Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor's thesis

Design and Implementation of a Framework to Automate the Inclusion of Patterns in Existing Architectures

Aaron Röhl

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	Jasmin Guth, M.Sc.
Commenced:	June 1, 2017
Completed:	December 1, 2017
CR-Classification:	I.7.2

Abstract

This bachelor's thesis aims to provide a framework to support the evaluation and automated inclusion of proven solutions, so-called patterns, in software architectures. By simplifying the identification of bad design choices and automating some parts of the architecture's redesign, the whole design process will provide standardized solutions, be less error-prone and less time-consuming. Nevertheless, it is no replacement for a software architect but a useful and supportive tool. While the framework supports different technologies and standards, the examples are based on OASIS' TOSCA and cloud computing patterns. Using the NFR-Framework, a method for analyzing existing patterns with regard to their influence on a system's non-functional requirements will be explained. Besides the resulting information, a pattern detection framework will then be used in order to determine the impact of different patterns on an existing architecture. Further, a basic approach for automated pattern inclusion will be introduced. A corresponding Java-based implementation that shows the framework's main functionality is part of this thesis. Using this application, the framework was tested by evaluating the influence of different cloud computing patterns on an example TOSCA topology. However, this bachelor's thesis is theoretical and the application is just a prototype; empirical data supporting the advantages of this framework is still missing.

Kurzfassung

Das Ziel dieser Bachelorarbeit war die Entwicklung eines Frameworks, welches die Evaluation und das Einbinden von bewährten Lösungen, so genannten Patterns, in Softwarearchitekturen ermöglicht. Indem die Identifikation von schlechten Designentscheidungen vereinfacht wird, sowie durch die Automatisierung eines gewissen Teils des Umgestaltungsprozesses der Architektur, wird der gesamte Designprozess zu einem gewissen Maße standardisiert, weniger fehleranfällig und weniger zeitaufwändig. Dennoch ist das Framework vielmehr ein unterstützendes Werkzeug als ein Ersatz für einen Softwarearchitekten. Während verschiedene Standards und Technologien unterstützt werden, basieren die Beispiele auf Cloud-Computing Patterns und TOSCA-Topologien. Es wird eine an das NFR-Framework angelehnte Methode vorgestellt, die den Einfluss von Patterns auf die nichtfunktionalen Anforderungen eines Systems analysiert. Neben den daraus gewonnenen Informationen wird ein Pattern-Erkennungsframework eigesetzt, um den Einfluss verschiedener Patterns auf eine gegebene Architektur zu ermitteln. Des Weiteren wird eine grundlegende Herangehensweise für die automatisierte Pattern-Inklusion vorgestellt. Eine zugehörige Java Implementierung, welche die grundlegenden Aspekte des Frameworks unterstützt, ist ebenfalls Teil dieser Arbeit. Die Anwendung wurde genutzt, um den Einfluss von ausgewählten, vorevaluierten Cloud-Computing Patterns auf eine ausgewählte TOSCA-Topologie zu ermitteln. Diese Bachelorarbeit ist rein theoretisch und die Anwendung ein Prototyp; aktuell fehlt es noch an empirischen Daten, welche die Vorteile des Frameworks belegen.

Contents

List of Figures

List of Algorithms

List of Abbreviations

AADL Architecture Analysis & Design Language. 29

ADL Architecture Description Language. 28

API Application Programming Interface. 22

CC Cloud Computing. 16

CSAR Cloud Service ARchive. 19

HTTP HyperText Transfer Protocol. 25

IaaS Infrastructure-as-a-Service. 23

JRE Java Runtime Environment. 41

NFRs non-functional requirements. 13

OASIS Organization for the Advancement of Structured Information Standards. 19

OSLC Open Services for Lifecycle Collaboration. 25

PaaS Platform-as-a-Service. 23

RDF Resource Description Framework. 25

SIG Softgoal Interdependency Graph. 19

TOSCA Topology and Orchestration Specification for Cloud Applications. 13

UML Unified Modeling Language. 28

1 Introduction

Today, cloud computing is a ubiquitous used model for on-demand network access to a distributed pool of computing resources. Many big companies like Google, Facebook and Twitter rely on this state-of-the-art technology in order to handle the high amount of daily traffic. While the previous static models can't fulfill modern requirements like high availability and optimal resource utilization, cloud computing achieves these goals by provisioning and releasing resources on-demand with minimal effort for the service provider [MG+11]. Due to the increasing complexity of cloud computing software systems, the branch of software architecture became more and more important over the last years. A software architecture describes individual components as well as their relationships and interaction in an overall system. Therefore, the role of a software architect includes the design of a suitable model that fulfills specific functional and non-functional requirements (NFRs) as well as considering pros and cons for different approaches [BM+02]. In order to specify the interaction between different components while maintaining interoperability, software architects make use of architecture description languages like UML or OASIS' Topology and Orchestration Specification for Cloud Applications (TOSCA) to describe dependencies and requirements among all parts of the overall system. In TOSCA, this description is called a service topology [Sta13]. A pattern is an abstract concept that describes a proven solution to a recurring problem, independent from the used technologies. Using patterns often increases the quality of a system and accelerates the design process [FLR+14].

This bachelor's thesis describes a framework to optimize a given software topology by evaluating, automatically detecting and adding proven patterns to better fulfill predefined NFRs. The framework can be adapted to different technologies but will be explained based on the example of TOSCA topologies and cloud computing patterns.

Motivation

In contrast to functional requirements, NFRs are way more difficult to satisfy. Because of their global impact on the system, their informal, textual goal definitions, tracing difficulties and missing strategies to achieve the specified goals, NFRs are often neglected [CSB+05]. Even if their importance is understood, missing tools make it hard for software architects to fulfill the given requirements [GY01]. The lack of appropriate tools has a big impact to the economic viability of a software system. If a specific requirement of a system is not fulfilled, an architect has to analyze the whole structure of the system, reevaluate all previous design choices, try to come up with a new approach and consider all possible trade-offs. The whole procedure is very time consuming and not structured in an organized way, increasing the costs of the software unpredictably. Therefore, the framework described in this bachelor's thesis can reduce the costs of a software regarding its NFR optimizations by structuring and automating the previously described process.

Structure

The work is structured as follows:

Chapter 2 – Fundamentals and Definitions
The fundamental keywords and methods used in this work will be explained in this chapter. All further appearances of the terms Non-functional Requirements, Patterns and OASIS' TOSCA refer to the definitions in this chapter. Further, the NFR-Framework and Pattern Detection is explained.

Chapter 3 – Related Work
Some selected work related to the main topics of this bachelor's thesis will be presented in this chapter. Every item is also evaluated for the use with this framework.

Chapter 4 – Approach
The three main steps of the pattern inclusion framework will be described conceptually. The individual patterns' influence evaluation will be covered as well as the basic idea behind pattern inclusion.

Chapter 5 – Implementation
This chapter introduces the Pattern Inclusion Application (PIA), which is a graphical tool that was designed to state the use of the pattern inclusion framework. Everything necessary related to the application's structure and functionality will be covered.

Chapter 6 – Validation
Every step of the framework and its algorithms will be clearified based on an example TOSCA topology. This chapter should increase the comprehensibility of the framework.

Chapter 7 – Conclusion and Outlook
This last chapter provides a short summary of the main indeas behind this bachelor's thesis as well as the corresponding solutions. Further, the advantages of the pattern inclusion framework and possible future improvements will be discussed.

2 Fundamentals and Definitions

This chapter deals with fundamental definitions which apply to all following occurences of the defined terms. Besides differentiations of keywords, some of the used methods and technologies will be explained.

2.1 Non-functional Requirements (NFRs)

In order to define non-functional requirements (NFRs), it is necessary to define functional requirements first. Functional requirements always describe a concrete task a system or component must be able to perform, including all inputs, outputs and behavior of the system or component to complete the task. All requirements that don't fit into functional requirements are aggregated under in the term NFRs, which includes very abstract goal definitions like *Availability* or *Efficiency*[Eid05].

ISO 25010 arranges NFRs hierarchically. There are eight main NFRs: *Functional Suitability, Reliability, Performance Efficiency, Operability, Security, Comparability, Maintainability* and *Transferability*. All other requirements like *Accessibility*, *Reusabilty* and *Availability* are just sub-requirements of the eight previously named ones. A change that was made in a sub-requirement is always propagated bottom-up, which means that a sub-requirement's modification always influences its parent while a direct change of a parent not always influences all its children [LOZ10]. In this bachelor's thesis, the term NFR always refers to the requirements defined in ISO 25010, shown in figure 2.1.

2.2 Cloud Computing Patterns

Software patterns describe proven solutions to recurring problems in different stages of the development process. The use of patterns can improve the undstandability of a system's architecture and ease maintenance. They provide a common vocabulary and holistic solutions for all kind of problems [HC07]. While patterns like Singleton or Factory offer solutions for common implementation problems across all industries, they cannot be applied to architectural design problems or the maintenance process. Therefore, every stage of the software development process has its own patterns as well as every branch of industry has specific patterns for their own needs. This diversity may result in difficulties for the software designers and developers to keep an overview of all existing patterns [Wol94]. Henninger and Corrêa [HC07] mention 425 patterns only for the user interface;

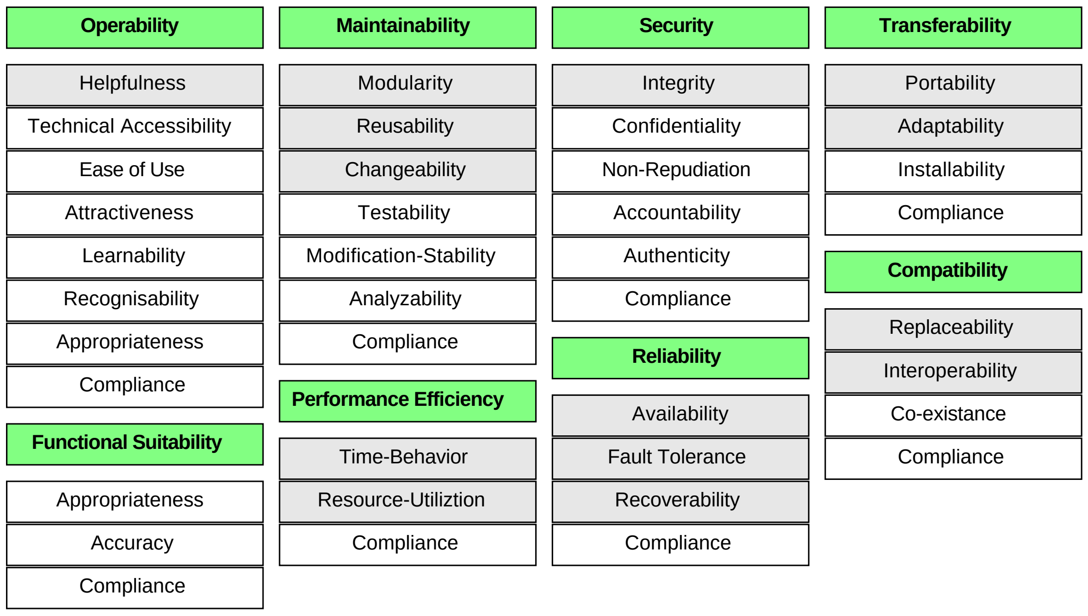

Figure 2.1: NFR Hierarchy as defined in ISO 25010

the sum of all patterns lies above 1500. Since the article was published in 2007, probably even more patterns exist today. Therefore, a general technique will be explained that works with every previous and future pattern.

The framework described in this bachelor's thesis is applicable to every kind of pattern independent from its branch of industry or corresponding stage in the development process. Architecture design patterns from a field called Cloud Computing will be used as an example.

Cloud Computing (CC) is an ascending technology that enables ubiquitous, convenient and on-demand network access to a network of computing resources like databases, infrastructure or services [MG+11]. The goal is processing and storing large amounts of data while maintaining a fast response time. This is achieved by provisioning and releasing distributed resources over the internet with minimal human intervention. Many CC system operators prefer renting third party resources like Amazon Web Services[1] or the Google Cloud Platform[2] to run their applications [BAA12].

The CC patterns used as example in this thesis can be applied to every kind of cloud computing architecture. It doesn't matter if the deployment model is a private, public or hybrid cloud. The most important patterns are described below.

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

2.2.1 Watchdog Pattern

Because CC systems are distributed, they depend on the availability of every component. A component failure may result in an overall system failure. Thus, it is important that every application component instance is redundant, very fault tolerant and, if a failure occurs, is replaced by a new instance as fast as possible. This can be seen as one of the most important challenges of CC [BAA12].

In order to keep track of failures, gather necessary information, create statistics, replace inoperable instances and provide a suitable interface for the user, the watchdog pattern can be applied to a software architecture. By installing a new service that connects to all application instances and monitors their condition, the watchdog always knows about the current status of all components, and therefore of the overall system. If an application doesn't answer, it is automatically replaced by its redundant correspondent. Failures are logged, saved and possibly depicted in an informing graphical user interface for further investigation. This results in a high availability of every instance and the overall system. A drawback is the necessity of the stateless component pattern. That means that every instance must be independent from the status of other application instances. Using this pattern, every failed application can simply be replaced by another instance of itself. The overall system's integrity will not be affected. If the system was not designed using this pattern, it can be an unbearable overhead to rewrite the whole architecture and its components [FLR+14].

2.2.2 Environment-based Availability

Due to the necessity of a distributed and dynamic infrastructure for CC systems, many users like to rent an existing solution from a CC provider like Amazon Web Services. The provider offers an environment on which a customer deploys his applications. In order to maintain a high availability of the overall system, the provider has to guarantee the availability of the environment hosting individual nodes like individual virtual servers, middleware components or hosted applications. This is done by specifying conditions for the offering and a time-frame in which the offering must be available. As an example, Amazon Web Services' EC2 guarantees scalable calculation resources with an up-time of 99.95% per accounting period [3]. The nodes themselves must be monitored and, in case of failure, replaced by another working instance. Since the customer is only provided with the necessary information about the condition of its nodes, he has to cope with failures on his own. Using heartbeats, the availability of every instance can be assured. If a heartbeat wasn't answered, the corresponding instance must be replaced. The Watchdog Pattern is a solution for automated failure management that can be used for this purpose. It is not trivial to determine the overall availability of an environment based application. It depends on the availability of the environment and the quality of failure management [FLR+14].

[3]https://aws.amazon.com/ec2/sla/ (10-20-2017)

The Environment-based Availability is no pattern per se. Rather, the application owner may rent an infrastructure or platform that assures either Environment- or Node-based Availability. Therefore, the representation in an architecture is kind of tricky. Nevertheless, it is possible. The idea behind using Environment-based Availability as a pattern will be described in 4.3.

2.2.3 Node-based Availability

In contrast to the Environment-based solutions, Node-based offerings guarantee the availability of every single node. A customer rents the infrastructure or platform and deploys his applications. The provider has to guarantee the availability of every single node by specifying conditions and a minimum up-time as with Environment-based Availability. Therefore, the availability of the overall system can be calculated by multiplying the availabilities of all nodes as in formula 2.1. The formula shows that if a single node has a low availability, the overall availability decreases substantially.

$$Availability(System) = \prod_{Nodes} Availability(Node) \tag{2.1}$$

Using formula 2.1, it can be calculated that if the guaranteed availability of two nodes is 99.95% and 80.00% respectively, the overall availability is just 79.96%. In order to achieve a high availability, the provider often incorporates redundant hardware components. The environment is responsible for monitoring all node failures and, if a node fails, provisioning new hardware. The overall availability can be further increased by deploying multiple instances of the same node and by replacing them if necessary. The Watchdog Pattern can be used for this purpose. By sending test data and heartbeats, the correct functionality of all nodes can be guaranteed.

Just like the Environment-based Availability, Node-based Availability is no real pattern. There is a possibility to represent Node-based Availability in a topology, though. The representation focuses on maximization of each component's independence. Further information about the idea behind Node-based Availability as a pattern will be provided in section 4.3.

2.3 NFR-Framework

In contrast to the traditional product-oriented approaches which often use metrics, the NFR-Framework offers a goal-driven, process-oriented approach to deal with the organization, representation and analysis of NFRs. It is goal-driven because a single NFR is represented as a goal that should be achieved and process-oriented because design choices can be justified during the development process. The framework was designed to determine and justify trade-offs between different development options [SSC03]. NFR goals are evaluated using synergistic and conflicting interdependencies. Every development option will be examined regarding its influence on the specified NFRs. Therefore, trade-offs between different design choices can be determined [PC02]. The NFR-Framework offers a systematic way to evaluate the influence of a specific design choice on predefined NFRs. In this bachelor's thesis, it will be used to analyze the impact of patterns on a software architecture. The value of a pattern consists of the positive and negative effects on the architecture's NFRs.

