *Execution Environment*, and the *Environment-based Availability* pattern can be used in both ways again, in a standalone application structure or in the context of the use with an *Elastic Platform*. The *Public Cloud* pattern is also present here for the same reason as explained in the pattern taxonomy for *IaaS*. As a special characteristic in this taxonomy, the two-way directed arrow between *Elastic Platform* and *Elastic Queue* must be mentioned. When a *PaaS* is detected in combination with a *Message-oriented Middleware*, there is the possibility for an *Elastic Queue*. If this is confirmed, there must be an *Elastic Platform*. Therefore, the arrow must point bidirectional. The patterns *Elasticity Manager* and *Elastic Load Balancer* need an underlying *Elastic Platform*.

4.5.3 Software as Service

The Software as a Service (SaaS) model was not considered, because the topology for a cloud application with this underlying cloud service model would only consist of one component, the software or application component itself. Also it is necessary, to differ from topologies with application component nodes. Those topologies are usually modeled as *PaaS* systems, but they also contain application component nodes to model their future deployment.

4.6 Algorithm

The main algorithm for the pattern detection is divided in three steps. At the beginning, a simple keyword search is executed. After a successful detection of some keywords, probabilities for patterns, that may be used in the topology, are set based on the pattern taxonomies. Afterwards, the TOSCA topology is mapped to a graph. As the third step, a search for isomorphic subgraphs in the topology is operated. In some topologies there is the possibility of both cloud service models *IaaS* and *PaaS*. This can appear, if the lowest node is, for example, an OpenStack. OpenStack offers virtual hardware, but if upon this node a *PaaS* is modeled, the algorithm will detect both. OpenStack then delivers an *Elastic Infrastructure* and above it, the platform is modeled.

Keyword Search & Probabilites

Keywords are predefined and searched in the name property of every Node Template of the topology. These keywords are used to identify specific components, e.g., Amazon

Elastic Beanstalk [6] or an Apache Tomcat server. If Amazon Elastic Beanstalk is detected in a topology, it is ensured that a *PaaS* is offered, because it is a *PaaS* application management service. Furthermore, the patterns *Elastic Platform* and *Elastic Load Balancer* can be derived to be implemented as well. This can be ensured, because of the knowledge about the occurrence of the Amazon Elastic Beanstalk service. Now those patterns are marked as detected in the pattern taxonomy. Probabilities are set specific for each pattern based on detected keywords. If a pattern was recognized with keywords, the probabilities for all direct successor nodes in the taxonomies are set. In case of a probability that was already set for a node, it will be just overwritten if the current probability is lower. The following layers of probabilities are used in this descending hierarchy:

1. Detected
2. High
3. Medium
4. Low
5. Impossible

Mapping of Topology

In this step of the algorithm, the topology is mapped to a graph, to be comparable to the pattern graphs. This is done by implementing the concept for mapping TOSCA topologies explained in Subsection 4.4.

Subgraph Isomorphism

For the subgraph isomorphism, all possible subgraphs of the topology graph are created. According to the pattern taxonomy, the possible pattern graphs are tested for subgraph isomorphism in each subgraph of the topology using the VF2 algorithm. The result set contains those pattern graphs, that are isomorphic to a subgraph. Now each pattern graph is compared to its isomorphic subgraph. This has to be done, because the proof of subgraph isomorphism does not proof the equal structure of components of the two graphs. This is done afterwards, by comparing the labeled nodes. If they are matching too, the pattern is added to the detected pattern list.

[6] https://aws.amazon.com/de/elasticbeanstalk/

4.7 VF2 Algorithm

The VF2 algorithm is an improved version of the VF and Ullman's algorithm and optimized for large graphs. VF2 was developed to efficiently solve the graph isomorphism problem [CFSV04]. Supposing that one has two graphs, a data graph H and a query Graph G, VF2 selects the first vertex of G and maps it to a vertex in the data graph. In the next step, VF2 selects a vertex that is connected to the already matched vertices in G and tries to map it to an also connected vertex in H. This is repeated recursively until all vertices of G matched a vertex in H or one vertex cannot be matched. In this case, the algorithm goes one step back and selects another connected vertex. If this one matches, the algorithm continues as known, otherwise it goes one step back. The performance is significantly dependent from the choice of the next vertex.

Algorithmus 4.1 Matching algorithm

```
Input: an intermediate state s; the initial state s0 has $M(s0) = \emptyset$
Output: the mappings between the two graphs
function MATCH(s)
    if M(s) covers all the nodes of $G_2$ then return M(s)
    else
        Compute the set P(s) of the pairs candidate for inclusion in M(s)
        for all $(n, m) \in P(s)$ do
            if F(s,n,m) then
                Compute the state s' obtained by adding (n,m) to M(s)
                MATCH(s)
            end if
        end for
        Restore data structures
    end if
end function
```

In the pseudo-code algorithm, shown in Algorithm 4.1, G_2 is the data graph and G_1 the subgraph that should be isomorphic. M(s) represents a partial mapping solution, which contains only a subset of the components of the whole mapping function M. The set P contains all candidates that must be investigated for a matching. The function F returns true, if it is ensured that (n,m) matched and was added to the partial isomorphism.

