Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Versioning of Applications Modeled in TOSCA

Lukas Harzenetter

Course of Study: Softwaretechnik

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

Supervisor: Dr. rer. nat. Uwe Breitenbücher
Dr. rer. nat. Oliver Kopp

Commenced: October 4, 2017

Completed: April 4, 2018

Abstract

Cloud applications are subject to frequent changes to enable fast feedback on user requests about problems and features, or to provide security updates. In both cases, the topology of a cloud application may change. If, for instance, these changes are only applied to one software component and are not propagated into the service's topology, the whole application stack may not start anymore. Thus it is crucial to maintain a runnable version of the service's topology to ensure the availability of the cloud application.

The Topology Orchestration Specification for Cloud Applications (TOSCA) is an emerging standard to describe and manage cloud applications. Since it is not able of handling multiple versions of an artifact, this thesis presents an approach to version TOSCA definitions. It is achieved by appending a version identifier at the end of each root element's id. This way it is ensured that the approach conforms to the TOSCA specification. Further, it ensures the validity of all artifacts by preventing users from performing changes to committed versions.

In addition to the versioning approach, a second approach for determining differences between two versions is presented. Besides a generic difference representation, it provides two types of visualizing changes: 1.) a textual visualization for all elements and 2.) a graphical visualization for Topology Templates. Both approaches are implemented prototypically in the context of Eclipse Winery, a web-based modeling tool for TOSCA elements.

Contents

1 Introduction **17**
1.1 Motivation 17
1.2 Structure 18
1.3 Running Example 18

2 Background, Fundamentals, and Related Work **21**
2.1 Topology Orchestration Specification for Cloud Applications (TOSCA) 21
2.2 Eclipse Winery 25
2.3 OpenTOSCA Ecosystem 26
2.4 Versioning 27
2.5 Difference Calculation 29
2.6 Package and Version Naming 33
2.7 Regular Expressions 34

3 An Approach to Version TOSCA Definitions **37**
3.1 Requirements 38
3.2 Version Identification 38
3.3 Support for Existing TOSCA Definitions 43
3.4 TOSCA Definition Version States 44
3.5 Applying Changes to TOSCA Definitions 47
3.6 Freezing and Releasing a TOSCA Definition 55

4 An Approach for Differences between TOSCA Definitions **57**
4.1 Requirements 57
4.2 Determining Differences 58
4.3 Topology Difference Visualization 61
4.4 Textual Difference Visualization 64

5 Prototype and Validation **67**
5.1 Extension to Eclipse Winery 67
5.2 Validation 74

6 Summary and Outlook **81**

Bibliography **83**

List of Figures

1.1 Topology of the running example 19

2.1 Structure of a Cloud Service Archive (CSAR) 23
2.2 Dependencies between the TOSCA definitions 24
2.3 Dependencies of Service Templates 25
2.4 Current components of Eclipse Winery 26
2.5 Schematic view of the OpenTOSCA Ecosystem 27
2.6 Example model to visualize differences 29
2.7 Meta model for the models shown in Figure 2.6 32
2.8 Difference Meta-Model (MMD) according to the meta model shown in Figure 2.7 32
2.9 Difference model of the models shown in Figure 2.6 33

3.1 Extended example topology including version identifiers. 37
3.2 State diagram defining TOSCA definition version state transitions 45
3.3 Activity diagram to decide the initial TOSCA definition version state 46
3.4 Steps required to update a definition 48
3.5 Steps to update a TOSCA definition's id 49
3.6 Invalid and valid version advancement 51
3.7 UML activity diagram to freeze a TOSCA definition 55
3.8 UML activity diagram to release a TOSCA definition 56

4.1 Second version of the example topology. 58
4.2 Visualization of changes in a topology template 63

5.1 Simplified class diagram showing the functionality of the new `VersionUtils`. . . 69
5.2 Overview over all Node Types defined in the current repository 70
5.3 Overview of multiple versions of a Node Type 71
5.4 View of a single TOSCA definition 72
5.5 Dialog to add a new version 73
5.6 View of a editable definition version 73
5.7 DRC topology in the 2011 version. 77
5.8 DRC topology in the 2018 version 77
5.9 Difference visualization between two DRC topologies. Left part. 78
5.10 Difference visualization between two DRC topologies. Right part. 79