To evaluate a specific design option using the NFR-Framework, a Softgoal Interdependency Graph (SIG) must be created. A SIG can be used to analyze trade-offs between a design choice and different NFRs. Because not every aspect of the NFR-Framework will be used later on, the following description focuses on the important parts for this thesis' purpose. The whole range of functionality can be found at [CNYM12]. Generally, softgoals are goals which have no clear-cut definition that could be satisfied. NFRs are represented by NFR softgoals. A SIG also contains operationalizing softgoals which are used to describe possible design solutions that may influence the NFRs. Both previously named softgoals are composed of a type and a topic. For NFR softgoals, the type describes an aspect like reliability or operability, for operationalizing softgoals aspects like resource management or monitoring. The topic describes the target's subject which is associated with the softgoal [PC02]. The topic is always depicted in square brackets, the type in front of it.

Softgoals may influence other softgoals. Therefore, synergistic and conflicting interdependencies exist. An interdependency will be valued with the effect the source has on the target softgoal. That means that a relationship can be surely positive (++ or *MAKE*), partially positive (+ or *HELP*), partially negative (- or *HURT*) and surely negative (- - or *BREAK*) [SSC03]. It is important to notice that these units are abstract since an exact measurement of the NFRs and their trade-off is not possible by definition.

Different connection types can be used to specify the relation of a goal and its children. Connected with an *AND*-relationship, all sub-goals must be satisfied to satisfy their parent. As an example, the SIG in figure 2.2 contains the parent goal "Monitoring", which consists of the sub-goals "create realtime overview" and "create statistics of failure". The monitoring goal will only be satisfied if both of them are fulfilled. In contrast to that, if not all but at least one sub-goal must be satisfied to fulfill the parent, the *OR*-relationship should be used.

2.4 Topology and Orchestration Specification for Cloud Applications (TOSCA)

TOSCA [Sta13] is a cloud computing related standard for platform independent semi-automatic creation and management of application layer services across different environments. Version 1.0 was first published by the Organization for the Advancement of Structured Information Standards (OASIS)[4] in November 2013. Using service templates, components and their relationships can be described and their creation and modification

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

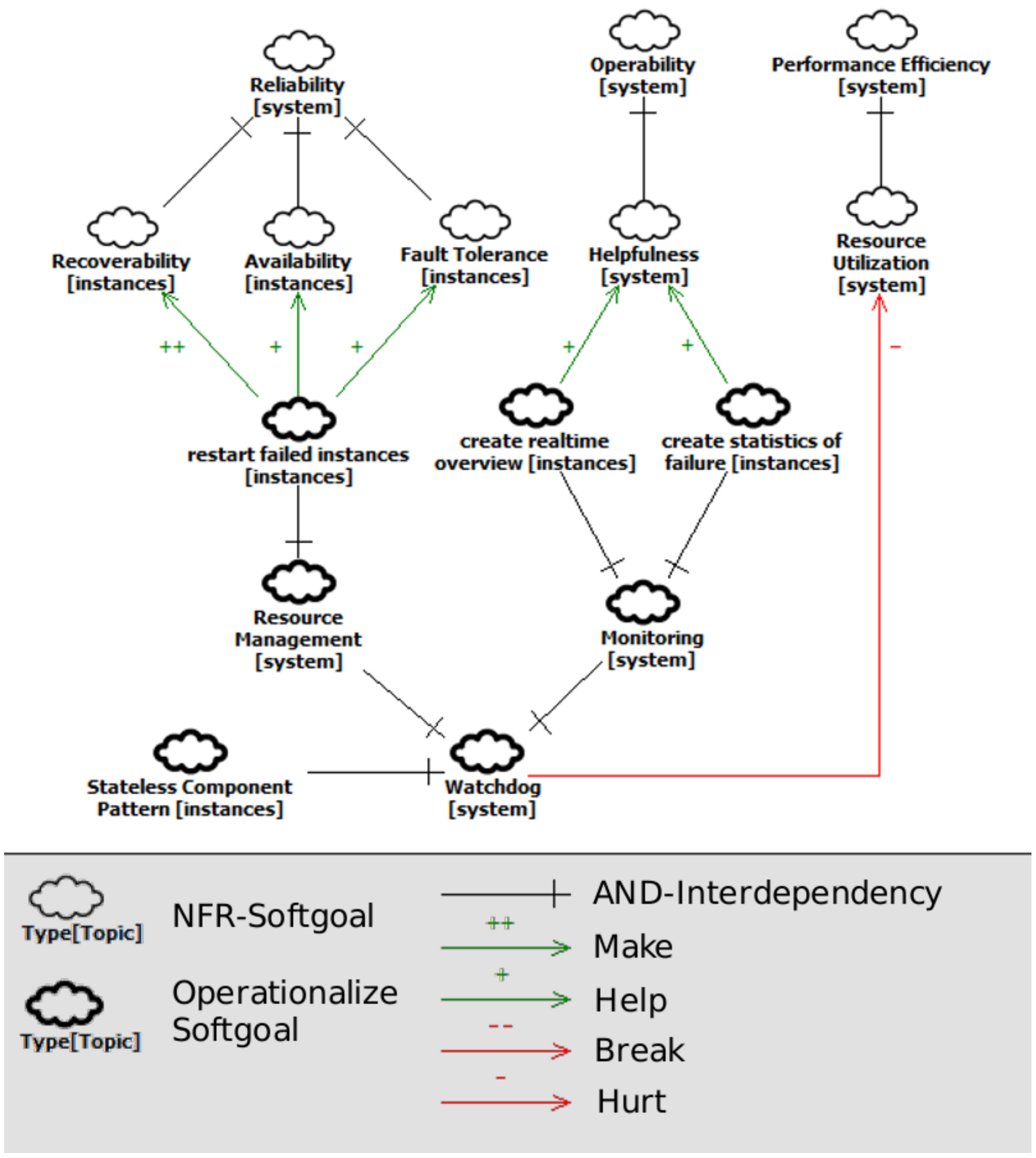
Reliability [system]
Operability [system]
Performance Efficiency [system]
Recoverability [instances]
Availability [instances]
Fault Tolerance [instances]
Helpfulness [system]
Resource Utilization [system]
++
+
+
+
+
-
restart failed instances [instances]
create realtime overview [instances]
create statistics of failure [instances]
Resource Management [system]
Monitoring [system]
Stateless Component Pattern [instances]
Watchdog [system]
Type[Topic]
NFR-Softgoal
Type[Topic]
Operationalize Softgoal
AND-Interdependency
++
Make
+
Help
--
Break
-
Hurt

Figure 2.2: SIG of the Watchdog Pattern

orchestrated. The components and their interdependencies are described in service topologies, which means they are hierarchically ordered. Using this information along with the orchestration info, cloud applications can be deployed in different environments and stay interoperable. They are manageable throughout their complete life-cycle, including scaling, patching and monitoring. In order to deploy an application in different environments, a compressed archive called Cloud Service ARchive (CSAR) will be used. A CSAR archive is a zip file that contains different information about the application, including metadata, TOSCA definitions, the service topology and more. The service orchestration of the TOSCA standard doesn't play an important role in this bachelor's thesis. Therefore, the focus is on service topologies. For further information on service orchestration, see Binz et al. [BBH+13].

A service topology can be represented as a typed, directed graph. Every component corresponds to a node and every interrelationship to an edge. A node describes one single component of the system. This component can be a virtual server, web server, application, execution environment, database et cetera. For every node, individual properties like port and version can be defined. The edges can be labeled with *dependsOn*, *connectTo*, *deployedOn* or *hostedOn*. The meaning of each label is described below [BBH+13]:

hostedOn - If the source component is hosted on the relationship's target, this label will be used (e.g. an operating system is hosted on virtual hardware)

deployedOn - If the source component is deployed on the relationship's target, this label will be used (e.g. a .war application is deployed on a tomcat server)

dependsOn - If a source component depends on the relationship's target, this label will be used (e.g. an application depends on a Java Runtime Environment)

connectTo - If the source component connects to the relationship's target, this label will be used (e.g. an application connects to a database server)

Figure 2.3 shows an example topology including different components as well as all previously named relationships. The topology is hierarchically structured. In order to run the *WebApplication*, all dependencies must be satisfied. Therefore, *OpenStack-Liberty-12* must be started first, followed by *Ubuntu-14.04-VM*, *Java7* and *Tomcat_7*. Only after all these components as well as the *MySQL_Server* are ready, the *WebApplication* can be run.

OpenTOSCA[5] was developed by the University of Stuttgart and provides an open source ecosystem for development, deployment, management and instantiation of TOSCA compliant applications. It consists of three main parts:

1. *Winery*, which is a graphical modeling tool for tosca service topologies
2. *OpenTOSCA container*, which offers a runtime environment for TOSCA applications
3. *Vinothek*, which provides a self-service portal for the management of TOSCA applications

[5]http://www.iaas.uni-stuttgart.de/OpenTOSCA/

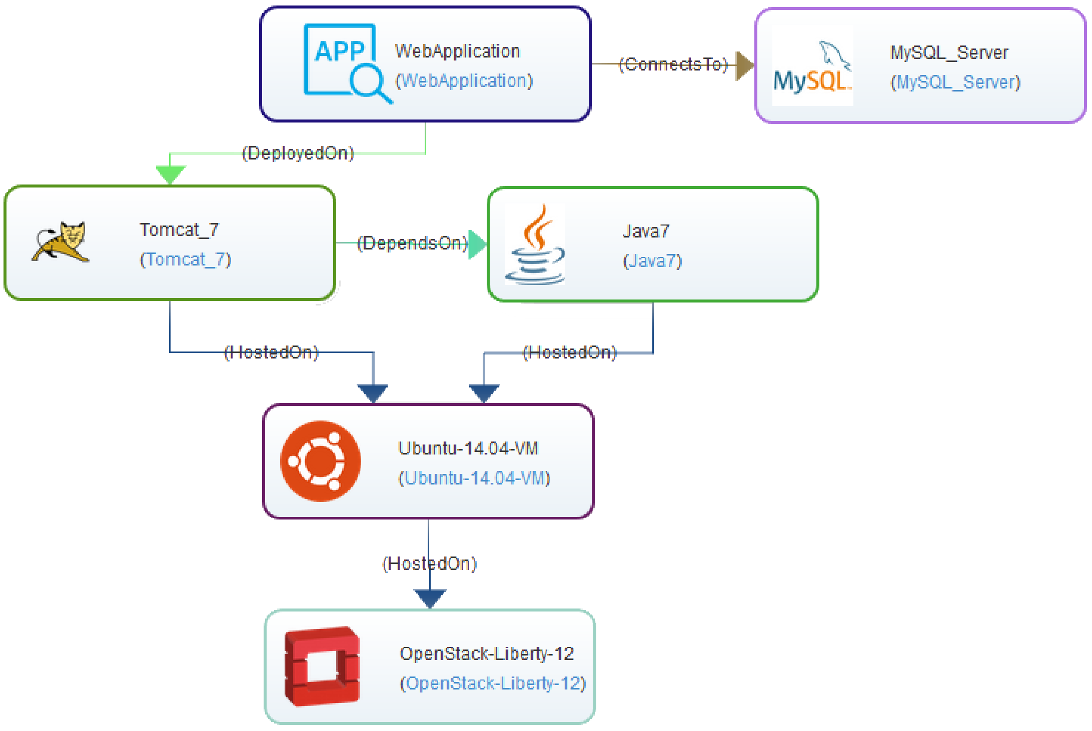

Figure 2.3: Example TOSCA-Topology

Winery is a HTML5 web-based modeling tool that allows the user to define reusable components and relationship types, as well as designing software architectures. It supports in- and export of topologies and their affiliated information on nodes and relationships as CSAR archives. As CSAR archives are standardized by OASIS, all TOSCA conform environments can handle the designed architectures [KBBL13].

The *OpenTOSCA container* provides an Application Programming Interface (API) for the execution of management plans in its own execution environment. A management plan describes the necessary steps to get a management task like instantiation, backup or termination done. The container further includes a controller which orchestrates all components and tracks their progress [BBH+13].

Vinothek allows the user to provision new applications through a HTML5 and JavaScript based web-application. The main goal is simplicity: every user should be able to handle execution plans on different TOSCA runtimes. Therefore, *Vinothek* hides most technical details in order to provide a simplistic interface for management tasks. All actions are propagated via a RESTful API to a corresponding management component which executes a given plan [BBKL14].

OpenTOSCA was mentioned at this point because the pattern detection framework that will be used later on in this bachelor's thesis was first part of *Winery*. The next section provides more information about the pattern detection process.

2.5 Pattern Detection

Existing software architectures may already contain different design patterns. Pattern detection can be useful in order to retrieve and re-validate design choices. The manual way to do so is to analyze the whole software architecture. Besides this time consuming process, all used patterns must be known by the analyst which is very unlikely considering more than 1500 [HC07] existing patterns. To speed this process up, techniques for automated pattern detection were invented. Wohlfarth [Woh17] introduced a detection framework for CC patterns. This framework was included in the *Winery* component of the openTOSCA ecosystem. Therefore, every CSAR archive describing a TOSCA conform application can be loaded and analyzed. Detecting patterns in the current architecture is even possible during the design process.

The detection framework by [Woh17] recognizes the following CC patterns: *Platform-as-a-Service (PaaS), Environment-based Availability, Execution Environment Pattern, Public Cloud, Elastic Platform, Node-based Availability, Elasticity Manager, Elastic LoadBalancer, Elastic Queue, Relational Database, Message-oriented Middleware, Infrastructure-as-a-Service (IaaS)* and *Elastic Infrastructure*. Since the cohesion of different CC patterns can be subdivided in strong, exclusive and undetermined relations, we know that, for example, *PaaS* and *IaaS* cannot exist together whereas *IaaS* and *Elastic Infrastructure* are often used together [FLR+11]. The pattern detection framework considers this cohesions and introduces a five layered probability hierarchy. The probability of every pattern is classified as either Detected, High, Medium, Low or Impossible, depending on their specific cohesion with other patterns.

The framework described in this bachelor's thesis makes use of a pattern detection framework in order to reason about the influence of different patterns on an architecture's NFRs. In case a detected pattern violates a specific requirement, the removal of this pattern can be considered. If removal is no option, it is possible to compensate the loss with the application of another pattern that hasn't been applied yet. In addition to that, the probability of each detected pattern can be used as an indicator of relevance.

3 Related Work

The framework described in this bachelor's thesis can be adapted to different state-of-the-art technologies. In order to understand the full potential, related work will be presented in this chapter. The following examples can be modified to work with the technologies described here. While this is just a small excerpt of all related work, the most important technologies will be covered.

3.1 TOSCA

TOSCA offers an open, standardized solution for different CC problems. First, a software architecture has to be hierarchically described using XML or YAML files and be saved as a CSAR archive. Later, management plans offer a solution for automated updates, deployment and termination of the previously described application. Together, the TOSCA standard contains a textual description for components and interdependencies in a cloud computing architecture as well as a standard for the orchestration of applications themselves [SBB+16]. Related techniques and solutions will be discussed in this section.

3.1.1 Communication Standard

Oasis' TOSCA was developed to solve cloud application interoperability and flexibility issues. Without a common and open standard, every application has its own API, making interaction between services very difficult. Using standardized interfaces, the interoperability between services can be increased significantly.

Besides TOSCA, which deals with these issues in a CC environment, the Open Services for Lifecycle Collaboration (OSLC)[1] standard which was released in 2008 describes a set of specifications which focuses on integration techniques for the whole software life-cycle. By specifying rules for communication per HyperText Transfer Protocol (HTTP), RESTful APIs and the Resource Description Framework (RDF), OSLC deals with the interoperability of services in different domains. It was not designed for architectural design of applications but for a common standard in service communication and can therefore not be used to detect or include patterns in a given software architecture [EN13]. The services itself should implement a standardized interface though. Standards like OSLC help setting up TOSCA conform applications by simplifying the communication process.

[1]https://open-services.net/ (10-24-2017)

3.1.2 Cloud Service Orchestration

Besides a standard for the description of architectures, TOSCA offers a standard for the orchestration of cloud services. The term *orchestration* includes different management operations that can be performed manually or automated by cloud providers and application owners, respectively. Therefore, every layer of a cloud application (infrastructure, platform and service) has its own management operations as, for example, requesting new hardware resources or start and stop a specific node. All management operations can be categorized in one of the following categories [RBDP15]:

- **Resource selection -** Possible candidate software and hardware resources are selected by the application owner to satisfy functional and non functional requirements.
- **Resource deployment -** All components of an application are instantiated on a corresponding node of a cloud service. This operation also includes the configuration and establishment of connections between individual nodes such as applications and database servers.
- **Resource monitoring -** In order to keep track of the satisfaction of NFRs like *Availability*, produced information (up-time, load spikes, ...) must be monitored.
- **Resource controlling -** While monitoring, automated reacting to occurring events can be mandatory to maintain the availability of a system. Not only automated but also manual corrective actions like requesting new resources and rescaling of applications fall into this category.