5 Implementation

In this chapter the implementation details of the framework are described and two examples are explained. For further usages the occurred problems during the implementation plus the actual limitations are named.

5.1 Used Technologies

In this section the used technologies and libraries are described which are used to implement the design pattern detection framework.

5.1.1 Winery

The framework is realized as an extension for Winery. In the graphical user interface itself, the amount of development effort was small and will not be treated in detail. To be able to use the pattern detection, a button with the label "Detect Pattern" is added and connected with the REST-API of the Winery repository. The whole logic behind the detection is implemented in the repository using Java and is located in the Winery repository module in the package "org.eclipse.winery.repository.patterndetection".

5.1.2 JGraphT

For the implementation of the graphs for the patterns and the TOSCA topology, the JGraphT [1] library is used. It is a free Java graph library for simple use of different graph types, like directed, undirected, as well as weighted, unweighted, or user-defined edges. JGraphT focuses on data structures and the use with algorithms. JGraphT is

[1] http://jgrapht.org/

dual-licensed under LGPL [2] and EPL [3]. Each graph consists of a set of vertices and a set of edges. The vertices can be any kind of Java classes.

Relationship Edge

To be able to use labeled edges, the default edge class which is providing the source and the target vertex, is extended by a label, represented by a single String. This label defines the Relationship Type. The different types are named in Chapter 4 in Section 4.4.

5.2 Code Structure

The main class is the Detection class, where the detection algorithm starts. The model package contains the patterns and all classes used to model the graphs. In the pattern package all patterns are created as java classes. In the keywords package, enums are created for each abstract component. Each of these enums is filled with common used keywords.

5.2.1 Keywords

There are five enums for the keywords. Each enum realizes one abstract component as described in Section 4.2. These enums are imported and used as ArrayLists.

5.2.2 TNodeTemplateExtended

During the pattern detection, the original TOSCA topology gets labeled to assign each node to a layer. Therefore, objects of the TNodeTemplateExtended class are used, which are holding the origin TNodeTemplate object, a label, and an optional keyword. The label is set during the mapping of the TOSCA topology while the keyword will be set, if the keyword search matches.

[2] https://www.gnu.org/licenses/lgpl-3.0.en.html
[3] https://www.eclipse.org/legal/epl-v10.html

5.2.3 PatternComponent

All pattern graphs have objects of the PatternComponent class as vertices. These PatternComponents hold information about the label and two integer values for minimum and maximum occurrences. This is very important if a subgraph isomorphism is detected and the two graphs are compared.

5.2.4 Pattern Taxonomy

The implementation of the pattern taxonomies for *PaaS* and *IaaS* is done with objects of a SimpleDirectedWeightedGraph. This graph offers directed edges with the ability to add a weight, which is important for the probabilities and they are set as follows:

- 0.99 for a detected pattern (cannot be 1.0 because this is the default value for an edge weight)
- 0.75 for high probability
- 0.5 for medium probability
- 0.25 for low probability
- 0.0 for an impossible pattern matching

5.2.5 Pattern Graph

All patterns are implemented in a separate Java class using objects of a DirectedGraph. This is a directed graph which has objects of the type PatternComponent as vertices and edges of the type RelationshipEdge.

5.2.6 Abstract Topology

For the implementation of the mapping of TOSCA topologies, an abstract topology is used. Therefore, a DirectedGraph is used, having TNodeTemplateExtended objects as vertices and RelationshipEdge objects as edges. This AbstractTopology class initially converts the topology to a DirectedGraph. Within the map method, all unlabeled nodes in this graph are labeled. This method implements the concept of mapping TOSCA topologies explained in Chapter 4.

5.2.7 Properties

All text values such as labels, pattern names, or keywords are outsourced to a properties file named patterndetection.properties to enable easy expandability and modification.

5.3 Examples

With two examples of use, the function of the pattern detection will be explained in practice. One example deals with a classic TOSCA topology modeling the architecture of the OpenTOSCA ecosystem. The second example of use is an AWS specific topology with AWS components.

5.3.1 OpenTOSCA Topology

The first example of use utilizes a topology created for the deployment of the OpenTOSCA ecosystem displayed in Figure 5.1 as input.