List of Tables

3.1 Alternatives for saving the version 40
3.2 Valid TOSCA definition ids in ascending order including their resulting versions . 42
3.3 Valid TOSCA definition ids which incorrectly define versions 43
3.4 Alternatives to support existing TOSCA definitions 44
3.5 TOSCA definitions and their derived version states 47
3.6 Alternatives to update referencing TOSCA definitions 50

4.1 Primary keys to identify elements inside TOSCA definitions. 59
4.2 Changes in the topology between the Service Templates `WebshopService_-w1` and `WebshopService_-w2-wip1`. 62

5.1 Requirement validation 75
5.2 Component versions of the DRC's topology. 76

List of Listings

2.1 Example for versions in BPEL according to Juric et al. [JSR09] 28
2.2 Textual difference visualization of the example model 29
4.1 Textual difference visualization example 65

List of Algorithms

4.1 Difference algorithm 61

List of Abbreviations

API Application Programming Interface.

BPEL Buissness Process Execution Language.

BPMN Bussiness Process Model and Notation.

CSAR Cloud Service Archive.

DA Deployment Artifact.

ECL Epsilon Comparison Language.

HTML Hypertext Markup Language.

IA Implementation Artifact.

ID Identifier.

MMD Difference Meta-Model.

SLA Service Level Agreement.

TDD Test Driven Development.

TOSCA Topology Orchestration Specification for Cloud Applications.

UI User Interface.

UML Unified Modeling Language.

URI Uniform Resource Identifier.

UUID universally unique identifier.

VCS Version Control System.

VM Virtual Machine.

WAR Web Application Resource.

WIP work in progress.

XML Extensible Markup Language.

YAML YAML Ain't Markup Language.

1 Introduction

In the first chapter, the motivation and goal for this thesis are described in Section 1.1, followed by a short overview outlining the thesis' structure in Section 1.2. Closing this chapter, a running example is introduced in Section 1.3 which is used to explain the concepts proposed in this thesis.

1.1 Motivation

In todays life, cloud computing has become a necessity for most people around the world, even if they do not recognize it. Services and applications which, for example, provide news, communication (such as email, instant messages or voice-over-IP) or any other service over the Internet are usually used every day. Most are cloud applications themselves or are backed by one providing endpoints for many applications to send and receive the requested data [Ley09].

All cloud applications need to be managed and maintained by their providers. To ensure frequent updates and security patches, the deployment and management of cloud applications should be automated to avoid error prone, manual tasks. The Topology Orchestration Specification for Cloud Applications (TOSCA) [OAS13b] is an emerging standard providing these features. It further provides a way of describing the underlaying infrastructure resources (such as Virtual Machines (VMs) hosted at infrastructure service providers), components (e.g., required middleware like a web server), and the structure of the application itself. Topology Orchestration Specification for Cloud Applications (TOSCA) is further described in Section 2.1.

To automatically deploy and run an application modeled in TOSCA, all required files, scripts and a Service Template can be packaged in a self-contained archive called Cloud Service Archive (CSAR). Therefore, it is important to ensure 1.) the reproducibility and 2.) the maintainability of the service. In most cases, the components used to model the application stack in a Service Template are bound to specific versions. Only if a component is completely forward and backward compatible, its version does not matter. However, during the applications life cycle the stack has to be changed regularly due to maintenance issues for example, because new versions of employed components are available which should be implemented for security reasons. Similarly, it must be ensured to reproduce deployments of the same service to handle request peeks for instance.

Additionally, *ToscaMart* [SBB+16] is an approach which tries to avoid "reinventing the wheel multiple times when similar solutions are [...] created by different developers for different applications" [SBB+16]. In case fragments or whole existing topologies are reused in other cloud applications, it is crucial to ensure that the reused topologies are modeled correctly.