It is important to mention that TOSCA is a standard, which means that no special application is provided for the design of service templates or the orchestration. Rather, different manufacturers may design a software that is compatible with TOSCA. A TOSCA conform management plan can then be executed by all of them. Because most orchestration services have a unique way of handling the application management, some orchestration technologies of different providers are explained below.

Oracle Orchestration Cloud Service

The Oracle Cloud[2] ecosystem includes rentable IaaS, PaaS and SaaS cloud solutions as well as an orchestration service. Customers may monitor workflows and manage their applications by executing scripts or invoke other web services in a scheduled manner. Not only Oracle Management Cloud agents are compatible but every node implementing a REST interface. Therefore, node sets of different applications all over the world can be orchestrated using predefined management plans. Besides *Resource selection* and *Resource deployment*, even the *monitoring* category is covered. Oracle offers a web-based pre-built dashboard showing different statistics about the status of the application over time. In addition to that, the orchestration service keeps track of the failure rate and anomalies

[2]https://cloud.oracle.com/ (10-27-2017)

in the workflow. The user will be alerted in case of an irregularity, enabling fast trouble shooting. The final *Resource controlling* step must be done manually via web interface [17].

In contrast to TOSCA, the applications' architecture is no direct part of the orchestration process in the Oracle Orchestration Cloud Service. Rather, management plans can be designed which will invoke remote nodes, causing some reaction. The separation of architectural design and the orchestration process can be sometimes useful, but it can also complicate the whole process. If, for example, a specific port was changed, TOSCA has no problem handling the situation. By changing the port in the service templates' properties, the orchestration is not affected. In contrast, every occurence of that node's port in every script of Oracles' orchestration service must be replaced by the new one if the port was changed.

Chef

Almost every big cloud service provider (IBM, Intel, Oracle, ...) has its own solution for cloud orchestration. Besides that, small stand-alone solutions also exist, though. Examples would be Puppet[3], Chef[4] and Cloudify[5]. While Chef and Puppet depend on own domain specific languages to describe the orchestration process, Cloudify is based on TOSCA just like the OpenTOSCA environment. Because OpenTOSCA was presented before, it won't be covered in this section. Rather, the functionality of Chef will be explained as an example for a stand-alone orchestration solution.

Chef is an open-source automation platform for the configuration and deployment of distributed systems based on a client-server model. Every physical and virtual machine, a so called node, needs to run a chef client. The client performs individual tasks such as configuration- or application management on its node. A task is defined by a script in a Ruby based domain specific language, also called a recipe. In order to get the latest recipes, every client connects to the Chef server. The user can write own recipes and upload them to the server. Therefore, a command line tool called Knife is available. Chef then distributes all updates and supporting resources (files, database dumps etc.) to all nodes which will perform the new tasks [KML+14].

A big advantage of stand-alone solutions like Chef is the adaptability to different used technologies. For example, the Oracle Orchestration Cloud Service is designed to work best with Oracle's cloud solutions. Therefore, using standalone services can be an advantage, especially if using different cloud environments (e.g. for a hybrid cloud). The *Resource selection* operations are performed by installing the Chef client on different nodes. Before uploading recipes to the server, Chef offers a test suite to prove the correctness of the scripts. *Resource deployment* is then achieved by starting a given recipe. Chef Automate

[3]https://puppet.com/ (10-29-2017)
[4]https://www.chef.io/ (10-29-2017)
[5]http://cloudify.co/ (10-29-2017)

gives an overview of the workflow and compliance of all clients and offers a solution for *Resource monitoring*. By allowing the user to address all nodes in case of failure, Knife can also be used for *Resource controlling*.

Orchestration and architectural design are mostly separated when using stand-alone solutions. That's why most of them have the same advantages and disadvantages as the Oracle Orchestration Cloud Service. An additional disadvantage is the heterogeneity of the domain specific languages used by different services. Therefore, an orchestration plan cannot be easily ported to another orchestration service. An exception is Cloudify, which uses the TOSCA standard for the design and orchestration.

3.2 Architecture Description Languages (ADLs)

An Architecture Description Language (ADL) offers a possibility to reason about the properties of a system in a common language. According to IEEE, an ADL is defined as "any form of expression for use in architecture descriptions" [DBB14]. By using a proven language, every stakeholder of a software project is able to understand the system's properties at a high level of abstraction. Good designed architectures offer a wide variety of advantages: it's easier to satisfy requirements and to react to changes. On the other hand, a bad designed architecture will most likely have disastrous consequences [GMW10]. The University of L'Aquila has published a list[6] of ADLs containing more then 120 entries. They considered every language that fits the IEEE definition. Because of the huge amount of languages, only some selected are exemplary described below and compared to the XML description of TOSCA's service templates. Even if TOSCA is no ADL, the standard contains a hierarchical description of software architectures which is very similar to an ADL's description. Because the framework presented in this bachelor's thesis is designed very generic, it is possible to adapt the later examples to work with one of the following ADLs as well.

Unified Modeling Language (UML)

The Unified Modeling Language (UML), invented by the Object Management Group, is one of the most known standards for graphical modeling of software systems and became the lingua franca for software development. While in 1995 only a small amount of organizations used modeling tools, this amount lays above 70% in 2008 for UML only [Wat08]. This growth happened due to the standardization of modeling processes as well as upcoming tools for analysis, design and implementation. Version 1 only incorporated some methods for language design, object-oriented programming and architectural design. Over time, UML was further developed to fit the respective standards of industry [15].

Since UML supports a wide variety of modeling tasks, different models are used to cover different problems. The models can be categorized in structural and behavioral. While

[6]http://www.di.univaq.it/malavolta/al/ (10-25-2017)

structural diagrams specify the composition of different components for different abstraction layers like classes, packages and deployment, behavioral diagrams are used to describe the expected behavior of a system. Both types of models are usually represented graphically but can also be represented as an XML Metadata Interchange (XMI) file containing all necessary information. In terms of TOSCA, a service topology can most likely be represented as a deployment diagram. A deployment diagram includes a system's software and hardware as well as their interrelation as an abstract description. It considers individual nodes which may run different applications. Execution environments can be specified as well as databases and connections. All components can be assigned with deployment specifications including configurational and parametric information like ports or version numbers [15]. Just like a TOSCA service topology, a deployment diagram cannot be instantiated but elaborated as concrete classes. A problem occurs if the user wants to specify a management plan for a given deployment diagram. Because UML was not designed for orchestration issues, the startup, update and termination of a whole system cannot be automated without extending the standard. Therefore, TOSCA should be preferred for the orchestration of a distributed system while the design of an architecture can be done using UML. At the end of the design process, this UML description must be converted to a TOSCA service topology however.

Architecture Analysis & Design Language (AADL)

The Architecture Analysis & Design Language (AADL) standard was released by SAE[7] International in November 2004 and provides modeling concepts for the design and analysis of performance critical application systems. Software, hardware and system components are described as well as their interaction in an abstract way. The focus lays on specifying and analyzing real-time embedded system models; whatsoever, even other kinds of systems can be modeled though. In order to describe a whole system, AADL distinguishes between three types of components: *application software* like processes, data or threads, *execution platforms* like processors or memory and *composites* such as, for example, another system. Every component is described with a unique identifier (name), its interfaces, sub-components and possible interactions. In addition to that, AADL offers a possibility to describe the interactions between components in different ways including *message passing*, *event passing*, *synchronized access to shared components* and *remote procedure calls*. Even properties like ports or version numbers can be specified [FGH06].

Because AADL was designed to enable validation and analysis of a systems' critical properties like performance, timing and dependability, it is widely used for the design of complex real-time safety-critical applications [RKK07]. It contains a huge amount of functionality for a fine-grained design. In other words, AADL was not designed for modeling distributed cloud applications. A lot of functionality is not useful in this domain and using the whole

[7]http://www.sae.org/ (10-26-2017)

framework would be overkill. For example, a cloud application doesn't need any specification about the different threads of a component. Therefore, AADL offers a level of abstraction that is too low for CC applications.

3.3 Pattern Detection

The framework described in this bachelor's thesis depends on the detection of patterns already included in a given software architecture. Different solutions exist for different ADLs. This section covers some pattern detection techniques including the framework by Wohlfarth [Woh17], which will be used exemplary later on.

3.3.1 Interactive DEsign Assistant (IDEA)

In their paper, Federico Bergenti and Agostino Poggi [BP00] provide a general-purpose technique for automated common pattern detection in UML class diagrams. The main idea is to improve a software design by proposing possible improvements to the detected patterns. In the first step, rules for every pattern are described using Prolog. As an example, the name of a class implementing the Factory pattern ends with the suffix *Factory*. Afterwards, these rules are applied on a XMI file of an UML diagram. Every pattern is associated with a set of possible improvements, so called critiques. A critique is rated low, medium or high, depending on its relevance on the pattern. IDEA provides two outputs: a pattern- and a todo-list. The pattern-list contains all detected patterns as well as their corresponding participants (classes) in the UML diagram while the todo-list contains all possible improvements ordered by their relevance.

Due to the abstract nature of patterns, only a subset of them can be detected automatically. To detect patterns like *Singleton* (only a single instance of a class is existent at a time and it exists one single, global access point to it), the UML model needs to include some information about the amount of possible instances of an object. This is not covered by the UML standard and therefore often expressed informally or not at all. Detecting such patterns is impossible because corresponding rules cannot be defined. Other patterns like the *Abstract Factory* and the *Observer* are expressed as concrete classes and their interrelationships. The *Abstract Factory*, for example, consists of (at least) four classes: the product, the concrete product, the creator and the concrete creator. Every concrete product inherits from product and every concrete creator inherits from creator. Finding such a substructure in a given UML diagram can be achieved by defining and evaluating corresponding Prolog rules of inference and is therefore possible [BP00].

IDEA uses rules of inference defined in Prolog to detect existing patterns in a given XMI/UML class diagram file. In addition to that, every pattern comes with a todo-list for possible improvements. This paper is a proof that the advantage of automated pattern detection was already known in 2000. Back then, most computer systems were monolithic and the term cloud computing as we know it today was not even existent. Therefore, IDEA is based on the detection of patterns in a single application's architecture; distributed or CC systems were not considered. The basic idea of defining and evaluating logical rules for pattern detection can, however, be adapted to all kind of architectures. Even to current ones.

3.3.2 Pattern Detection Framework for TOSCA-Topologies

In his bachelor's thesis, Wohlfarth [Woh17] describes a method to find proven patterns in an existing TOSCA architecture by detecting graph isomorphisms. A TOSCA service template is a topologically described architecture of a cloud application which can be represented as a graph. Components are described as nodes and relationships as edges between them. This graphical representation will then be used for the pattern detection process. Every pattern is described as a possible subgraph of the given architecture. The main task is to find a correspondence for every node of the pattern graph in the topology graph. If such a mapping exists, the pattern is marked as found. The output consists of a list of patterns which is divided into the categories *detected*, *high*, *medium*, *low* and *impossible*, depending on their probability. The whole pattern detection process can be described as follows:

1. **Keyword Search & Probabilities** The type of every component is determined by comparing its name to a predefined set of options. For example, every node whose name contains the term *tomcat* is labeled as *server*. Therefore, a knowledge base containing names and labels is mandatory. Based on the detected labels, some first estimations about the probability of each pattern can be done. Because some patterns like *elastic load balancer* and *elastic platform* are interlinked, their probabilities also are. Therefore, if one was detected, the probability of the other will be adjusted.
2. **Mapping of Topology** The TOSCA service template is mapped to a labeled graph.
3. **Subgraph Isomorphism** All possible subgraphs of the topology graph are created. Then, the VF2 algorithm will be used to detect all graph isomorphisms between subgraphs and pattern graphs. If an isomorphism was found, the probability of this pattern is set to *detected*.
4. **Result** The output is a list of patterns and their corresponding probabilities.

On the left hand side, figure 3.1 shows an example TOSCA architecture. In step 1, every component will be labeled and an initial probability for every pattern is set. Node names that contain predefined keywords like *Ubuntu* and *nginx* will be labeled with *Operating System (OS)* and *Server*, respectively. Because the root node contains the keyword *OpenStack*, the probability of *IaaS* is set to high while *PaaS* is set to low. Further, the probability for IaaS and PaaS related patterns is updated. A labeled version of the architecture's graph is shown on the right hand side of figure 3.1. Since the architecture is already depicted as a graph, step 2 can be omitted. In step 3, isomorphisms between all subgraphs and all pattern graphs are detected. In the example, an isomorphism for the *Node-based Availability* graph exists. Therefore, the *Node-based Availability* pattern is categorized as *detected*. Finally, the

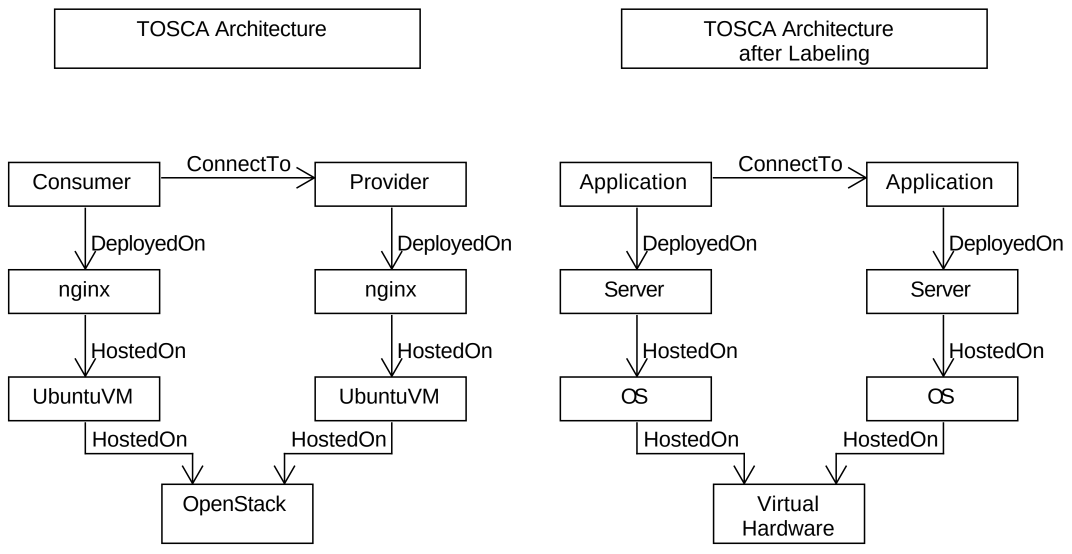

Figure 3.1: Pattern Detection Labeling Process

user gets feedback about all patterns and their probabilities. This example should only give a little insight into the pattern detection process and does not cover all detected patterns.

The pattern detection framework is strongly dependent on detecting graph isomorphisms. Therefore, the time complexity of the used algorithm plays an important role for the usability of the framework. Usually, the input graphs are very small (less than 20 nodes). That's the reason why an expensive subgraph isomorphism algorithm like VF2 is acceptable (runtime complexity best-case $\Theta(N^2)$ and worst-case $\Theta(N! * N)$) [CFSV04].

In contrast to IDEA, the pattern detection framework is applicable to every kind of architecture that can be represented as a graph. Although single application architectures can be analyzed as well as the architecture of distributed systems, the restriction to graphical representable patterns is also a disadvantage. Rules of inference are more powerful than using only graph isomorphisms. It is therefore possible to detect slightly more patterns using logical rules. However, the power of the detection framework is sufficient for our purpose. It will be used exemplary later on because a Java implementation and a graphical representation for some patterns already exists.

4 Approach

Patterns describe proven solutions for recurring problems. They are widely spread and commonly accepted. The reason is simple: patterns provide a common vocabulary and often have a positive impact on the designed application. The influence on the overall system is, however, difficult to predict. While advantages and disadvantages of patterns are mostly described informally in state-of-the-art literature, the framework described in this bachelor's thesis makes use of the NFR-Framework in order to evaluate the influence of patterns on different NFRs. Therefore, the informal description can be used to analyze the impact of each pattern in a systematic way using SIGs. By combining this information with a detection framework, the reason for positive and negative influences on an application can be identified. Furthermore, possible causes of unsatisfied NFRs can be named. The user may now decide to change some of the used patterns. Therefore, this framework offers a possibility for automated pattern inclusion. After identifying possible problem sources, a pattern countermeasure can be applied to the architecture. Because manual modeling of patterns is error-prone and time-consuming, it should be automatically achievable in one click. It is important to mention that this framework only has a supporting and advisory role during the design process; it's not meant to make own decisions. It will still be up to the designer to weigh up advantages and disadvantages of implementing different patterns.