Keyword Search

In the first step of the algorithm, the Node Templates of the original topology are searched for predefined keywords. If a keyword is found, a new object of the type TNodeTemplateExtended containing the origin Node Template, a label, and the keyword is created. In this use case, the following nodes are detected with keywords and labeled accordingly:

- OpenStack-Liberty-12 is detected with the keyword *OpenStack*. A new TNodeTemplateExtended is created with the label *Virtual_Hardware* and the keyword *OpenStack*.
- Ubuntu-14.04-VM matches the keyword *Ubuntu*. It is labeled with *OperatingSystem* and the keyword *Ubuntu*.
- Tomcat_7 matches with the predefined keyword *Tomcat* and gets labeled with *Server* and the found keyword.
- Java7 is labeled with *Service* and the found keyword *Java*.

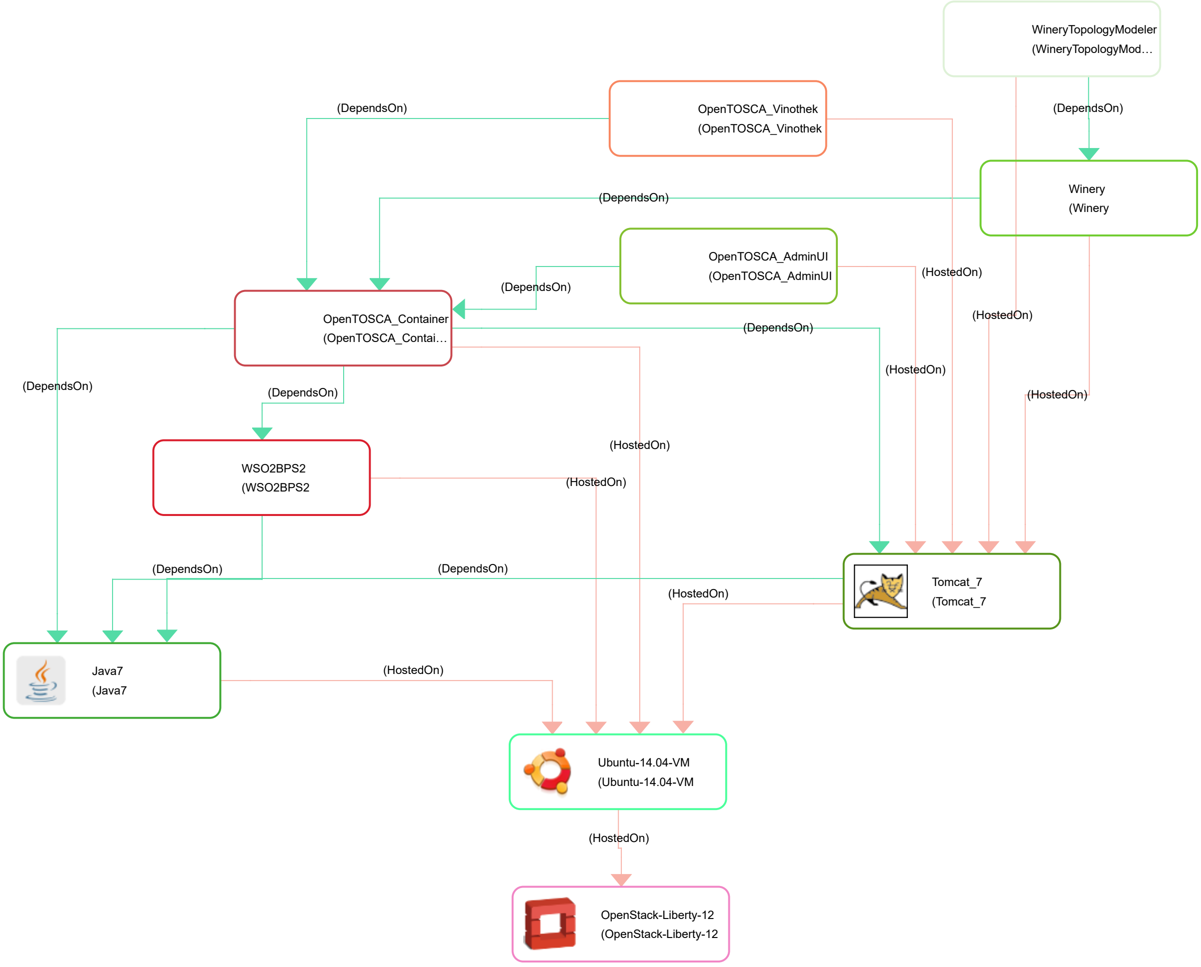

Figure 5.1: TOSCA topology of the OpenTOSCA Ecosystem

Probabilities

Because of the with keywords detected OpenStack component, the algorithm initially assumes an *IaaS*. Now the pattern taxonomy for *IaaS* is used to set probabilities for possible patterns as shown in Figure 5.2. At this moment, the algorithm has two detected patterns: the *IaaS* and the *Elastic Infrastructure* pattern. But because of the fact, that there are more components upon the infrastructure with labels *Server* and *Service*, the algorithm now includes an *PaaS* as underlying structure. Identical to the *IaaS* taxonomy, also the *PaaS* taxonomy in Figure 5.3 is edited.