As an example, consider a research approach implementing a proof-of-concept prototype as part of an Executable Paper [KE11]. Assume it defines a Java web application in version 2.3 running on a Tomcat 8 web-server which both depend on Java 8. Since the web application was currently

updated to use Java 9 features, the service is not starting anymore because the Java node in the topology was not updated to version 9. In the meantime, another developer created a new CSAR and deployed it on a production server resulting in an unavailable service. If fragments of this topology are additionally reused by another cloud application, both services are not available anymore. This can escalate quickly in case that multiple other topologies rely on the first one.

Thus, to prevent similar scenarios, an approach to version TOSCA artifacts is needed to ensure runnable Service Templates and thereafter runnable CSARs. Further, it must be ensured that stable versions can be distinguished from unstable ones to clearly indicate a service's state and avoid confusions as outlined above.

1.2 Structure

This thesis first introduces the necessary background information and fundamentals including related work in Chapter 2. Followed by an approach to version TOSCA definitions, solving the issues outlined in the motivation, in Chapter 3 starting at page 37, and a method for calculating differences between two given versions of a TOSCA element in Chapter 4 starting at page 57. The proof-of-concept implementation in the context of Eclipse Winery [KBBL13] for both approaches is described in Chapter 5 starting at page 67. Finally, the thesis concludes with a summary and outlook given in Chapter 6 starting at page 81.

Throughout this thesis, a single user scenario is considered, where only one user edits TOSCA definitions at a time. This avoids the complex topic of conflict detection and handling. However, it should be considered in future work.

1.3 Running Example

The topology shown in Figure 1.1 will act as running example throughout this thesis. It is decapitated according to the *Vino4TOSCA* visual notation introduced by Breitenbücher et al. [BBK+12]. The example topology consists of a Webshop written in Java which is hosted on a Tomcat 8[1] web server. Tomcat itself depends on a Java 8[2] runtime and is hosted on an Ubuntu 16.04[3] operating system. The whole stack is deployed on an Amazon EC2[4] virtual machine.

At the bottom of the Ubuntu, Tomcat, and Amazon EC2 nodes in the topology properties are defined to further specify a single node. For example, the Tomcat web server is listening for requests on port 80, Ubuntu as the operating system allocates 16GB of RAM, and the Amazon EC2 virtual machine is reachable via the IP address 21.20.1.2.

[1] https://tomcat.apache.org
[2] https://java.com
[3] https://ubuntu.com
[4] https://aws.amazon.com/ec2