Figure 4.1 shows the basic workflow of this framework. It consists of three main steps. In step one, an informal, textual pattern description will be analyzed in a systematic way using the NFR-Framework. This process is described in section 4.1. In the second step, patterns of a given architecture will be identified and their influence on the overall system

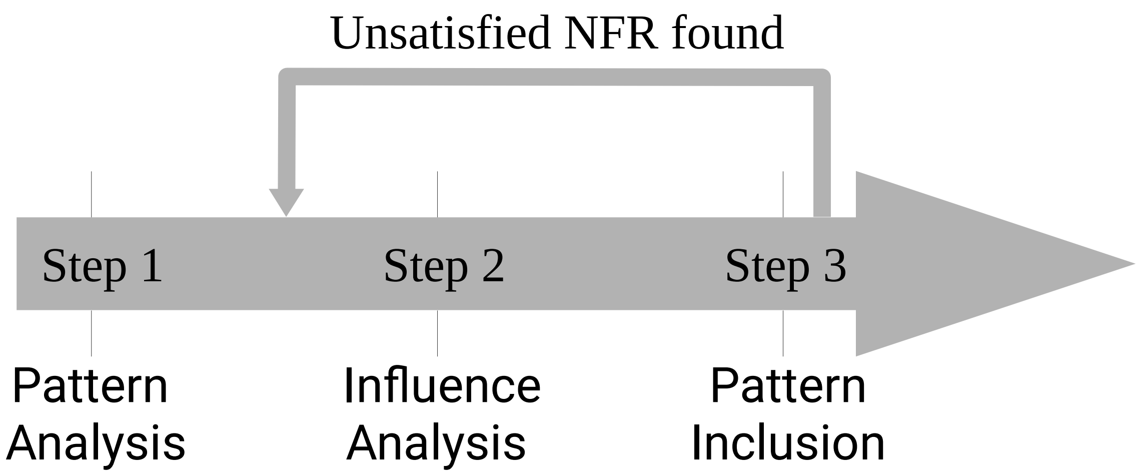

Figure 4.1: Three Steps of the Pattern Inclusion Framework

will be evaluated. The details will be explained in section 4.2. By applying predefined patterns to the architecture, it is possible to adjust unsatisfied NFRs in the last step. In section 4.3, the automated pattern inclusion process will be described, including some exemplary algorithms. Step one must be done only once for every pattern. Step two and three may be repeated if some unsatisfied NFRs remain even after the pattern inclusion process.

4.1 Pattern Analysis using the NFR-Framework

Although all patterns are different, their influence on NFRs can be analyzed in a systematic way using the NFR-Framework. In Gamma [Gam95], for example, a pattern's description usually contains information like

- **Intent -** A short description why the pattern is important.
- **Also Known As -** A list containing names the pattern is also known as.
- **Motivation -** A more detailed description of the pattern's purpose.
- **Applicability -** A list of circumstances in which the pattern can/cannot be applied.
- **Structure -** A depiction of the pattern's structure, often provided as a diagram in an ADL like UML.
- **Participants -** A description of classes or objects the pattern is composed of.
- **Collaborations -** A textual explanation of the relationships between the previously defined participants and their mandatory mutual guarantees.
- **Consequences -** A list of positive and negative influences on the application if the pattern will be applied.
- **Implementation -** An example implementation of the pattern, given as pseudo code or in a state-of-the-art language.
- **Known Uses -** Some use cases of the pattern that may be found in the wild.
- **Related Patterns -** A list of patterns that are somehow interlinked with the current pattern, that is, compatible, interchangeable or based on each other.

Especially important for our purpose are the points *Motivation*, *Applicability*, *Structure*, *Participants*, *Collaborations*, *Consequences* and *Implementation*. Even if not every pattern's description is organized into these categories, the important information can usually be somehow extracted.

To analyze an existing pattern, the *Motivation*, *Applicability* and *Consequences* have to be evaluated. The evaluation process consists of four steps:

1. Extract NFRs

2. Identify Operationalizing Softgoals
3. Refine given Operationalizing Softgoals
4. Create the SIG

Due to the missing impact of CC patterns on some NFRs from ISO 25010, only the gray scaled ones as well as their parents from figure 2.1 are considered in the following. If, for example, front-end patterns should be analyzed, *Attractiveness* or *Recognizability* play a more important role and should be therefore rather considered then *Resource-Utilization*. Further, every pattern influences other NFRs and should be evaluated per se.

In the following, the Watchdog pattern serves as an exemple to clarify every step of the process. Its definition reads as follows:

"*Applications cope with failures automatically by monitoring and replacing application component instances if the provider-assured availability is insufficient.*" [FLR+14]

An improvement of the application's availability seems to be the desired consequence of this pattern. Therefore, the NFR *Availability* can be extracted from the description. Because no other NFRs can be found, we now continue with the second step. The Operationalizing Softgoals are decisions that may have an impact on some NFRs of a system. The description of the Watchdog pattern mentions monitoring and management functionality such as replacing failed instances. Both operations can be used as Operational Softgoals. In addition to that, the *Applicability* description of the Watchdog pattern states that the Stateless Component Pattern must be implemented by every instance [FLR+14]. For this, another Operationalizing Softgoal will be added. To enable a more fine-grained analysis of the pattern, every Operationalizing Softgoal will be further refined in step three. This can be done by extracting additional information from the pattern's description. Monitoring, for example, consists of a realtime overview and a statistics of failure which provides additional information about failures over time. The management task can be refined by adding a new softgoal for restarting failed instances. Allocation of new resources would be another possible refinement. This is, however, not part of the Watchdog pattern as described in [FLR+14] and therefore not considered here. The Operationalizing Softgoal *Stateless Component Pattern* is a requirement. It is a precondition and not considered further because it won't change the current system in any way. In the fourth step, the final SIG will be created. The pattern itself always serves as root softgoal. All Operationalizing Softgoals which were identified in step two will now be added per *And*-Interdependency. A possible reading is: "The Watchdog pattern consists of Resource Management, Monitoring and the Stateless Component Pattern". Furthermore, the refinements of all Operationalizing Softgoals will be added to their corresponding parent per *AND*-Interdependency. Every NFR identified in step one will be added as NFR-Softgoal. If the NFR is just a subrequirement as defined in 2.1, its parent will also be added and connected to its child per *AND*-Interdependency. Now, it's up to the user to find other NFRs that were addressed by the Watchdog. Exemplary, all gray scaled items in figure 2.1 will be evaluated.

- **Operability -** Will be added as NFR-Softgoal because it's the parent of:
  - **Helpfulness:** Both, the real-time overview and the statistics of failure, provide an overview of the status of the application and are therefore helpful for the application owner. Thus, *Helpfulness* and its parent *Operability* will be added as NFR-Softgoal. Because the Operationalizing Softgoals *create realtime overview* and *create statistics of failure* are responsible for the additional helpfulness, both are connected per *Help*-relationship to the *Helpfulness* NFR.
- **Reliability -** Will be added as NFR-Softgoal because it's the parent of:
  - **Availability:** The Operationalizing Softgoal *restart failed instances* increases the *Availability* of an application by reducing the time an application cannot work properly because of an unavailable component. Therefore, the *restart failed instances* Operationalizing Softgoal will be connected to the *Availability* Softgoal per *HELP*-relationship.
  - **Fault Tolerance:** By restarting failed instances, the system is more fault tolerant because every failure will be handled immediately and the overall downtime is decreased. Thus, *restart failed instances* will be connected with the newly added *Fault Tolerance* NFR Softgoal, using a *Help*-relationship.
  - **Recoverability:** If a failure occurs, the defective component will be replaced instantly. Therefore, *Recoverability* will be added as NFR Softgoal and *restart failed instances* will be connected using a *Help*-relationship.
- **Performance Efficiency -** Will be added as NFR-Softgoal because it's the parent of:
  - **Resource-Utilization:** In order to run a Watchdog service, additional resources are needed. The NFR Softgoal *Resource-Utilization* and its parent will be added. In addition, the negative effect of the root Operationalizing Softgoal *Watchdog Pattern* on the *Resource-Utilization* Softgoal will be modeled using the *Break*-relationship.

The Watchdog pattern only influences the above listed NFRs, which means that all other NFRs (*Modularity, Reusability, Changeability, Maintainability, Integrity, Security, Portability, Adaptability, Transferability, Time-Behavior, Replaceability, Interoperability* and *Compatibility*) won't be added. The final SIG is depicted in figure 2.2.

This procedure must be applied to every pattern. The resulting SIG offers a comprehensible way to discuss a pattern's influence on different NFRs. Further, individual details are apparent from the refined Operationalizing Softgoals. In the following sections, the determined impact of each pattern will be used to evaluate an application's state and possible improvements. Even if the SIG itself won't be used any further, it can be helpful to reason about a specific design decision. For completeness, the SIGs of the Environment-based Availability and the Node-based Availability are contained in the Appendix. Both patterns consists of either an elastic infrastructure or an elastic platform. Therefore, both pattern's SIGs are very similar. The only difference is in their direct connection to the *Resource Utilization* softgoal. Environment-based Availability provides a common

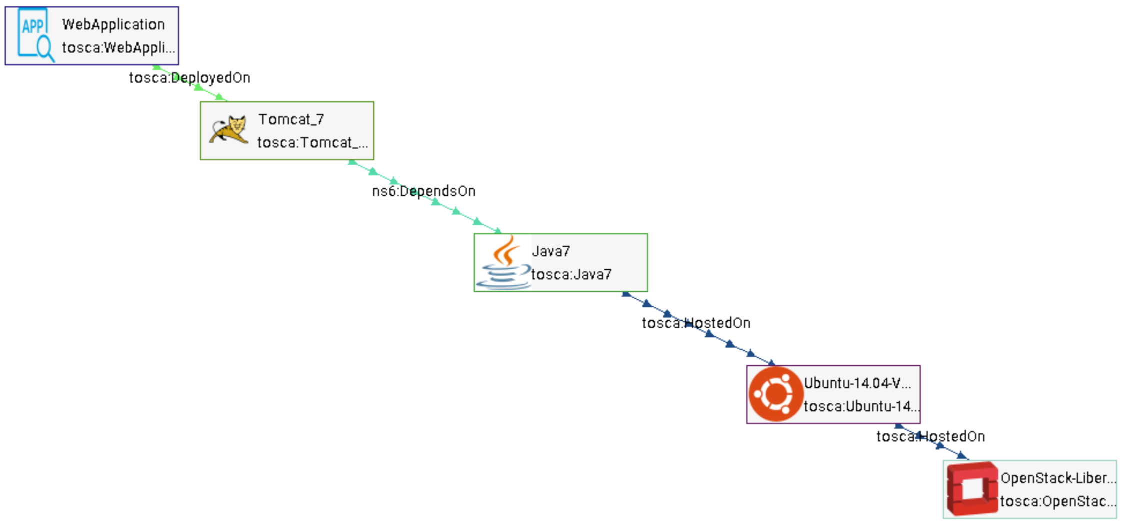

Figure 4.2: Example Topology for the Influence Analysis

environment which has a positive influence on the resource utilization. In contrast, Node-based Availability provides no such component and has therefore a negative impact.

4.2 Influence Analysis using Pattern Detection

The influence of every architecture's pattern should be evaluated in order to detect bottlenecks and bad design choices. This bachelor's thesis doesn't implement an own pattern detection algorithm; rather, any preferred detection framework is suitable. Two possible candidates are described in section 3.3. An extended version of the Pattern Detection Framework for TOSCA-Topologies will be used exemplary later on because the basic implementation doesn't cover individual patterns like *Watchdog*. In order to reliably evaluate an architecture, every pattern must have been evaluated with a corresponding SIG before. This can be achieved by following the instructions described in 4.1. The SIG shows the impact of an individual pattern on the whole application. By combining these information with the detected patterns, it is possible to reason about the positive and negative influences every pattern has on the overall application. In addition to that, the influence of every unapplied pattern on the system's NFRs can be evaluated. This enables the user to weigh up other possible options against their current present correspondents.

The influence analysis consists of three steps:

1. Detect each pattern's probability in a given architecture
2. Evaluate every pattern
3. Inform the user about the results

Every step will be further explained by using topology 4.2 as an example. The six nodes *OpenStack*, *Ubuntu*, *Java*, *Tomcat*, *Web-Application* and *Watchdog* are all based on each other, creating one big dependency chain. In step one, the probabilities for all patterns must be determined. The modified detection framework came to the following results:

- **Detected:** Platform-as-a-Service, Environment-based Availability, Watchdog
- **High Probability:** Execution Environment Pattern, Public Cloud
- **Medium Probability:** $\emptyset$
- **Low Probability:** Elastic Platform, Node-based Availability, Elasticity Manager, Elastic LoadBalancer, Elastic Queue, Relational Database, Message-oriented Middleware
- **Impossible:** Infrastructure-as-a-Service, Elastic Infrastructure

In step two, the gathered information will be combined with the influence a pattern has on the NFRs. It is now possible to reason about adding and removing patterns to and from the architecture. Using the results of the Watchdog analysis, the conclusion should be that removing the Watchdog pattern would result in less *Resource Utilization*, but the overall Reliability and Operability of the system would also decrease significantly. Now it's up to the user to weigh up advantages and disadvantages of removing the Watchdog and making a decision. Furthermore, unapplied patterns can be evaluated to see if they possibly influence the NFRs in the desired way.

4.3 Automated Pattern Inclusion Process

After identifying the trade-offs between different patterns, the user may want to apply some new patterns and remove others. Because this process can be very time-consuming and error-prone, the whole procedure should be automated. This way, the result is always the same no matter who the designer is and it can be achieved in almost no time. If the automated pattern inclusion is such a game changer, the question comes up why it has been so rarely investigated yet. There are multiple answers for that. First, every pattern is unique, which means that every pattern must be implemented in it's own way. There is usually no possibility to reuse the code of one pattern for another. Considering more than 1500 existing patterns [HC07], the implementation process would take a lot time; even for a large team. The second reason is the complexity of some patterns. While patterns like *Watchdog* are very simple to implement, other patterns like *Node-based Availability* are harder to realize. Both implementations are covered later in this section. Before, the general idea behind the algorithms and the necessary preparation is described.

4.3.1 Preparation

Every topology can be represented as a graph. Therefore, the pattern inclusion algorithms work on graphs and make use of standard operations like *getAllNeighbors*, *addVertex*, *addEdge*, *getEdgeBetween(Vertex 1, Vertex 2)* and so on. In the first step, a given topology description (e.g. a TOSCA service template) must be parsed into a graph. Every component corresponds to a node and every interdependency to an edge. The vertices are usually provided with additional information like the component's name, its type and version number. Interdependencies are mostly also labeled. It can be very helpful to add most of the available data to the graph. The more information is available, the easier it is to implement pattern inclusion algorithms.

Role-Labeling of Components

The identification of an individual component's role in the overall system may come in handy. Therefore, every component will be additionally labeled with one of the following keywords:

- **Virtual Hardware -** describes a *Virtual Hardware* component that offers virtual hardware and runs an *Operating System*. OpenStack can be exemplary named.
- **Operating System -** describes an *Operating System* component like Ubuntu or Windows that is hosted on a *Virtual Hardware* and runs other applications and services.
- **Server -** defines a *Server* component that is solely used to host other applications and services. An example would be Apache Tomcat.
- **Messaging -** will be used for two different node types: every message broker like Apache ActiveMQ and every message topic that applications subscribe to will be labeled with *Messaging*.
- **Storage -** describes all kind of data storage nodes like databases and database management systems (MariaDB, MongoDB, ...).
- **Service -** defines different services that are mandatory for the overall system but no explicit part of the application. Execution environments like Java and Python fall into this category.
- **Application -** every instance of an application that has not been labeled yet will be labeled as *Application*.

Before each component can be labeled, lists containing identifying keywords must be created. As an example, the list for *Service* contains, amongst other things, the entries Java and Python. If it is possible to match a component's type with one of these entries, the component will be labeled as *Service*. If not, the process continues with the next list. Because every application is unique, listing their names is not possible. Therefore, if the component's type has not matched any of the other entries, it will be labeled as *Application*. In order to achieve good results, it is necessary to keep the lists up to date.

Normalization of the Inputgraph

Creating an application's architecture, there are many choices to make. Every designer has his own way of representing interdependencies and correlations in an architecture. All of them have advantages and disadvantages but none of them is wrong. Problems may occur when an algorithm depends on a specific design convention. If this convention was hurt, the algorithm doesn't work properly. In order to prevent such situations, all input graphs should be normalized. Normalization means that different design choices are converted into a single representation of the graph which will then be used to apply the algorithms. Thus, different representations will be handled the same way because the input graph will always be the same. However, the normalization process depends on the used ADL and their possibilities to describe same behavior in different ways. Thus, the normalization must be implemented individually for different ADLs. Using TOSCA topologies as an example, execution environment nodes can cause problems if the Node-based Availability pattern should be applied. Applications and their execution environment can't be hosted on two different operating systems because the two nodes are strongly interdependent. A more detailed description including a possible solution for TOSCA topologies will be covered in the following section.

4.3.2 Algorithm Concepts

Using the node's information, algorithms for the automated inclusion of different patterns can be developed. As mentioned before, every pattern is unique and must therefore be implemented in its own way. This task must be done once and as soon as a sufficient set has been built, it can be reused multiple times and ease the process of software design. Exemplary, algorithms for the Watchdog, the Environment-based Availability and the Node-based Availability pattern will be described below.