The probability for *Environment-based Availability* is set high, because only one server component is detected, so the whole environment runs in one instance. On the other side the probability for a *Node-based Availability* is reduced to low. All patterns under

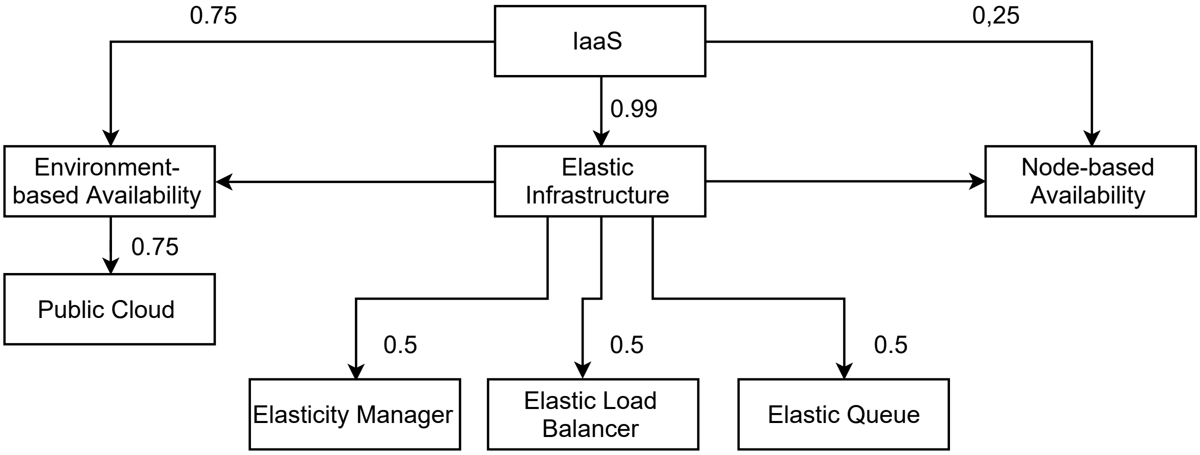

Figure 5.2: Taxonomy for IaaS with probabilities

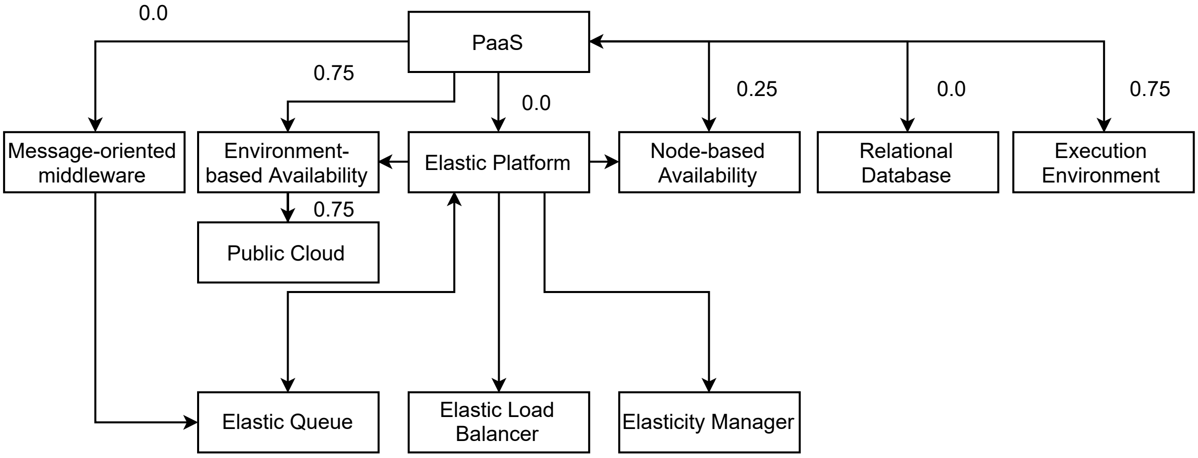

Figure 5.3: Taxonomy for PaaS with probabilities

Elastic Infrastructure are dependent from the installation of OpenStack and cannot be detected in this way.

For the *PaaS* taxonomy, the *Execution Environment* as well as the *Environment-based Availability* are marked with high probability because of the existence of services and one server. Impossible is the *Elastic Platform* because of the *Elastic Infrastructure*.