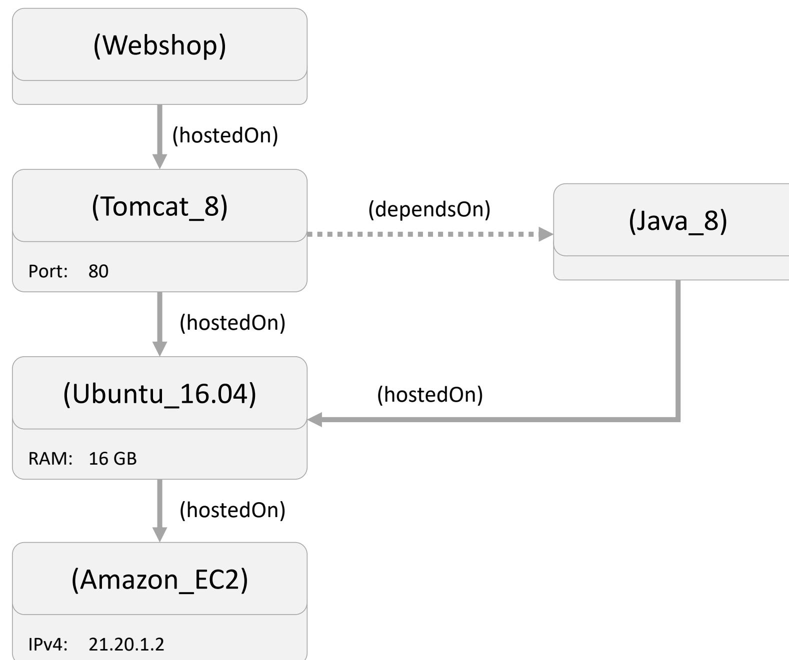

Figure 1.1: Topology of the running example

2 Background, Fundamentals, and Related Work

This chapter introduces necessary information required to understand the approach and implementation. It introduces the OASIS standard TOSCA (Section 2.1), the open source modeling tool Eclipse Winery (Section 2.2) including the TOSCA implementation OpenTOSCA (Section 2.3), as well as background information and related work about versioning (Section 2.4), difference calculation (Section 2.5), version naming (Section 2.6), and regular expressions (Section 2.7).

2.1 Topology Orchestration Specification for Cloud Applications (TOSCA)

The Topology Orchestration Specification for Cloud Applications (TOSCA) is an emerging OASIS standard which was published in 2013 [OAS13b]. It provides the ability to describe the topology of cloud applications in a standardized and provider-independent way [BBKL14a]. All required elements are modeled in TOSCA definition documents serialized in XML files. Throughout this thesis, "TOSCA definition" is used as an umbrella to collectively refer to all elements defined in TOSCA. Accordingly, "TOSCA definition type" refers to the type of the TOSCA definition (e.g., Service Template, or Node Type). In addition, a new "TOSCA Simple Profile" was introduced in 2017 [OAS17]. It is defined in YAML rather than in XML files. However, the original definition specified in XML is still valid [OAS17] and is further explained in the following since this thesis builds atop of TOSCA 1.0. More information about TOSCA 1.0 and its elements are available in the TOSCA Specification [OAS13b] and the TOSCA Primer [OAS13a].

2.1.1 TOSCA Concepts

To describe a service, TOSCA introduces a *Service Template* containing a *Topology Template* which defines the actual application stack. The topology model consists of *Node Templates* and *Relationship Templates* which both are instances of *Node Types* and *Relationship Types* respectively. An example Topology Template following the running example is illustrated at the left side of Figure 2.1. The running example is extended to introduce the core concepts of TOSCA. It is described in Section 1.3.

To deploy and run the webshop topology, all five components described in the running example (Webshop, Tomcat_8, Java_8, Ubuntu_16.04, and Amazon_EC2), must be modeled in TOSCA. This is done by creating Node Types for each of these components. Therefore, the Node Types `Webshop`, `Tomcat_8`, `Java_8`, `Ubuntu_16.04`, and `Amazon_EC2` are created. Similarly the Relationship Types `hostedOn`, and `dependsOn` are constructed because each node is hosted on another (e.g., Tomcat is hosted on the Ubuntu operating system), or depends on other nodes (e.g., Tomcat depends on Java). Node Types, such as the `Amazon_EC2` for instance, and Relationship Types may provide

management interfaces which can be used to define management operations for controlling the life cycle or to perform arbitrary tasks on a running instance. In this case, the `Amazon_EC2` Node Type defines interfaces to start and stop the VM. These interfaces are implemented in Implementation Artifacts (IAs) which can be instantiated as Java applications, simple shell scripts, or any other program performing the required operation.

Implementation Artifacts are defined in separate TOSCA definition types: *Node Type Implementations* and *Relationship Type Implementations*. Both are required to materialize the interfaces of the respective type. Additionally, in the case of Node Type Implementations, it is also possible to define Deployment Artifacts (DAs) representing the actual implementation of a Node Type. For example, the `Tomcat-8-Impl` Node Type Implementation declares a DA containing the Tomcat installer.

It is possible to override or extend the DAs defined in Node Type Implementations. Figure 2.1 shows a Deployment Artifact attached to the webshop Node Template overriding the default DA declared in the corresponding `Webshop-Impl` Node Type Implementation. In the example, the DA shown in Figure 2.1 contains the WAR-file which will be deployed on the Tomcat server.

Implementation Artifacts (IAs) and Deployment Artifacts (DAs) specify all files and their properties in *Artifact Templates*. Artifact Templates are instances of *Artifact Types*. Whereas a specific Artifact Template contains actual software, an Artifact Type defines a reusable type such as the file type, for example, the Java Web Application Resource (WAR) used by the Webshop.