Watchdog

The Watchdog is defined as an application that copes with failures of components automatically. It monitors other application's instances and replaces them with a redundant correspondent if necessary [FLR+14]. This description leads to the conclusion that a new application component representing the watchdog must be created. Furthermore, it is said that it monitors other application's instances. This can be achieved by creating a relation between the newly created Watchdog component and all other application instances. The functionality (monitoring and restarting instances, etc.) cannot be modeled in TOSCA and, therefore, won't be part of the algorithm.

In order to include the Watchdog pattern in an existing graph, all other nodes must be labeled first. Then, a new node must be created. The node's type will be set to *Watchdog*. Using this convention, the implementation of multiple Watchdogs can be prevented. In case another node with the type *Watchdog* is already existent, the algorithm doesn't change the architecture and terminates. Else, the Watchdog component will be added to the graph. In the final step, the newly added node will be connected with a *connectsTo* relationship to every other node whose role was set to application. An illustration in pseudo code is shown in algorithm 4.1. The operating system on which the *Watchdog* will be hosted is not specified. This is due to the fact that there may be multiple operating system nodes in the architecture and it is not possible to determine the preferred one. If the architect wants to specify an OS node hosting the *Watchdog* component, this relation has to be modeled manually.

Algorithm 4.1 Watchdog

```
procedure APPLYWATCHDOGPATTERN(Graph $G$)
    LABELGRAPHNODES($G$)
    $N_{watchdog} \leftarrow$ NewNode($Type \rightarrow Watchdog$)
    AddNode($G, N_{watchdog}$)
    for all $N_{next} \in$ NodesOf($G$) do
        if RoleOf($N_{next}$) = Application then
            AddRelation($G, N_{watchdog}, N_{next}, connectsTo$)
        end if
    end for
end procedure
```

Environment-based Availability

If a cloud provider guarantees the availability of an environment that can be used to deploy own applications, the provider offers Environment-based Availability [FLR+14]. While Environment-based Availability is no pattern per se, it is strongly coherent with the Execution-Environment, which is a real pattern. In the Environment-based Availability, the provider has to offer and guarantee the availability of an execution environment for applications. Because a TOSCA topology contains no information about the provider's services, it can't be represented properly in a TOSCA architecture. The Execution-Environment pattern, on the other hand, can be easily represented in a TOSCA architecture. It consists of a user-defined execution environment node and multiple other nodes that depend on it. An example can be found in figure 4.3. Multiple application instances share the same services and/or servers, which run on the same operating system. If, for example, two applications depend on the Java Runtime Environment (JRE), they may either use the same JRE or two different ones, hosted on two different operating systems. In case of the Environment-based Availability, the CC provider offers a common runtime, which will then be used by all applications. If, on the other hand, the user deploys a JRE himself and it will be used by at least two different applications, the Execution-Environment pattern has

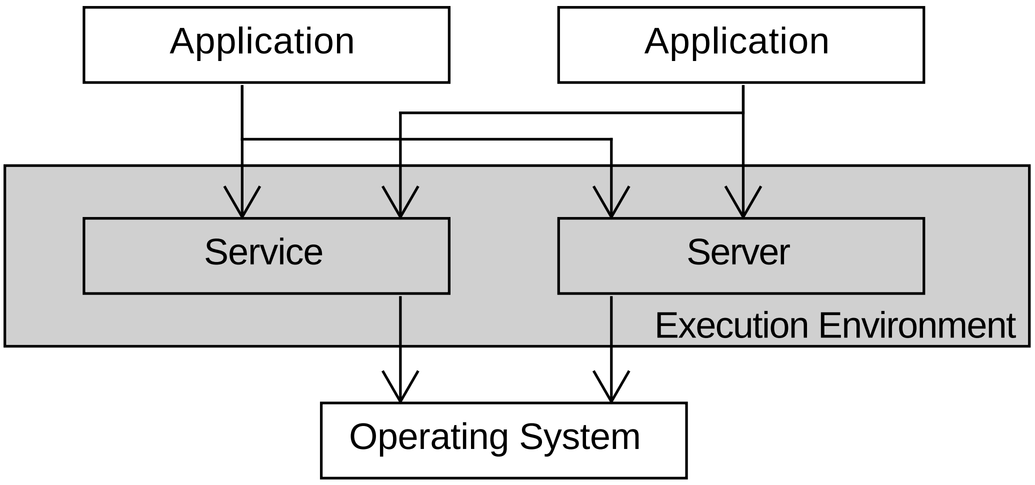

Figure 4.3: Execution Environment Pattern as defined in [Woh17]

been applied. Because both patterns are very similar and Environment-based Availability can't be represented in a TOSCA topology, it will be further treated as a synonym for the Execution-Environment pattern.

In order to apply the Environment-based Availability to a given architecture, all service- and server nodes need to be identified by labeling the graph. If the type of multiple services and servers, respectively, is the same, they will be merged. This way, only one instance of a given execution environment will be left over. Algorithm 4.2 describes a way to apply the Environment-based Availability pattern automatically to a given graph. In order to merge two nodes N_1 and N_2, the target of every incoming edge to N_1 will be set to N_2. Furthermore, the source of all outgoing edges of N_1 will be set to N_2. Thereafter, N_1 can be removed (including all incoming and outgoing edges) without violating the integrity of the overall system. After this process finished, only one single instance of every type of service and server node remains. In addition, the relationships have been modified to keep the system's integrity. A problem may occur if the merged instances have ports or similar configurations defined. If an application depends on an open port defined in N_1, this node should not be merged with a N_2 because the merging process just re-adjusts the relationships. If N_2 hasn't defined the same port as open, the merging process could break the system. This is, however, not considered here. Rather, it is left as an open issue for future work.

Node-based Availability

In case the provider guarantees the availability of individual nodes instead of the whole environment, Node-Based Availability is given [FLR+14]. Figure 4.4 shows an example for the Node-Based Availability. Just as the Environment-based Availability, it is no pattern per

Algorithm 4.2 Environment-based Availability

```
procedure MergeTwoNodes(Graph $G$, Node $N_1$, Node $N_2$)
    for all $e(v_1, v_2) \in$ EdgesOf($N_1$) do
        if $e.v_1 = N_1$ then
            e.v$_1 \leftarrow N_2$
        else if $e.v_2 = N_1$ then
            e.v$_2 \leftarrow N_2$
        end if
        RemoveNodeWithEdgesFromGraph($N_1, G$)
    end for
end procedure
procedure ApplyEnvironmentBasedAvailability(Graph $G$)
    LabelGraphNodes($G$)
    for all $N_1, N_2 \in$ NodesOf($G$) $\wedge N_1 \neq N_2$ do
        if TypeOf($N_1$) = TypeOf($N_2$) then
            if RoleOf($N_1$) = RoleOf($N_2$) = (Service $\vee$ Server) then
                MergeTwoNodes($G, N_1, N_2$)
            end if
        end if
    end for
end procedure
```

se. Rather, it only describes the type of availability the provider offers. Because the user probably wants to make use of the offered availability, he wants every application to run on a separate underlying and software-stack. Therefore, the Node-based Availability will be treated as a pattern where every Node is independent from the underlying infrastructure of others. The only node that will be shared amongst all application stacks is the virtual hardware component. In case one node fails, other nodes are not affected as long as they don't depend on the failed one directly. To stay with the JRE example: if two applications require the same JRE, they will still use two different JRE nodes. In case one of these node breaks, only the depending application stops working but the other still remains intact. If they have shared the same execution environment, both of them would've been broken. This definition will be used every time it is referred to Node-based Availability even if it is not generally applicable. Rather, this definition only works for an IaaS provider. If a PaaS is provided, it is possible that there is only one single OS node, too. Nevertheless, the availability of all individual hosted nodes can still be assured. The structure shown in figure 4.4 describes the Node-based Availability that will be referred to from now on. For unambiguous explanations later on, all other kinds of Node-based Availability will be neglected.

The implementation process is more complicated than the others before. After labeling all nodes with their corresponding role, all application nodes are selected. Using breadth-first iteration, a subgraph for each application node is built. The subgraph always includes the application node and all children that are directly or indirectly connected to it, except for

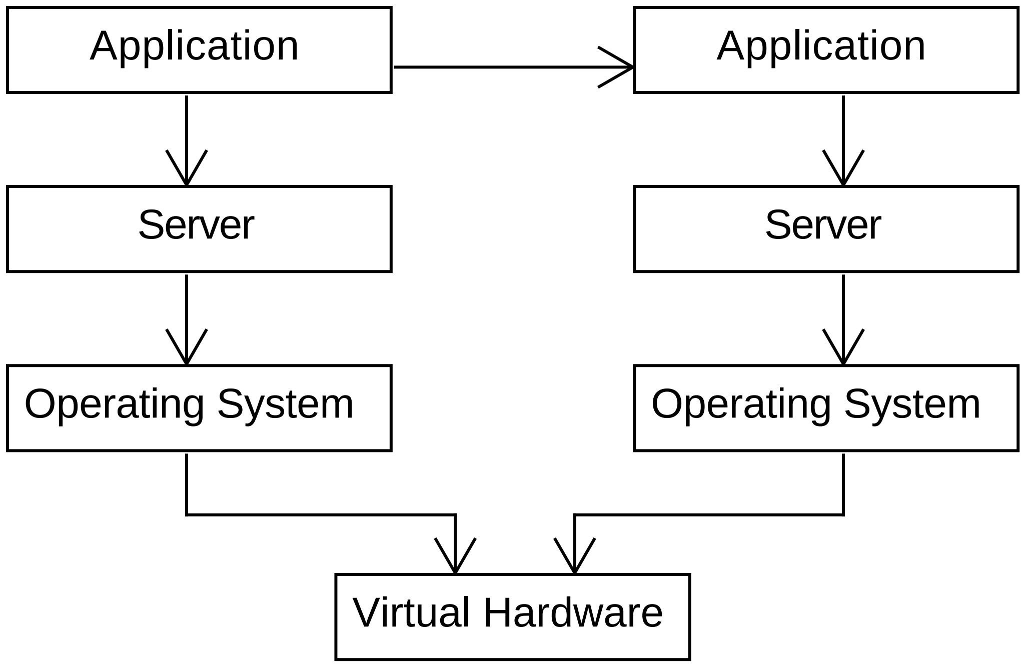

Figure 4.4: Node-based Availability as defined in [Woh17]

other application nodes. The interrelations between the remaining nodes are retained, too. It is important to mention that the nodes and edges are exactly the same as in the input graph, no copies. This becomes important when the subgraphs will be merged again. Now, each subgraph will be forked. That means that every node which has no application-, virtual hardware- or messaging-role will be copied and the incoming and outgoing edges are adopted. The initial node will then be deleted. After each subgraph has been forked, they will be merged together again. The nodes whose role is application, virtual hardware or messaging haven't been copied. Therefore, they are all the same and merged into one single node. All other nodes that were copied in the forking process are new ones which means that they won't be merged. Rather, every application has its own individual substructure now and is independent from the underlying infrastructure of other application nodes. For a better understanding, algorithm 4.3 describes this whole process in pseudo code.

As an example, figure 4.5 shows an initial input graph on the left side. The nodes are colored for a better understanding only. After the application nodes have been filtered out and their underlying structure has been evaluated, the resulting subgraphs are depicted on the right side. Now, each one will be forked. Therefore, all nodes whose role is not application, virtual hardware or messaging will be copied, including their incoming and outgoing relations. Further, the nodes that were copied will be deleted. The result can be found in figure 4.6. In the last step, all forked subgraphs will be merged again. Because

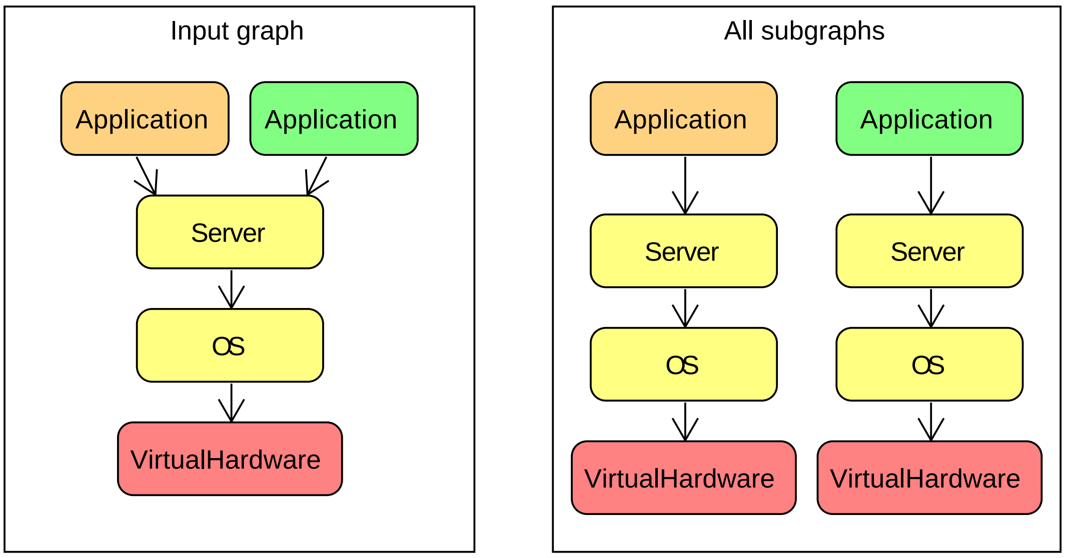

Figure 4.5: Node-based Availability Subgraph Creation

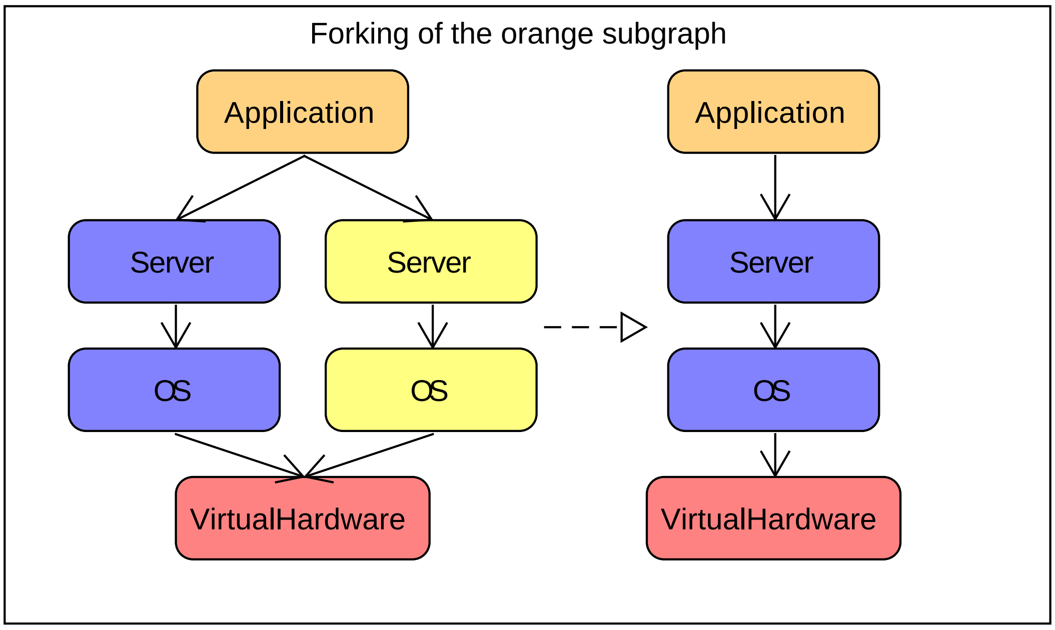

Figure 4.6: Node-based Availability Subgraph Forking

Algorithm 4.3 Node-based Availability

```
procedure GetSubGraphForNodeInGraph(Node $N_{app}$, Graph $G$)
    $G_{sub} \leftarrow$ NewDirectedGraph
    AddNode($G_{sub}, N_{app}$)
    for all $N_{next} \in$ BreadthFirstIterator($N_{app}, G$).next do
        if RoleOf($N_{next}$) $\neq$ Application $\wedge$ NodeHasPredecessorInGraph($N_{next}, G_{sub}$) then
            AddNodeWithRelationsToGraph($N_{next}, G_{sub}$)
        end if
    end for
    return $G_{sub}$
end procedure
procedure ForkSubGraph(Graph $G_{sub}$)
    for all $N_{sub} \in$ NodesOf($G_{sub}$) do
        if RoleOf($N_{sub}$) $\neq$ (Application $\wedge$ VirtualHardware $\wedge$ Messaging) then
            $N_{clone} \leftarrow$ CloneNodeWithRelations($N_{sub}$)
            AddNode($N_{clone}, G_{sub}$)
            RemoveNodeWithRelationsFromGraph($N_{sub}, G_{sub}$)
        end if
    end for
end procedure
procedure ApplyNodeBasedAvailability(Graph $G$)
    LabelGraphNodes($G$)
    NormalizeGraph($G$)
    $G_{return} \leftarrow$ NewDirectedGraph
    for all $N_{app} \in$ ApplicationNodesOf($G$) do
        $G_{sub} \leftarrow$ GetSubGraphForNodeInGraph($N_{app}, G$)
        ForkSubGraph($G_{sub}$)
        AddSubGraph($G_{return}, G_{sub}$)
    end for
    return $G_{return}$
end procedure
```

the hardware node is the only one that is still the same in both subgraphs, the resulting graph will also have one single VirtualHardware node only. Both other nodes (server and OS) can't be merged because they are now different ones. That's the reason why they are now existent twice. The Node-based Availability has been successfully applied. The result is shown in figure 4.7.