Mapping of the Topology

In this step of the algorithm, the topology will be completely mapped. Starting from the lowest node, in this case the OpenStack, all nodes now get labeled. Outgoing from the operating system, all nodes with a HostedOn relation get labeled with *Service*. These are

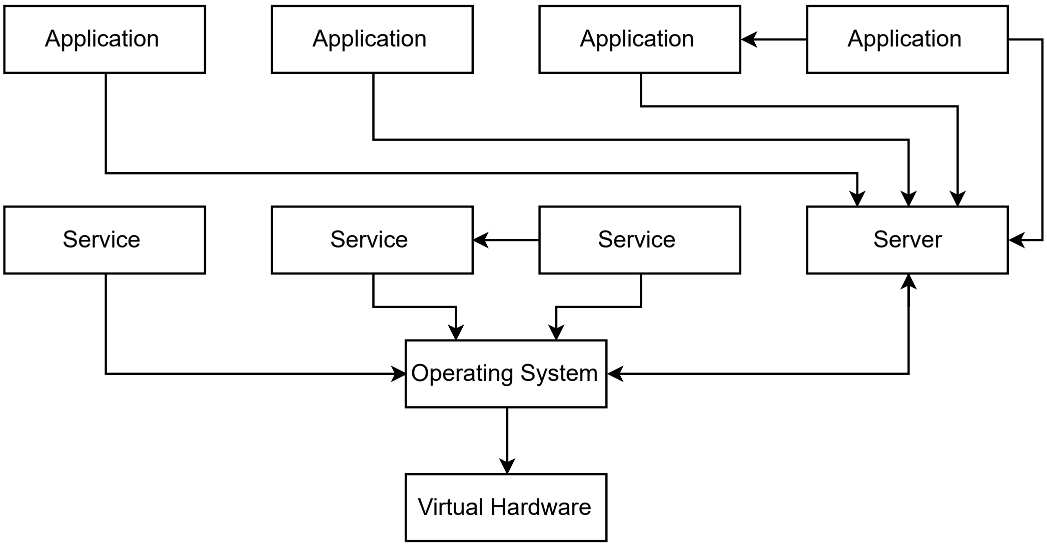

Figure 5.4: Topology with labels

the WSO2BPS2 node and the OpenTOSCA_Container. The Tomcat server was already detected and labeled as a server. Based on the mapping concept, the source nodes of all incoming DeployedOn relations get labeled as *Application*. The labeled nodes will then be added with their edges to a directed graph.

Figure 5.4 represents the graph with the labeled nodes. This is the shape of graph, that is used in the following subgraph isomorphism algorithm.

Subgraph Isomorphism

All patterns are represented as graphs. Now for the topology all possible subgraphs are generated, ignoring the subgraphs with only one vertex. It is necessary to compare the subgraph with the pattern graph afterwards, to ensure that they are isomorphic and have the same structure. Then two lists with the subgraph and the patterns are compared using the VF2 algorithm. As the result, all detected patterns are returned as well as the probabilities for other patterns.

Result

After the algorithm finishes, the following, listed patterns are detected:

- *Environment-based Availability*
- *Execution Environment*

- *Platform as a Service*
- *Elastic Infrastructure*

5.3.2 AWS Topology

The second example handles a topology for the deployment of the OpenTOSCA ecosystem to an AWS service as modeled in Figure 5.5.