Node Types may require special prerequisites which must be available before they can be started. For instance, the Webshop needs a servlet container where it can be run in. A Tomcat server is capable of running Java servlets but requires a Java runtime itself. Therefore, the `Tomcat_8` Node Type can define a *Capability Definition* indicating that it is capable of running Java servlets and a *Requirement Definition* that it needs a Java runtime. Following this paradigm, the `Webshop` Node Type can define a Requirement Definition to specify the needed servlet runtime, whereas the `Java_8` Node Type declares its capability to provide a Java runtime.

Plans can be specified in the context of a Service Template describing specific processes regarding the service. They are used to define the service's creation (also called building plans), management, and termination. Plans have to be defined as workflows which is why any arbitrary process model, for example, the Buissness Process Execution Language (BPEL) [OAS07] or the Bussiness Process Model and Notation (BPMN) [OMG11], can be employed to describe these processes. To model a workflow in TOSCA, a domain-specific language called *BMPN4TOSCA* was introduced by Kopp et al. [KBBL12]. In Figure 2.1, management plans are shown at the right bottom.

In addition to the functional elements described above, TOSCA also allows the definition of non-functional properties. These are defined generically in *Policy Types* and *Policy Templates* as their instances. A common policy for example, is the definition of Service Level Agreements (SLAs) to ensure a specific quality of service for special Node Types. The *Policy4TOSCA* approach presents a way for defining and using policies in TOSCA [WWB+13]. It is also possible to employ policies during the provisioning of the modeled application [KBF+17].

Further, Node Types, Relationship Types, Requirement Types, Capability Types, Artifact Types and Policy Types can define properties. In the example, the `Tomcat_8` Node Type defines a port property to specify the port in the Node Template on which the web server should be listing to. However, the properties are not only useful for configuring the deployment but also to bind instance information like the IP-address of the provisioned `Amazon_EC2` VM.