Normalization

The normalization of TOSCA topologies is very important. The Node-based Availability algorithm, for example, produces wrong output if it is applied on a non-normalized graph. The problem can be explained using a small graph, containing only three nodes. One such

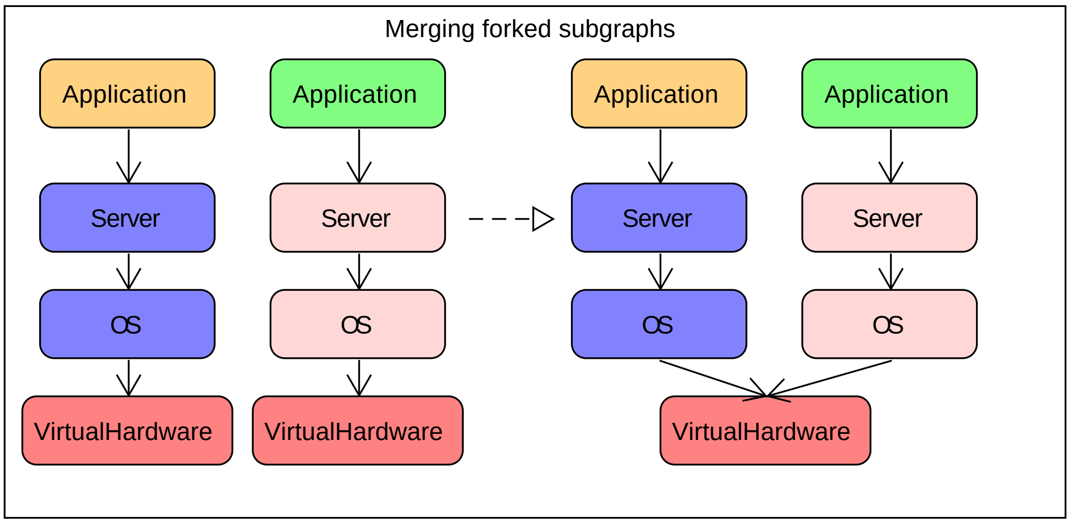

Figure 4.7: Node-based Availability Subgraph Merging

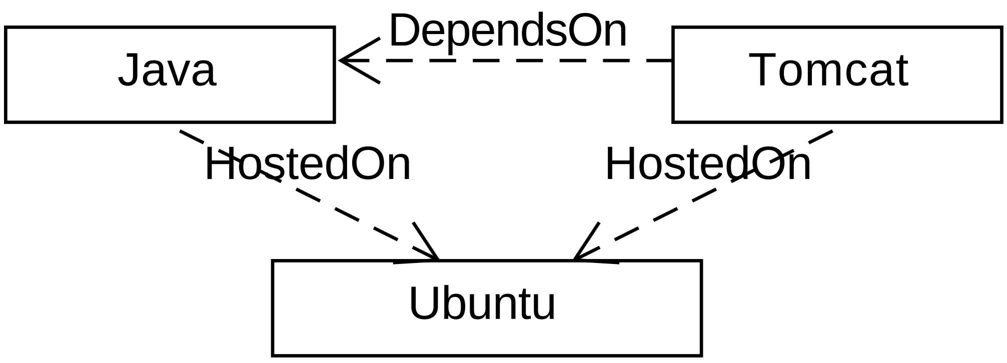

Figure 4.8: Triangle Problem in a TOSCA Topology

graph is depicted in figure 4.8. The root node is always an operating system like Ubuntu. Further, there are at least two nodes hosted on this OS. Out of all these nodes, at least one has to depend on another node that is an execution environment. In our example, Java is the execution environment and Tomcat is the server that depends on it. Running an application without its execution environment is not possible. Therefore, they should not be forked and separated from each other in the Node-based Availability algorithm. Because of the triangular nature of the problem's structure, it will be further referred to as Triangle Problem.

The reason why the Triangle Problem makes the Node-based Availability algorithm fail is because after forking, the execution environment and the node that depends on it would

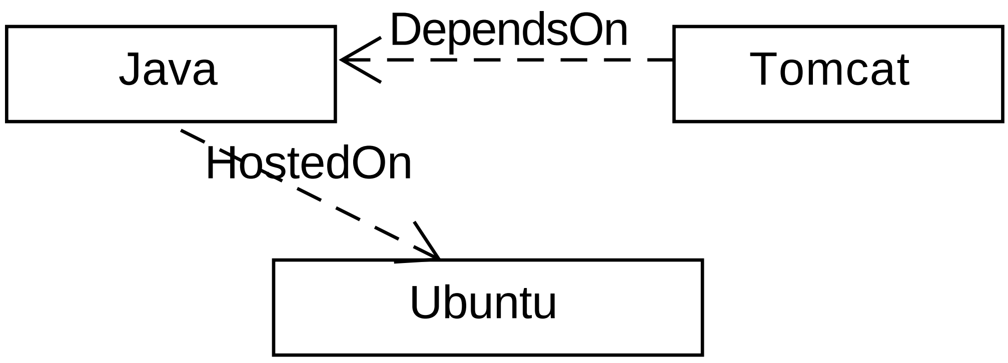

Figure 4.9: Solved Triangle Problem in a TOSCA Topology

be hosted on two different operating systems. Tomcat, for example, depends on Java and can't be hosted on an OS that has no Java available. Thus, the graph must be normalized in order to solve the Triangle Problem before the Node-based Algorithm can be applied. This process guarantees that execution environments and their depending nodes are always hosted on the same OS.

Normalization is pretty simple. As figure 4.9 shows, the *HostedOn* relation between the *DependsOn* source and the operating system must be removed. Removal of the relation should be no problem because the remaining *DependsOn* connection still induces that the source node is hosted on the same OS as the target execution environment. Because the depending nodes are now only implicitly hosted on the operating system, the Node-based Availability pattern's algorithm won't fork the underlying OS node.

Algorithm 4.4 shows a possible implementation in pseudo code. In order to normalize a given graph, all environment nodes must be identified first. This can be done using a list containing keywords describing possible execution environments like *Java* and *Python*. The process to determine these nodes is similar to role-labeling from then on. All nodes that depend on an environment node are then filtered out. In case a dependent node is connected per *HostedOn* relationship to the same OS as the environment node, this connection will be removed.

Algorithm 4.4 Normalization

```
procedure NormalizeGraph(Graph $G$)
    for all $N_{env} \in \mathsf{EnvironmentNodesOf}(G)$ do
        $N_{envOS} \leftarrow \mathsf{OSNodeOf}(N_{env})$
        for all $N_{dep} \in \mathsf{DependantNodesOf}(N_{env})$ do
            $N_{depOS} \leftarrow \mathsf{OSNodeOf}(dependantNode)$
            if $N_{envOS} = N_{depOS}$ then
                $\mathsf{RemoveRelationBetween}(N_{dep}, N_{depOS})$
            end if
        end for
    end for
end procedure
```

5 Implementation

With this bachelor's thesis comes a Java project that allows the user to load an existing TOSCA service template's .csar archive and apply some predefined patterns in one click. The GUI provides an overview of the detected patterns, their influence on the NFRs as well as a graphical depiction of the topology graph itself. This way, the user can easily follow the changes made. Chapter 4 covered the basic idea behind the implementation of different algorithms to apply patterns on a given architecture. In the first step, the input graph must be normalized and all its node have to be role-labeled. Only after that, pattern inclusion is possible. This chapter provides a short overview of the GUI itself as well as a listing of the used technologies. Further, the basic code structure will be explained and the expandability of this project will be discussed.

5.1 Pattern Inclusion Application (PIA)

The Pattern Inclusion Application offers a graphical user interface for the pattern inclusion process. Figure 5.1 shows the GUI, which is separated into three different sections. Section A shows all available patterns and their probability. This view is updated every time the topology changes. The user can select a given pattern from the list. Its influence on the topology's NFRs will be shown in section B. Besides the name of the selected pattern, the influence on every NFR from ISO25010 is shown in an hierarchical tree view. If the NFR and its connection is colored green, the impact is positive. If the color is red, the impact is negative. The strength of the impact is shown using two, one or none plus and minus icons, respectively, in front of every NFR. Only if a NFR is not influenced by the selected pattern at all, the color is white. A button is placed on top of that view which allows the user to apply the selected pattern on the topology. The update will be visible immediately; section A will show a new list of detected patterns and section C will show the updated graph. It is possible to move the topology graph from section C with the mouse. If the mode of the display area was set to 3D, movement becomes rotation. In addition to that, zooming is possible using the mouse wheel. Every time a change was made, the graph will be updated. Further, a new graph will be displayed if a new service template was loaded. Opening another .csar archive is possible using the menu bar's $File \rightarrow Open$ dialog.

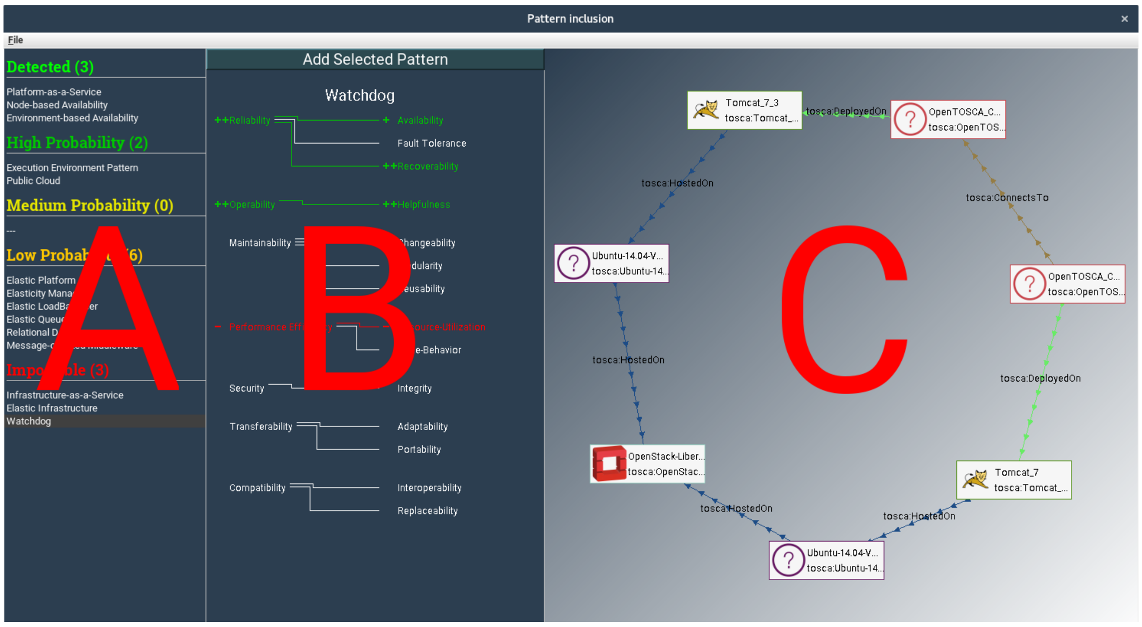

Figure 5.1: Pattern Inclusion Application GUI

5.2 Used Technologies

The whole project was realized using Java 8 which is the recommended version at this time. All used libraries and some features are described in this section.

5.2.1 JGraphT

In order to represent topologies internally as a graph data-structure, the JGraphT[1] library version 1.0.1 was used. It allows the user to simply create and modify graphs, using custom classes for nodes and relationships. Different graph types like weighted, unweighted, directed, undirected, labeled and unlabeled graphs are supported. Further, some basic algorithms and iterators like the VF2 algorithm and the breadth-first iterator are also provided. JGraphT is licensed under the LGPL[2] and the EPL[3].

By using custom entity-classes for nodes and relationships, a given topology can be transferred into a graph data-structure. All algorithms are then just performed on this graph; not on the topology itself. However, this is no problem because a backward conversion from graph to topology is possible, even if it is not supported by the implementation yet.

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

5.2.2 Jblas

Jblas[4] is a linear algebra Java library which offers data-structures like matrices and vectors as well as computational operations as, for example, matrix multiplication and dot product. It is a light-weight wrapper of BLAS[5] and LAPACK[6]. In the Pattern Inclusion Application, jblas (V.1.2.4) is used for the calculation of the topology graph's 2D and 3D self-aligning depiction. By saving a 2D/3D position for every node and calculating a direction vector for each node, the graph can be aligned automatically. Furthermore, using rotation matrices, the whole graph can be rotated around all axes, which gives a better overview over the topology. Jblas is licensed under a revised version of the BSD 3-clause license.

5.2.3 Pattern Detection Framework

In his bachelor's thesis, M. Wohlfarth [Woh17] describes a framework for pattern detection in TOSCA topologies. It was initially designed as an extension for openTosca's Winery. Therefore, the implementation had to be slightly modified to work with the the graph representation used by the pattern inclusion application. To separate the detection logic, the modified pattern detection framework has been exported and was added to the project as a Java library. Furthermore, it was expanded in the inclusion application to increase the amount of detectable patterns.

5.2.4 Java Swing

The GUI was implemented using Java Swing. It is a part of the Java Foundation Classes (JFC), which includes a set of features for building GUIs. Swing was used due to the fact that no additional libraries are necessary and Swing already covers a lot of functionality needed for the implementation. One of the two most important features are the Swing GUI Components, which include basic GUI related elements like buttons, taskbars and frames. The second significant feature is the Java 2D API. It was designed to easily create and modify 2D graphics [Ora]. This feature was used for the graphical representation of the architecture's graph.

5.3 Code Structure

The implementation consists of four big packages: GUI, Parser, Framework and Utilities. The GUI package contains all classes necessary for the depiction and functionality of the main window and its components. The Parser package includes all classes that are used to

[4]http://jblas.org/ (16.11.2017)
[5]http://www.netlib.org/blas/ (16.11.2017)
[6]http://www.netlib.org/lapack/ (16.11.2017)

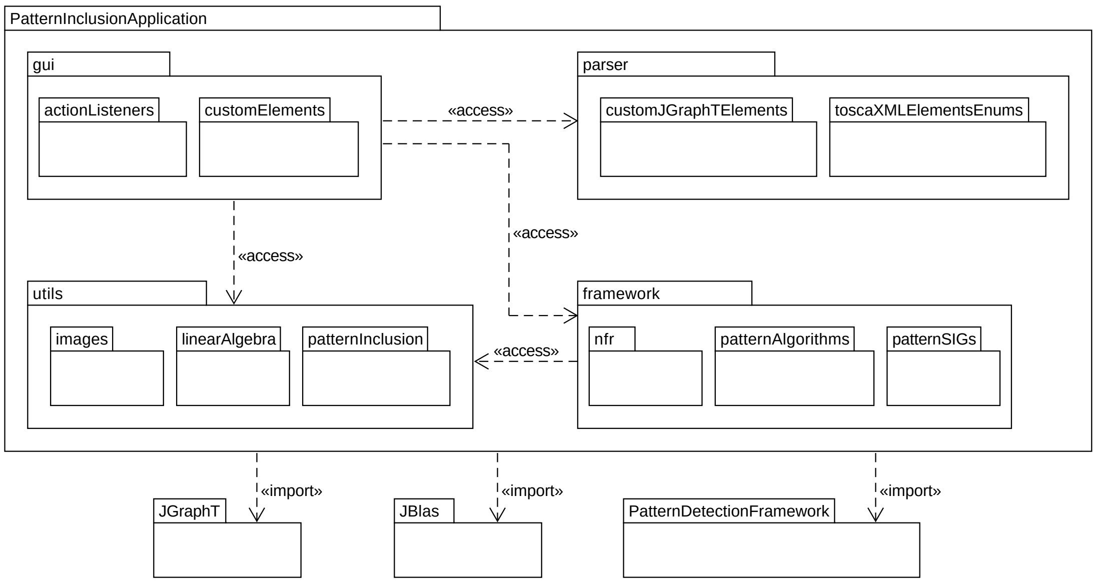

Figure 5.2: UML Package Diagram of the Pattern Inclusion Application

convert a given .csar archive to a JGraphT data-structure, including custom graph nodes and relations. Further, the Framework package consists of all algorithms and supporting classes the framework described in this bachelor's thesis is composed of. The fourth package contains Utility functionality for linear algebra, image editing and the framework itself. The interrelations and dependencies are shown in figure 5.2. The GUI package, which is the core, makes use of all other packages in order to display the topology graph and performs the inclusion algorithms. Further, the *framework* package accesses a modified version of the pattern detection class, which is also defined in the utility package. For completeness, the imported libraries are also depicted. In this section, the four main packages and their structure will be explained in detail.