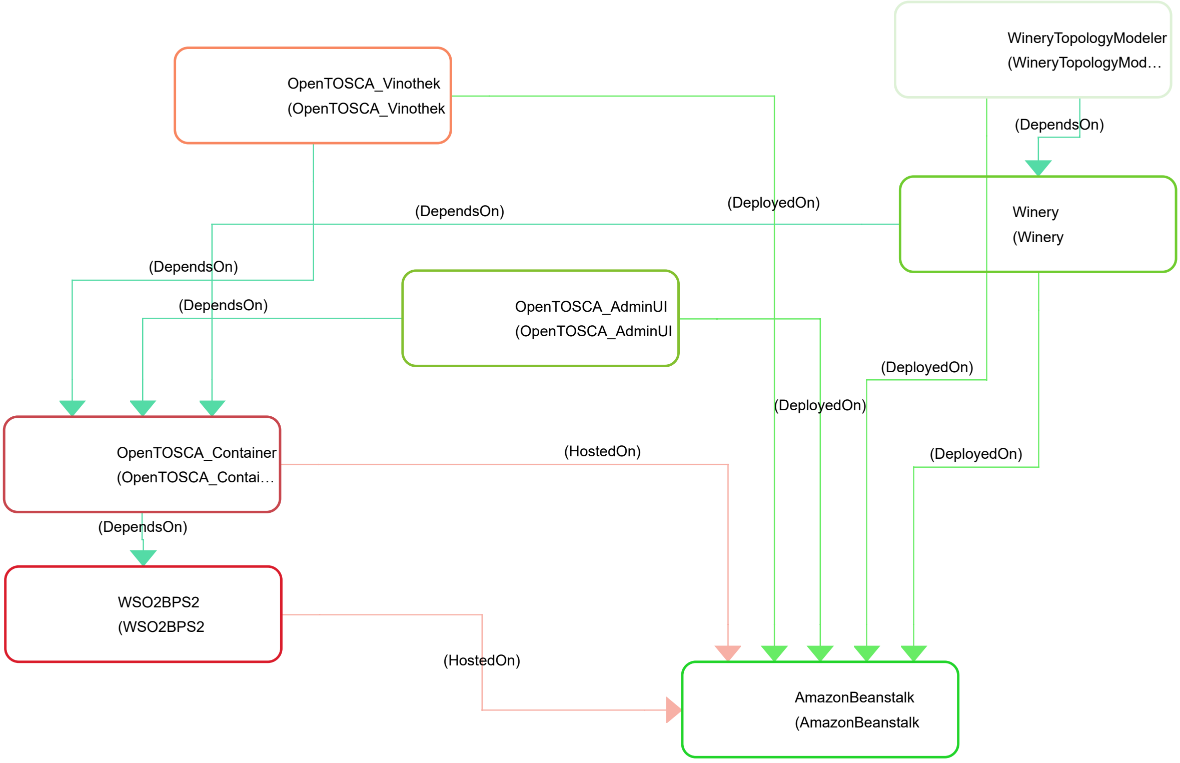

Figure 5.5: Topology for OpenTOSCA with AWS

Keyword Search

In this case, just the AmazonBeanstalk is detected with the keyword *Beanstalk*. It is labeled with *Service*.

Probabilities

In this topology, the algorithm knows because of the Amazon Beanstalk component, that

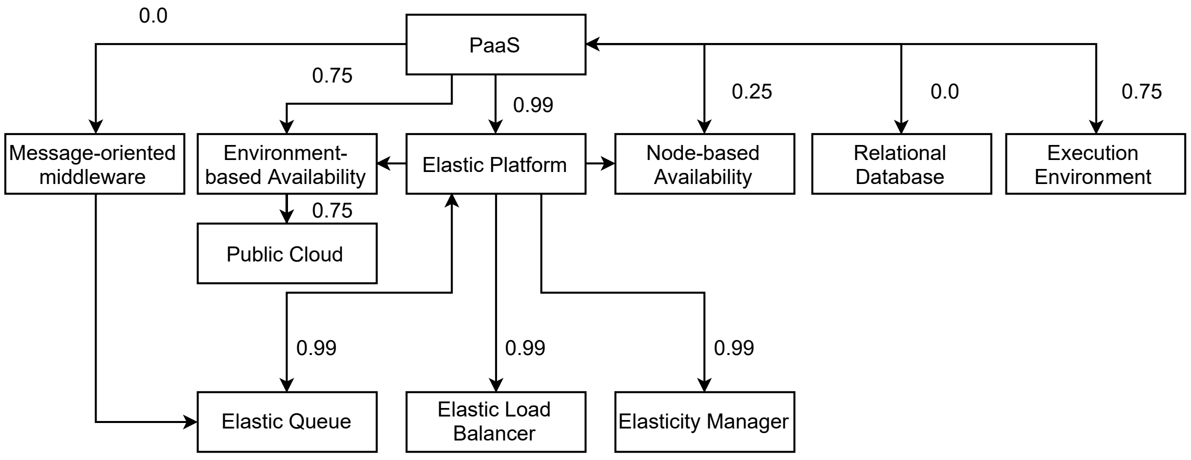

Figure 5.6: Taxonomy for PaaS with probabilities

a *PaaS* service model is present. Because of the Amazon Beanstalk multiple patterns can be marked as detected, since they are already included in Beanstalk. In Figure 5.6 the *Elastic Platform* pattern, *Elastic Queue* pattern, and the *Elasticity Manager* pattern are marked with the value 0.99 which indicates them as detected.

Result

The Amazon Beanstalk offers a complete *PaaS*. On this platform, the custom applications are deployed. There cannot be detected more patterns. The algorithm finishes after the probabilities and returns a list of patterns, which are defined for the use of Amazon Beanstalk:

- *Environment-based Availability*
- *Execution Environment*
- *Platform as a Service*
- *Elastic Platform*
- *Elastic Load Balancer*

5.3.3 Deviations

While comparing the two examples one notices, that in the AWS topology no *Elastic Infrastructure* is detected. Since, the single Amazon Beanstalk component hides all information about the underlying infrastructure, there is no possibility to make any statement about the infrastructure. But within the *Elastic Platform* the use of such an infrastructure is implied, but not mentioned as a detected pattern.