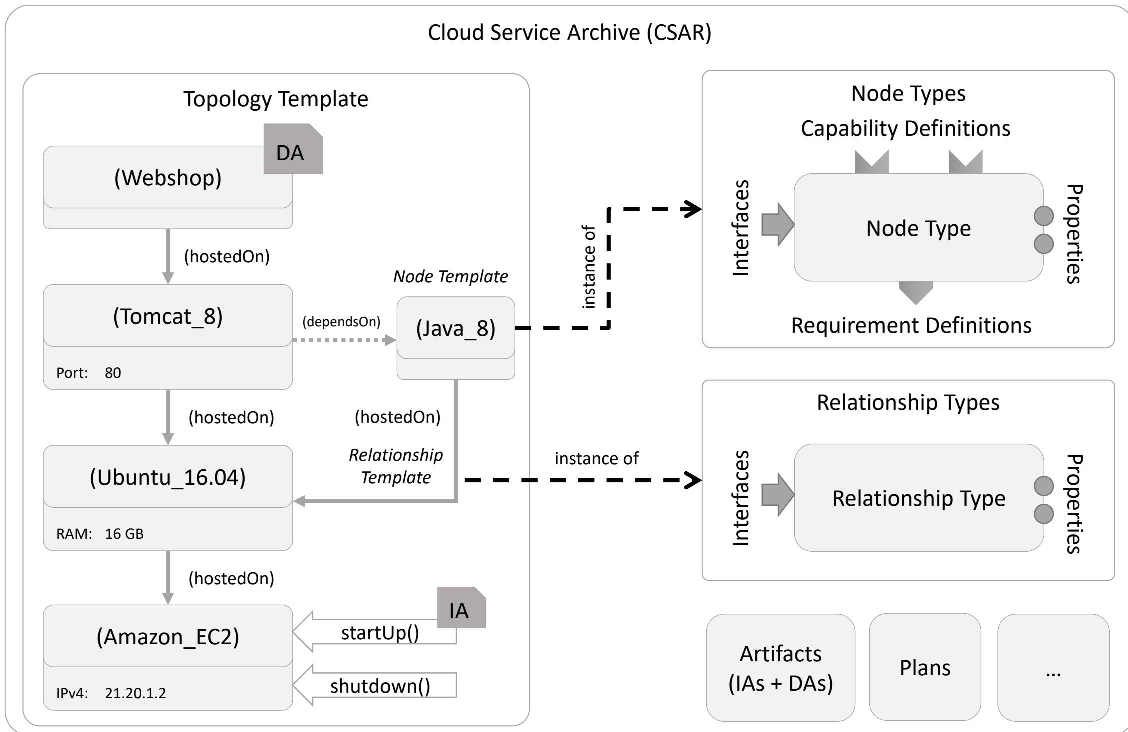

Figure 2.1: Structure of a CSAR

Summarizing all TOSCA definition types, TOSCA defines eleven root elements. These are 1.) Service Templates, 2.) Node Types, 3.) Node Type Implementations, 4.) Relationship Types, 5.) Relationship Type Implementations, 6.) Requirement Types, 7.) Capability Types, 8.) Artifact Types, 9.) Artifact Templates, 10.) Policy Types, and 11.) Policy Templates. Not all of these TOSCA definitions specify an id field. However, for the purpose of the thesis, it is assumed that each TOSCA definition has an id. If this is not the case, the id will be represented by the name attribute. But this does not change the presented concepts.

Finally, the Cloud Service Archive (CSAR) surrounding everything in Figure 2.1 bundles all TOSCA definitions and files together in one, self-contained archive. The use of CSARs has the advantage of strictly separating a service's development from its operation. A TOSCA execution engine just needs a self-contained archive to be able to perform the deployment, management and termination of the contained service.

2.1.2 Dependencies between TOSCA Definitions

To reference a TOSCA definition, for example to declare the type of a Node Template, an Extensible Markup Language (XML) specific identifier called *QNames* is used [OAS13b]. They are generated by combining the local name and the namespace of a TOSCA definition, to build an unique URI. The namespace is hereby surrounded by curly braces. For instance, the Webshop's namespace in the running example is `http://example.org/tosca/versioning/nodeTypes`, the corresponding QName is `{http://example.org/tosca/versioning/nodeTypes}Webshop`.

Figure 2.2 illustrates the dependencies among the TOSCA definitions on a high level. It is similar to an UML class diagram: arrows show a dependency to the TOSCA definition it points to, while the labels details the dependency. The diagram is intentionally not presented as a Unified Modeling Language (UML) class diagram to hide, for the scope of this thesis, unnecessary information. Further, for readability reasons the Service Template and therefore contained elements (such as Node Templates and Relationship Templates) are not shown in Figure 2.2. The relationships between Service Templates and other TOSCA definition types is more complex and is presented in more detail in Figure 2.3, including Node Templates and Relationship Templates.