5.3.1 GUI Package

Containing different classes for the creation of the application's window and its components as well as the main method, the GUI package is the project's core. This package contains three custom elements that are located in the sub-package *customElements*: First, the *patterListBar*, which can be used to show a list of detected patterns as well as their probability. Second, the *dependencyBar*, which depicts the influence a selected pattern has on different NFRs. Both bars are interlinked using the *BarConnector* class. The third element named *grapharea* defines a custom element that can be used to display a self-aligning graph in 2D and 3D, respectively. All action listeners concerning the main window are located in the sub-package *actionListeners*.

5.3.2 Parser Package

The parser package contains classes for the extraction of .csar archives and information about the creation of a topology graph. *CsarZipHandler* is a singleton which handles the unpacking of archives, parsing and returning of a JGraphT object. In order to gather necessary information from the XML files, the sub-package *toscaXMLElementEnums* contains the corresponding XML-specifiers. Further, the *customJGraphTElements* sub-package includes two classes: one for custom JGraphT nodes and one for custom edges. Using these sub-packages, the *ToscaParser* class manages extraction and graph creation.

5.3.3 Framework Package

The *framework* package contains entities and functionalities necessary for the pattern inclusion framework as, for example, a class for role labeling of a given topology graph. The first sub-package *nfr* includes an entity class for NFRs, as well as different factories used by the *patternSIGs* package. These factories build dependency trees based on ISO 25010 for an internal representation of NFRs and their interdependencies. The *patternSIGs* package contains a collection of classes representing the influence a specific pattern has on different NFRs. These information were extracted from the corresponding pattern's SIG. A collection of available pattern inclusion algorithms is stored in the *patternAlgorithms* sub-package.

5.3.4 Utility Package

While the other three packages are for one single purpose only, the *utils* package contains a mix of different utility classes. The first sub-package *images* offers an *ImageUtility* class that can be used for image manipulation like dyeing a BufferedImage. Sub-package *linearAlgebra* includes a class for basic matrix operations like rotation in 2D and 3D. The last sub-package *patternInclusion* contains an extended version of the *Detection* class by [Woh17].

5.4 Expandability

The Pattern Inclusion Application is just a prototype. In order to use it efficiently, more pattern algorithms need to be added. This is no problem, though. The only precondition is that every new algorithm must implement the *IPatternApplicant* interface. This interface provides one single *apply* method, which takes the current topology graph as input parameter and returns a version of the graph where the pattern has been applied. By implementing a custom version of this method, every arbitrary pattern can be added to the input graph. It is important to mention that the input graph won't be modified; rather, a copy including the new pattern will be returned, making it possible to compare the result with the input graph.

6 Validation

In order to validate the functionality of the framework and the algorithms, this chapter deals with an example TOSCA topology and explains the inclusion process for different patterns step by step. Role-labeling as well as normalization and pattern inclusion will be covered.

The pattern analysis as well as the influence analysis can't be validated without empirical data. Gathering information about the satisfied NFRs of a system requires a lot of time. *Availability*, for example, is the ratio of the operability to a system's up-time. Therefore, the system must be tested for a long period of time in order to achieve good results. For this bachelor's thesis, it is sufficient that the NFR-Framework has already been validated and its operability has been proven over time. This chapter only covers the validation of the implementation.

Figure 6.1 shows an example TOSCA topology. This topology has been chosen because it is very suitable to explain every step of the inclusion process for the Watchdog, the Environment-based and the Node-based Availability pattern. First, every component will be labeled using different keyword lists. Then, depending on the algorithm, the graph must be normalized. Only after these steps, the final pattern inclusion algorithms can be applied.

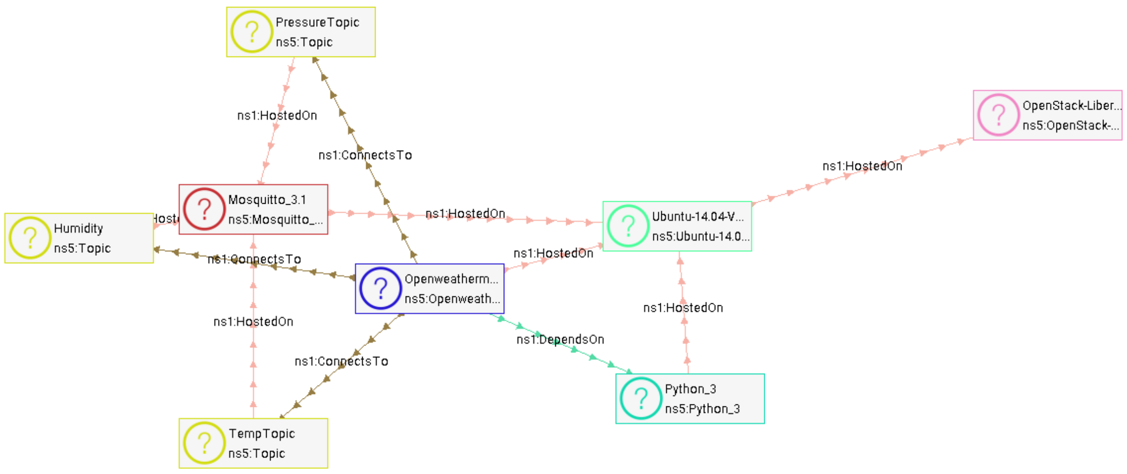

Figure 6.1: Openweathermap TOSCA Topology

6.1 Role-Labeling

Let's assume there are five different lists. The name and content of these lists is exemplary listed below:

1. **VirtualHardware -** OpenStack
2. **OperatingSystem -** Ubuntu, Windows
3. **Service -** Java, Python
4. **Server -** Tomcat, Apache, nginx
5. **Messaging -** ActiveMQ, Mosquitto, Topic

By comparing the type of every component with the entries of these lists, the components will be labeled as follows:

- **OpenStack-Liber...:** VirtualHardware
- **Ubuntu-14.04-V...:** OperatingSystem
- **Python_3:** Service
- **Mosquitto_3.1:** Messaging
- **Humidity, TempTopic, PressureTopic:** Messaging

Only the Openweathermap's type can't be found in any of these lists. That's the reason why it will be labeled as Application. Since every component has an individual role specification now, the role labeling process has been finished.

6.2 Normalization

In order to apply the Node-based Availability pattern, the graph must be normalized first because the result will be wrong otherwise. During the normalization process, some *HostedOn* edges will be removed to resolve the Triangle Problem. In the first step, all nodes representing an execution environment need to be identified by matching every node's type with a keyword list. The used list is shown below:

ExecutionEnvironments - Java, Python, Ruby

By comparing the node types with this list, it turns out that no node matches the Java or Python keyword. The only found execution environment was *Python_3*. In the second step, all *DependsOn* sources targeting the execution environment need to be identified. If this source node and the environment node are hosted on the same operating system, the structure needs to be normalized. In the example, *Openweathermap* is the only component that depends on *Python_3*. Further, both are hosted on the same OS. That's the reason why the *HostedOn* relationship between *Openweathermap* and the common *Ubuntu-14.04*

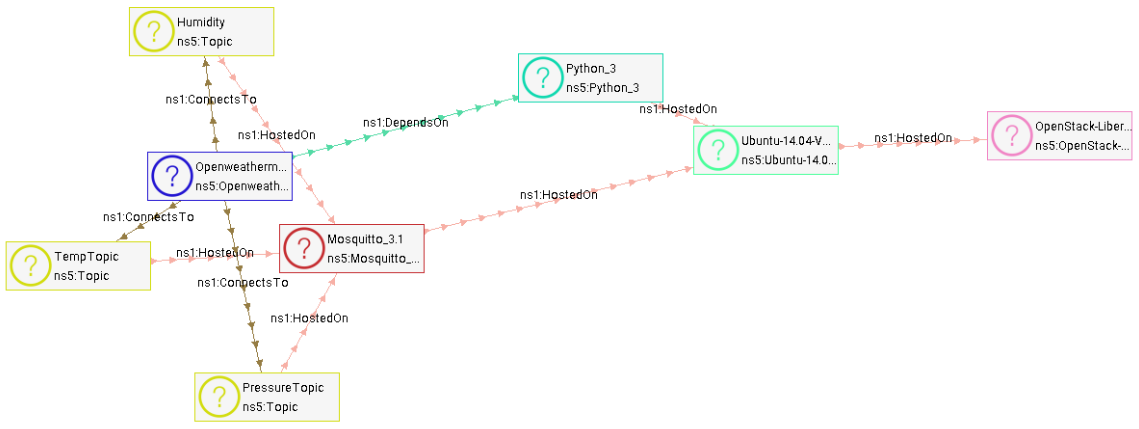

Figure 6.2: Openweathermap TOSCA Topology Normalized

node will be removed. However, no information is lost because the *DependsOn* relation still implies that *Openweathermap* is hosted on the same OS as its execution environment. Figure 6.2, which shows the result of the normalization process, depicts exactly the same topology except for one missing *HostedOn* edge between the *Openweathermap* and the *Ubuntu-14.04* node.

6.3 Pattern Inclusion

After normalization and role-labeling, different patterns can be applied to the topology. Every pattern is unique and must therefore be implemented in their own way. Exemplary, all execution steps of the previously described Watchdog, Node-based and Environment-based Availability algorithms will be explained in this section.

6.3.1 Watchdog Pattern

Algorithm 4.1 will be used to apply the Watchdog pattern to the labeled topology. In the first step, a new node must be created for the Watchdog component. This node will then be connected to every node whose role was identified as Application with a *ConnectsTo* relationship. The only node that was labeled as Application is Openweathermap. Therefore, the Watchdog node will only be connected to this one. The corresponding excerpt of the topology is shown in figure 6.3. Thereafter, the pattern has been successfully applied.

6.3.2 Node-based Availability

In order to apply Node-based Availability to the example topology using algorithm 4.3, the graph must be role-labeled and normalized first. Both algorithms have been explained before and the result is shown in figure 6.2. In the next step, all application nodes need

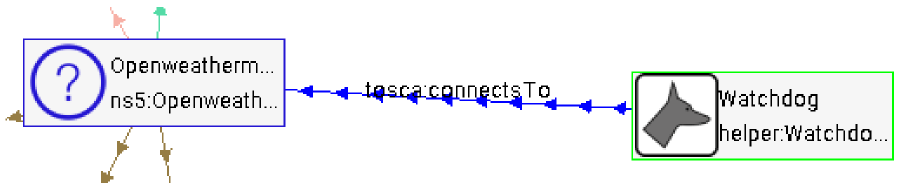

Figure 6.3: Applied Watchdog

to be filtered out. In our example, the only node that remains is the *Openweathermap*. The corresponding subgraph is then created by adding the node itself as well as all other non-application nodes and their relations to a new graph using a breadth first search. In our example, the subgraph matches the normalized input graph. The created subgraph will be forked in the next step. During the forking process, every node that has at least two predecessors and whose role is neither Application nor VirtualHardware or Messaging will be cloned. The only component that matches this description is the *Ubuntu-14.04* node. Therefore, this node will be duplicated. The outgoing edges will remain the same for every clone but each will have only one incoming edge. As a result, the *Python_3* as well as the *Mosquitto_3.1* component have their own OS node. The result is shown in figure 6.4. It is important to mention that the detection framework doesn't recognize the Node-based Availability properly. Even after applying, the probability is not set to *Detected*. This is, however, a fault of the pattern detection framework. As soon as the topology graph doesn't match the Node-based Availability graph for 100%, the pattern won't be recognized correctly. Per definition, the Node-based Availability has been properly applied, though.

6.3.3 Environment-based Availability

Environment-based Availability is already existent in the example graph. Therefore, the topology depicted in figure 6.4 will be used as input. By applying the Environment-based Availability, the result of the Node-based Availability should be reversed. After role-labeling, all node pairs whose nodes have the same type and role are filtered out. In our example, the only two nodes satisfying this dependency are both *Ubunt-14.04* nodes. Both of them will now be merged. This can be achieved by redirecting all incoming edges of node one to the second node. Thereafter, node one can be deleted. As a result, *Mosquitto_3.1* and *Python_3* will be hosted on the same component again. The result is the normalized base graph, shown in figure 6.2.

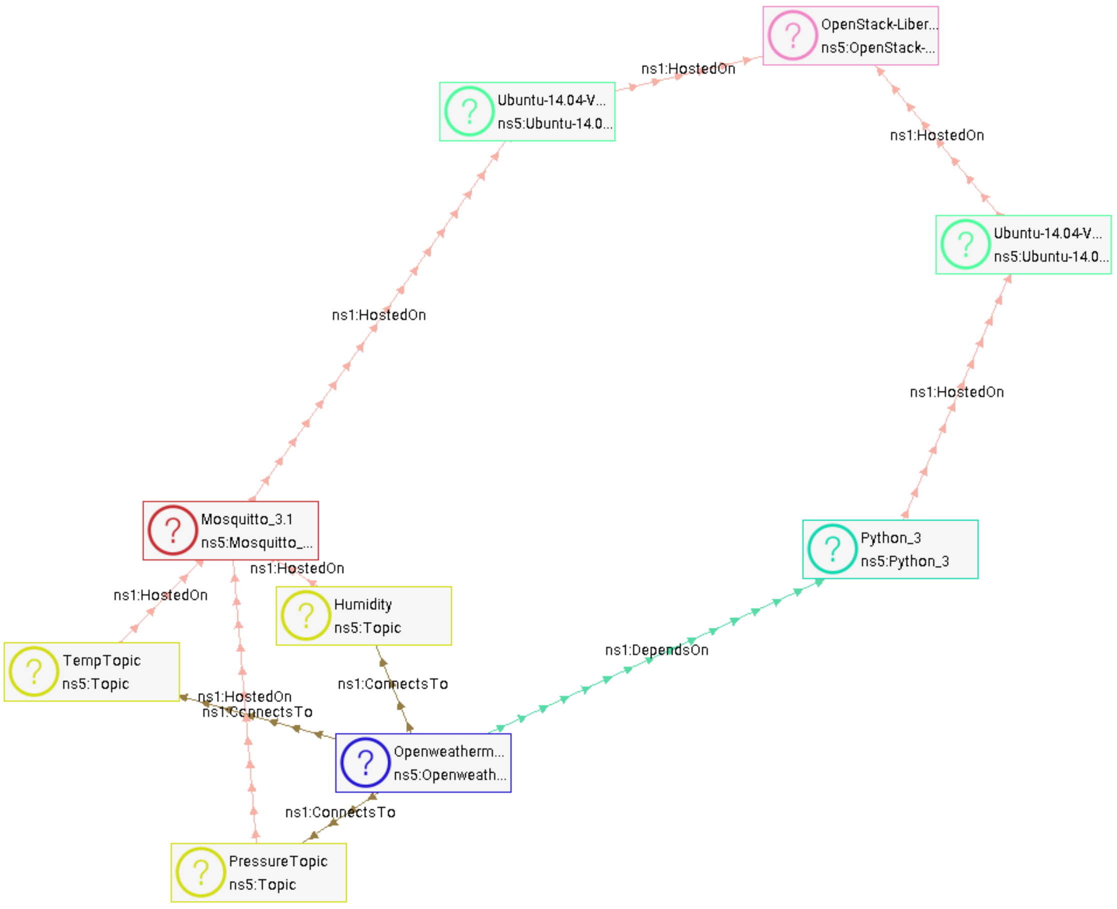

Figure 6.4: Applied Node-based Availability

7 Conclusion and Outlook

Big software systems need to be planned and designed like a new, big building. Thus, a software architect is part of every big software project. His job is the design of an application's architecture that satisfies functional-, as well as non functional requirements (NFRs). Because of similar problems during the design process of different applications, specific patterns for proven solutions of recurring problems have been developed. The framework described in this bachelor's thesis can be used to evaluate the influence of given patterns on different NFRs in an understandable and comprehendable way. The focus is on an excerpt of the NFRs defined in ISO 25010. By using the NFR-Framework, an influence analysis of different patterns can be done. Further, the detection of existing patterns can be used to weigh up the influence of different patterns on a whole system. The second part of this framework deals with the automation of including patterns in existing architectures, based on TOSCA related examples. Even if every pattern has a unique structure and must be implemented in its own way, common rules for pattern inclusion can be defined as, for example, role-labeling and normalization. The described framework offers no specific instructions for the application of different CC patterns. Rather, it can be seen as a general purpose technique for a comprehensible influence analysis of given patterns on a system's NFRs combined with an approach for automated pattern inclusion. Therefore, it's no replacement for software architects but a helpful tool for the analysis and optimization of existing architectures. For a better understanding, the framework is explained on three basic CC patterns: Watchdog, Node-based Availability and Environment-based Availability. All of them have been analyzed using the NFR-Framework and have been applied on an example topology for validation. PIA, which is enclosed with this thesis, is a graphical user interface for the application's NFR evaluation and pattern inclusion. It can be used to understand how the application of these three patterns will impact existing TOSCA architectures.