5.4 Problems

During the implementation, some problems occurred. A big issue was the mapping of the topology just based on the knowledge of relations, the lowest node, and keywords. To avoid a chaos of if and else constraints, the constraints shown in Figure 4.9 were defined. Another problem was the combination of *IaaS* and *PaaS*. This issue was approached several times before, because if the topology of a cloud application is fully modeled, it can happen that both, *IaaS* and *PaaS*, are modeled. This problem was solved by checking both in hierarchical order. With the subgraph isomorphism, the following problem was present: If a single subgraph isomorphism algorithm is used, he will only return the result, i.e., if an isomorphism exists, but not the area in the graph. To receive this, subgraphs of the topology were created. Difficult to implement was the creation of all these possible subgraphs for the labeled topology graph. For each node, all possible next nodes plus the combinations with all further detected subgraphs have to be saved as subgraphs. One more challenge was the lack of visual graphs for debugging purposes.

5.5 Limitations

The current implementation works successful, if only one base node is used. This means, that only one Node Template may exist with no outgoing relations. If there are more, the algorithm will not work as expected. Also the topology has to be connected and may only use the defined Relationship Templates. The TOSCA topology also may not have multiple edges between two Node Template. Furthermore, topologies with virtualization components such as Docker[4] are not yet supported. Only the *Cloud Computing Patterns* defined in Chapter 4 are available for detection.

[4] https://www.docker.com/

5.6 Expandability

The functionality can be expanded in different ways. On the one hand, the patterns can be expanded by writing new classes. On the other hand, the available keywords can be easily expanded by adding new ones to the according enum. Also, all text values can be modified in the properties file. By doing this, new Relationship Types can be added.

6 Conclusion & Outlook

The goal of this bachelor's thesis is a concept for an automatic detection of *Cloud Computing Patterns* in TOSCA topologies as well as a first implementation. First of all, the underlying technologies and concepts of TOSCA and OpenTOSCA are explained as well as the sense and the function of design patterns in general. Based on this, a concept for the Design Pattern Detection Framework is investigated. This concept mainly concentrates on the modeling of *Cloud Computing Patterns* as fragments of topologies and the preparation of TOSCA topologies for the pattern detection by mapping the single nodes to abstract components. This mapping starts with a keyword search, to find known components. Afterwards, a mapping algorithm, based on the different relations of components, maps the rest of the topology, to receive a topology with abstracted components. The modeling of *Cloud Computing Patterns* is done with the same abstract components. Both, the patterns and the topologies, are handled as graphs during the detection algorithm. With the use of pattern taxonomies, probabilities for possible patterns based on their interrelations are set. After the creation of all possible subgraphs of the topology graph, these subgraphs are compared to the pattern graphs using the VF2 algorithm. This algorithm detects whether a subgraph is isomorphic to a subgraph in a bigger graph. Pattern subgraphs can be found in topology subgraphs when they are isomorphic. Then this pattern is marked as detected.

The concrete implementation with a direct integration in Winery is explained in Chapter 5 with all important classes. Also, the functionality is explained using two exemplary topologies.

The created framework forms a basis for future work and further implementations. The framework can be extended to support more *Cloud Computing Patterns* as well as more keywords. A further extension could be the detection of *Composite Cloud Patterns*. Also, an improvement of the subgraph isomorphism is possible. In May 2017, the VF3 algorithm, an upgrade of the VF2 algorithm, is introduced by Carletti et al. [CFSV17]. This evolution of the VF2 Plus aims to enhance the performance on graphs that are at the same time large and dense. However, the effectiveness of VF3 is only validated experimentally. To conclude, a concept for pre-processing the topology graph and a separation into logical subgraphs could be useful, e.g., creating a subgraph for every virtual machine. This may reduce the computational time.

Bibliography

[Anz12] Y. Anzai. *Pattern recognition and machine learning*. Elsevier, 2012 (cit. on p. 21).

[BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. "OpenTOSCA - A Runtime for TOSCA-based Cloud Applications." English. In: *Proceedings of 11th International Conference on Service-Oriented Computing (ICSOC'13)*. Vol. 8274. LNCS. Springer Berlin Heidelberg, Dec. 2013, pp. 692–695 (cit. on p. 14).

[BBKL14a] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. "TOSCA: Portable Automated Deployment and Management of Cloud Applications." English. In: Advanced Web Services. New York: Springer, Jan. 2014, pp. 527–549. ISBN: 978-1-4614-7534-7 (cit. on p. 11).

[BBKL14b] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. "Vinothek - A Self-Service Portal for TOSCA." Englisch. In: *Proceedings of the 6th Central-European Workshop on Services and their Composition (ZEUS 2014)*. Ed. by N. Herzberg, M. Kunze. Vol. 1140. CEUR Workshop Proceedings. CEUR-WS.org, März 2014, pp. 69–72 (cit. on pp. 14, 16).