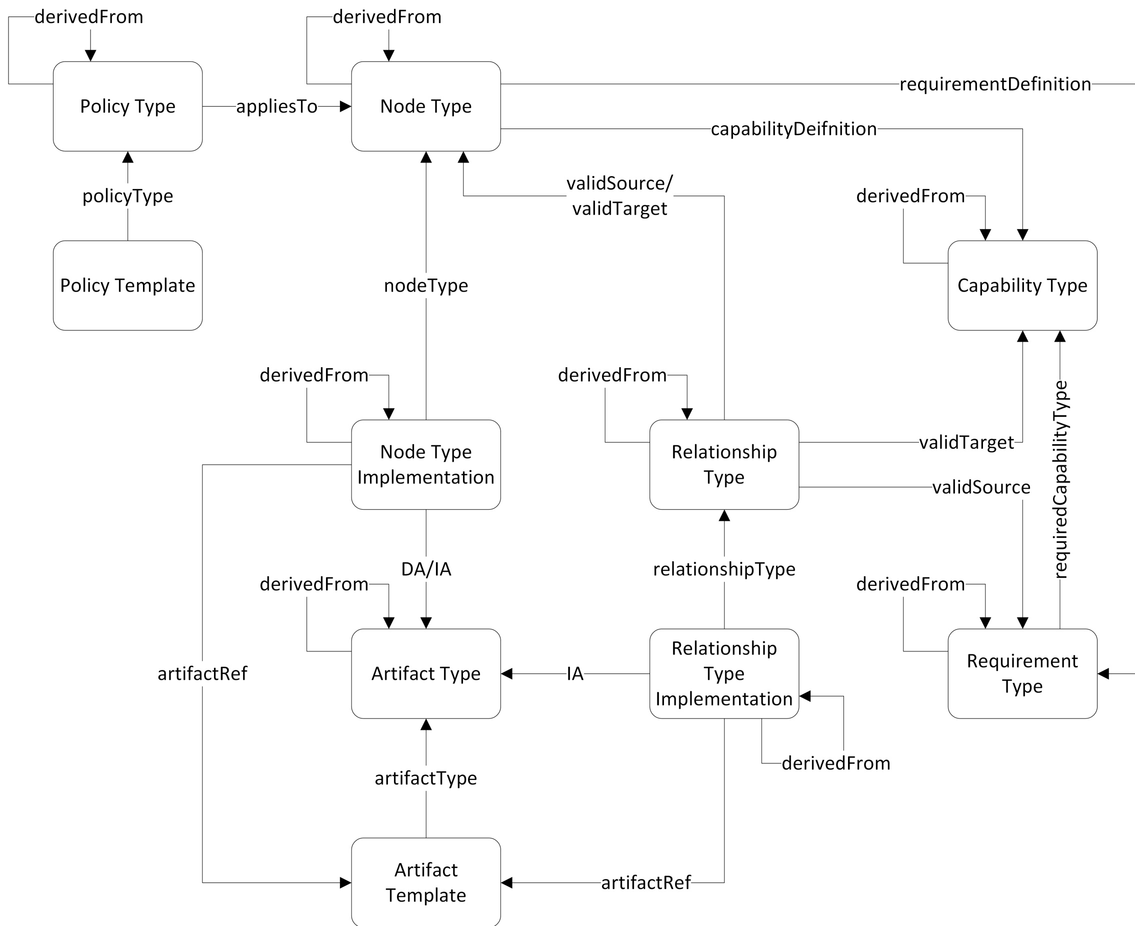

Figure 2.2: Diagram showing dependencies between the ten TOSCA definition types. The Service Template is not shown here for readability reasons. Relations are shown using arrows. The description on the relations are the elements which reference the TOSCA definitions type they point to.

In contrast to Figure 2.2, Figure 2.3 is a UML class diagram illustrating the main components of a Service Template. The reason for this is to also show the Topology Template, as well as the Node Templates, and Relationship Templates since they are the main elements of a Service Template. Elements of the Service Template are highlighted by a bold class name, whereas referenced TOSCA definition types are presented in regular style.