A common and generic framework leads to replicability and comprehensibility of the pattern analysis process. This is especially valuable if a software architect has to justify its decisions. In addition to that, the automated pattern inclusion comes with time savings and less error-proneness, which directly leads to a more productive design process.

The current framework offers a solution for the analysis and inclusion of patterns, based on software architectures. Because the individual steps of this framework are generic, they may be applied to more fine-grained patterns and architectures on a class or package level. For future work, additional algorithms for all kind of patterns can be developed as well as empirical data gathered. Further, the enclosing prototype of the application PIA can be extended. Currently, it is only possible to load existing TOSCA archives as well as evaluating and implementing the three above named patterns. In order to be helpful, more patterns need to be implemented and the export of modified topologies should be possible.

A Appendix

A.1 Environment-based Availability SIG

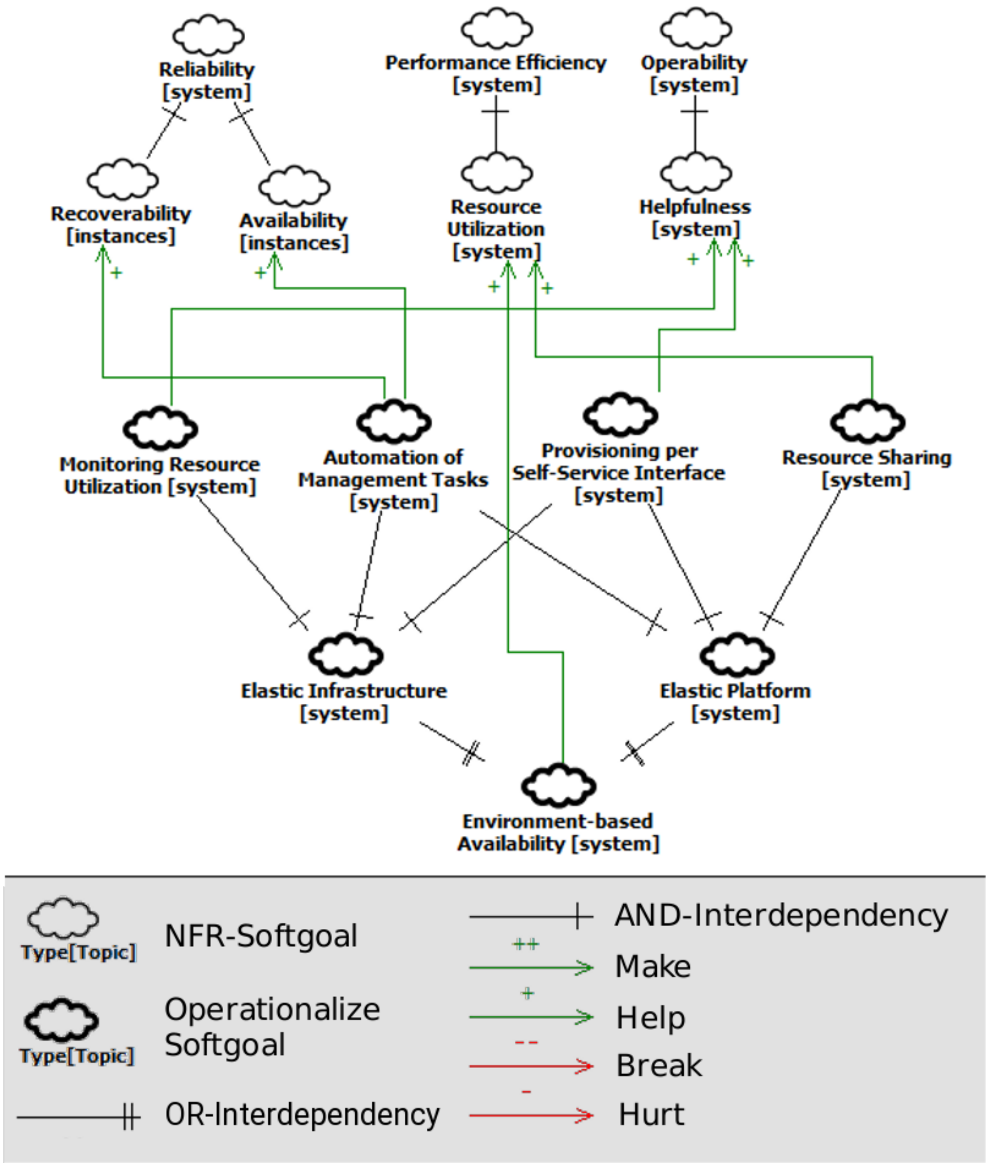
Reliability [system]
Performance Efficiency [system]
Operability [system]
Recoverability [instances]
Availability [instances]
Resource Utilization [system]
Helpfulness [system]
Monitoring Resource Utilization [system]
Automation of Management Tasks [system]
Provisioning per Self-Service Interface [system]
Resource Sharing [system]
Elastic Infrastructure [system]
Elastic Platform [system]
Environment-based Availability [system]
Type[Topic]
NFR-Softgoal
Type[Topic]
Operationalize Softgoal
OR-Interdependency
AND-Interdependency
++
Make
+
Help
--
Break
-
Hurt

Figure A.1: SIG of the Environment-based Availability

A.2 Node-based Availability SIG

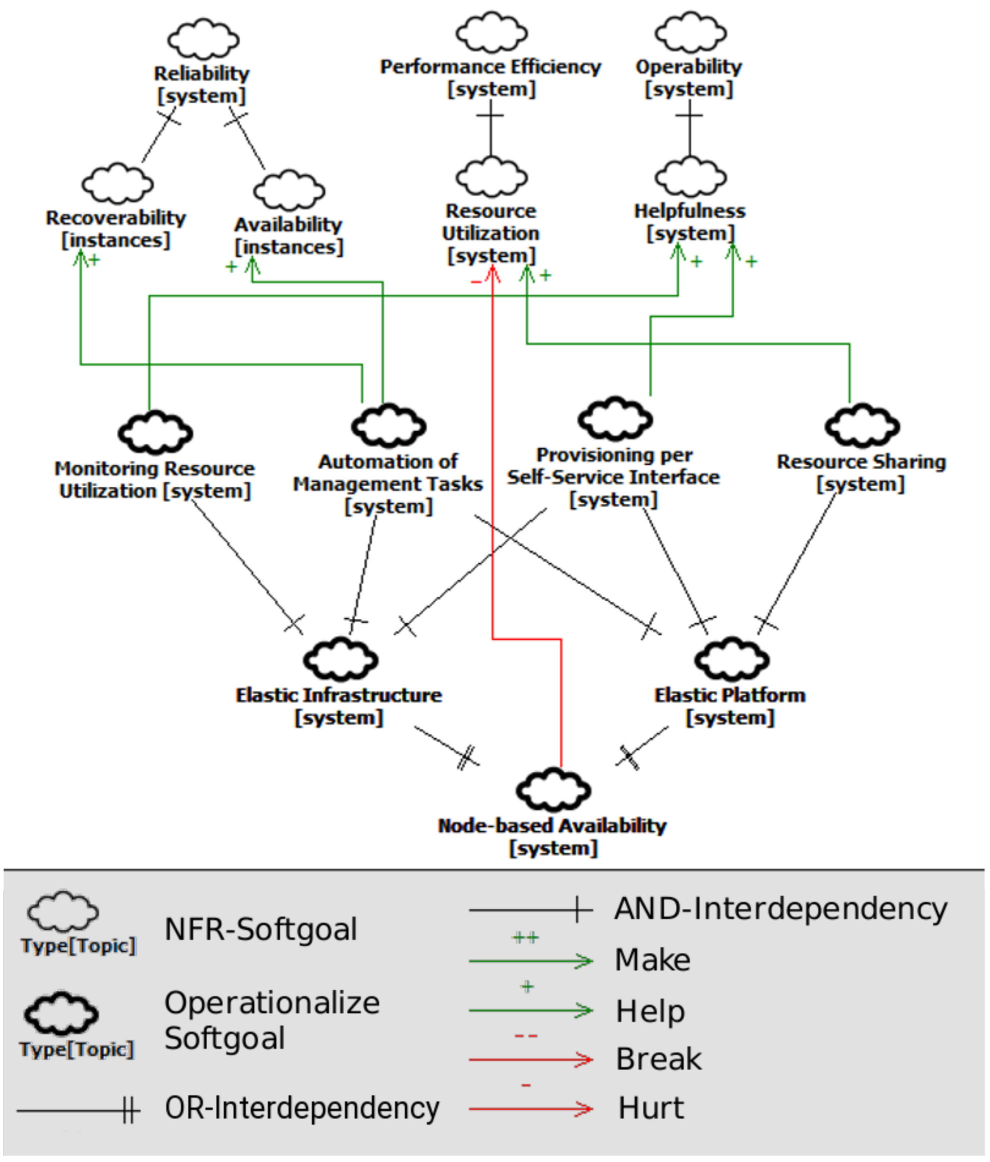
Reliability [system]
Performance Efficiency [system]
Operability [system]
Recoverability [instances]
Availability [instances]
Resource Utilization [system]
Helpfulness [system]
Monitoring Resource Utilization [system]
Automation of Management Tasks [system]
Provisioning per Self-Service Interface [system]
Resource Sharing [system]
Elastic Infrastructure [system]
Elastic Platform [system]
Node-based Availability [system]
Type[Topic]
NFR-Softgoal
Operationalize Softgoal
OR-Interdependency
AND-Interdependency
Make
Help
Break
Hurt

Figure A.2: SIG of the Node-based Availability

Bibliography

[15] *OMG Unified Modeling Language TM (OMG UML)*. 2015 (cit. on pp. 28, 29).

[17] *Oracle Orchestration Cloud Service Data Sheet*. Oracle, 2017 (cit. on p. 27).

[BAA12] L. Bautista, A. Abran, A. April. "Design of a performance measurement framework for cloud computing." In: *Journal of Software Engineering and Applications* 5.02 (2012), p. 69 (cit. on pp. 16, 17).

[BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. "OpenTOSCA–a runtime for TOSCA-based cloud applications." In: *International Conference on Service-Oriented Computing*. Springer, 2013, pp. 692–695 (cit. on pp. 21, 22).

[BBKL14] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. "Vinothek-A Self-Service Portal for TOSCA." In: *ZEUS*. 2014, pp. 69–72 (cit. on p. 22).

[BM+02] D. Bredemeyer, R. Malan, et al. "The role of the architect." In: *Resources for Software Architects* (2002) (cit. on p. 13).

[BP00] F. Bergenti, A. Poggi. "Improving UML designs using automatic design pattern detection." In: *12th International Conference on Software Engineering and Knowledge Engineering (SEKE)*. 2000, pp. 336–343 (cit. on p. 30).

[CFSV04] L. Cordella, P. Foggia, C. Sansone, M. Vento. "A (sub)graph isomorphism algorithm for matching large graphs." en. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 26.10 (Oct. 2004), pp. 1367–1372. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2004.75. URL: http://ieeexplore.ieee.org/document/1323804/ (visited on 11/02/2017) (cit. on p. 32).

[CNYM12] L. Chung, B. A. Nixon, E. Yu, J. Mylopoulos. *Non-functional requirements in software engineering*. Vol. 5. Springer Science & Business Media, 2012 (cit. on p. 19).

[CSB+05] J. Cleland-Huang, R. Settimi, O. BenKhadra, E. Berezhanskaya, S. Christina. "Goal-centric traceability for managing non-functional requirements." In: *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 362–371 (cit. on p. 13).

[DBB14] M. Drozdová, B. Bucko, I. Brídová. "Architectures of the next eLearning systems upgrade." In: *2014 IEEE 12th IEEE International Conference on Emerging eLearning Technologies and Applications (ICETA)*. Dec. 2014, pp. 109–114. DOI: 10.1109/ICETA.2014.7107556 (cit. on p. 28).

[Eid05] P. L. Eide. "Quantification and Traceability of Requirements." In: *NTNU, Norwegian University of Science and Technology* (2005) (cit. on p. 15).

[EN13] M. Elaasar, A. Neal. "Integrating modeling tools in the development lifecycle with oslc: A case study." In: *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2013, pp. 154–169 (cit. on p. 25).

[FGH06] P. H. Feiler, D. P. Gluch, J. J. Hudak. *The architecture analysis & design language (AADL): An introduction*. Tech. rep. Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2006 (cit. on p. 29).

[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*. ACM, 2011, p. 2 (cit. on p. 23).

[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 (cit. on pp. 13, 17, 35, 40–42).

[Gam95] E. Gamma, ed. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley professional computing series. Reading, Mass: Addison-Wesley, 1995. ISBN: 978-0-201-63361-0 (cit. on p. 34).

[GMW10] D. Garlan, R. Monroe, D. Wile. "Acme: An architecture description interchange language." In: *CASCON First Decade High Impact Papers*. IBM Corp., 2010, pp. 159–173. DOI: 10.1145/1925805.1925814 (cit. on p. 28).

[GY01] D. Gross, E. Yu. "From non-functional requirements to design through patterns." In: *Requirements Engineering* 6.1 (2001), pp. 18–36 (cit. on p. 13).

[HC07] S. Henninger, V. Corrêa. "Software pattern communities: Current practices and challenges." In: *Proceedings of the 14th Conference on Pattern Languages of Programs*. ACM, 2007, p. 14 (cit. on pp. 15, 23, 38).

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. "Winery–a modeling tool for TOSCA-based cloud applications." In: *International Conference on Service-Oriented Computing*. Springer, 2013, pp. 700–704 (cit. on p. 22).

[KML+14] G. Katsaros, M. Menzel, A. Lenk, J. R. Revelant, R. Skipp, J. Eberhardt. "Cloud application portability with tosca, chef and openstack." In: *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. IEEE, 2014, pp. 295–302 (cit. on p. 27).

[LOZ10] P. Lew, L. Olsina, L. Zhang. "Quality, quality in use, actual usability and user experience as key drivers for web application evaluation." In: *Web Engineering* (2010), pp. 218–232 (cit. on p. 15).

[MG+11] P. Mell, T. Grance, et al. "The NIST definition of cloud computing." In: (2011) (cit. on pp. 13, 16).

[Ora] Oracle. *About the JFC and Swing*. URL: https://docs.oracle.com/javase/tutorial/uiswing/start/about.html (cit. on p. 53).

[PC02] F. R. S. Paim, J. Castro. "Enhancing Data Warehouse Design with the NFR Framework." In: *WER* 2 (2002), pp. 40–57 (cit. on p. 19).

[RBDP15] R. Ranjan, B. Benatallah, S. Dustdar, M. P. Papazoglou. “Cloud resource orchestration programming: overview, issues, and directions.” In: *IEEE Internet Computing* 19.5 (2015), pp. 46–56 (cit. on p. 26).

[RKK07] A.-E. Rugina, K. Kanoun, M. Kaâniche. “An architecture-based dependability modeling framework using AADL.” In: *arXiv preprint arXiv:0704.0865* (2007) (cit. on p. 29).

[SBB+16] J. Soldani, T. Binz, U. Breitenbücher, F. Leymann, A. Brogi. “ToscaMart: a method for adapting and reusing cloud applications.” In: *Journal of Systems and Software* 113 (2016), pp. 395–406 (cit. on p. 25).

[SSC03] G. M. C. de Sousa, I. G. da Silva, J. B. de Castro. “Adapting the NFR framework to aspect-oriented requirements engineering.” In: *Proceeding of XVII Brazilian Symposium on Software Engineering*. 2003, pp. 83–98 (cit. on pp. 18, 19).

[Sta13] O. Standard. *Topology and orchestration specification for cloud applications version 1.0*. Tech. rep., OASIS Standard (November 2013). url:${$http://docs. oasis-open. org/tosca/TOSCA/v1. 0/os/TOSCA-v1. 0-os. html$}$, 2013 (cit. on pp. 13, 19).

[Wat08] A. Watson. “Visual Modelling: past, present and future.” In: *White paper UML Resource Page* (2008) (cit. on p. 28).

[Woh17] M. Wohlfarth. “Design Pattern Detection Framework for TOSCA-Topologies.” Bachelor Thesis. University of Stuttgart, 2017 (cit. on pp. 23, 30, 31, 42, 44, 53, 55).

[Wol94] P. Wolfgang. “Design patterns for object-oriented software development.” In: *Reading Mass* (1994) (cit. on p. 15).

All links were last followed on November 24, 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