[BBKL16] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. *TOSCA and OpenTOSCA*. Presentation. IAAS University of Stuttgart, 2016 (cit. on pp. 13, 14).

[CFSV04] L. P. Cordella, P. Foggia, C. Sansone, M. Vento. "A (sub)graph isomorphism algorithm for matching large graphs." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.10 (Oct. 2004), pp. 1367–1372. ISSN: 0162-8828 (cit. on pp. 25, 44).

[CFSV17] V. Carletti, P. Foggia, A. Saggese, M. Vento. "Introducing VF3: A New Algorithm for Subgraph Isomorphism." In: *Graph-Based Representations in Pattern Recognition: 11th IAPR-TC-15 International Workshop, GbRPR 2017, Anacapri, Italy, May 16–18, 2017, Proceedings*. Ed. by P. Foggia, C.-L. Liu, M. Vento. Cham: Springer International Publishing, 2017, pp. 128–139. ISBN: 978-3-319-58961-9 (cit. on p. 57).

[CFV15] V. Carletti, P. Foggia, M. Vento. “VF2 Plus: An Improved version of VF2 for Biological Graphs.” In: *International Workshop on Graph-Based Representations in Pattern Recognition*. Springer. 2015, pp. 168–177 (cit. on p. 25).

[FBBL15] C. Fehling, J. Barzen, U. Breitenbücher, F. Leymann. “A Process for Pattern Identification, Authoring, and Application.” German. In: *Proceedings of the 19th European Conference on Pattern Languages of Programs (EuroPLoP)*. ACM, Jan. 2015, pp. 1–9 (cit. on p. 19).

[FLR+11] C. Fehling, F. Leymann, R. Retter, D. Schumm, W. Schupeck. “An Architectural Pattern Language of Cloud-based Applications.” In: *Proceedings of the 18th Conference on Pattern Languages of Programs (PLoP)*. ACM, 2011 (cit. on p. 20).

[FLR+14] C. Fehling, F. Leyman, R. Retter, W. Schupeck, P. Arbitter. *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Wien; Heidelberg [u.a.]: Springer, 2014, XXVI, 367 Seiten. ISBN: 978-3-7091-1567-1 (cit. on pp. 11, 18, 30, 31, 40).

[Fow02] M. Fowler. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0321127420 (cit. on p. 17).

[Gam15] E. Gamma. *Design Patterns: Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software*. 1. Aufl. [Frechen]: mitp, 2015, 472 Seiten. ISBN: 978-3-8266-9700-5 (cit. on pp. 16, 17).

[HS08] H. He, A. K. Singh. “Graphs-at-a-time: query language and access methods for graph databases.” In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM. 2008, pp. 405–418 (cit. on p. 26).

[HSB+14] A. Homer, J. Sharp, L. Brader, M. Narumoto, T. Swanson. “Cloud Design Patterns.” In: (2014) (cit. on p. 17).

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. “Winery - A Modeling Tool for TOSCA-based Cloud Applications.” Englisch. In: *Proceedings of 11th International Conference on Service-Oriented Computing (ICSOC’13)*. Vol. 8274. LNCS. Springer Berlin Heidelberg, Dezember 2013, pp. 700–704 (cit. on p. 15).

[LHKL13] J. Lee, W.-S. Han, R. Kasperovics, J.-H. Lee. “An in-depth comparison of subgraph isomorphism algorithms in graph databases.” In: *Proceedings of the 39th international conference on Very Large Data Bases*. PVLDB’13. Trento, Italy: VLDB Endowment, 2013, pp. 133–144 (cit. on p. 25).

[MG11] P. Mell, T. Grance. *The NIST Definition of Cloud Computing*. 2011. URL: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf (cit. on p. 18).

[OAS13] OASIS. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0*. 2013 (cit. on p. 13).

[QLDG09] L. Qian, Z. Luo, Y. Du, L. Guo. “Cloud computing: An overview.” In: *Cloud computing* (2009), pp. 626–631 (cit. on p. 11).

[SZLY08] H. Shang, Y. Zhang, X. Lin, J. X. Yu. “Taming verification hardness: an efficient algorithm for testing subgraph isomorphism.” In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 364–375 (cit. on p. 26).

[Ull76] J. R. Ullmann. “An Algorithm for Subgraph Isomorphism.” In: 1976 (cit. on p. 23).

[Web03] A. R. Webb. *Statistical pattern recognition*. John Wiley & Sons, 2003 (cit. on p. 22).

[ZH10] P. Zhao, J. Han. “On graph query optimization in large networks.” In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 340–351 (cit. on p. 27).

[ZLY09] S. Zhang, S. Li, J. Yang. “GADDI: distance index based subgraph matching in biological networks.” In: *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. ACM. 2009, pp. 192–203 (cit. on p. 26).

All links were last followed on July 3, 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