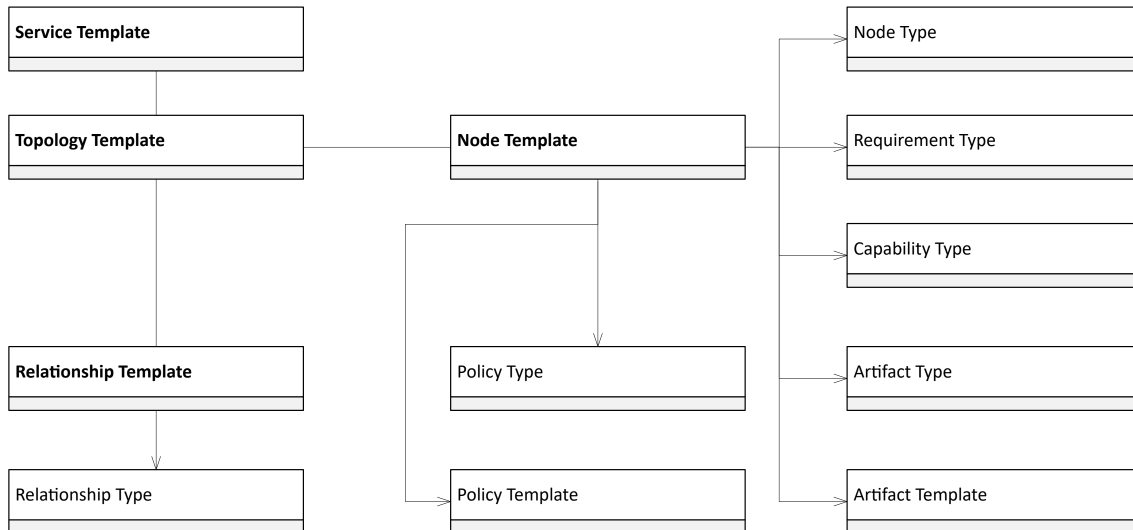

Figure 2.3: UML class diagram showing dependencies between Service Templates and other TOSCA definition types.

2.2 Eclipse Winery

Eclipse Winery is a web-based modeling tool for TOSCA-based applications [KBBL13]. It provides functionalities to create, modify and manage TOSCA definitions in an automated manner instead of writing pure XML code. The HTML5-based web front-end presents all elements of each definition separately, applies default values, and helps the user during the development by providing validity checks, auto-completion, as well as warnings and errors. It also offers the ability to do a consistency check on the whole definitions repository to ensure that the data conforms to the TOSCA standard.

Figure 2.4 shows Winery's current main components. As its database, Winery uses the local file system to save the TOSCA definitions directly in standard conform XML. The repository component implements the business logic in Java which can be accessed through a REST-API provided by JAX-RS. Standing on top, Angular applications provide front-end functionalities to the user.

In Winery, all TOSCA definitions are stored in their own XML-file inside a clearly defined directory structure in the file system. This structure depends on the TOSCA definition's type (Service Template, Node Type, Relationship Type, etc.), its namespace and id. For example, all files of a Service Template with the namespace `http://example.org/tosca/servicetemplates` and id `MyServiceTemplate` would reside in the following directories: `servicetemplates` > `http%3A%2F%2Fexample.org%2Ftosca%2Fservicetemplates` > `MyServiceTemplate`. Because namespaces contain slashes, they are encoded to not contain illegal elements.

For saving TOSCA definitions, Winery currently employs a Git-based file repository to enable primitive versioning techniques, in a way that changes made to a XML document can be committed or discarded. However, multiple versions of a definition cannot be distinguished from another because they do not get version identifiers assigned. This thesis solves this problem as the proof-of-concept implementation will extend Winery to support version identifiers for TOSCA definitions.

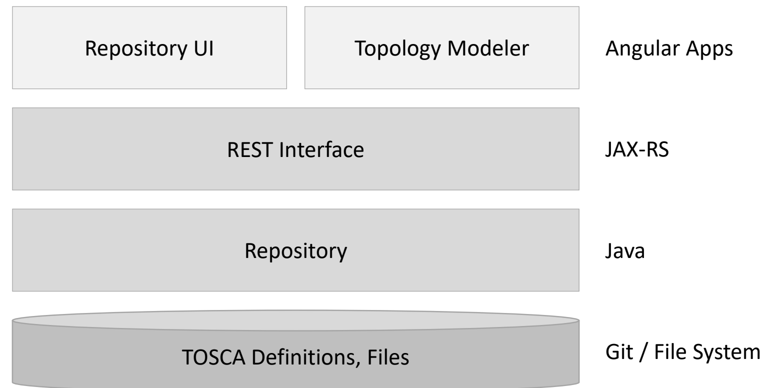

Figure 2.4: Current components of Eclipse Winery

2.3 OpenTOSCA Ecosystem

The previously presented modeling tool Winery is one part of the bigger ecosystem OpenTOSCA [BBH+13]. OpenTOSCA is an open source implementation for developing, managing, and running TOSCA-based applications. Besides Winery, there are the "Container" [BBH+13] and the "Vinothek" [BBKL14b]. Winery represents the first component in the ecosystem providing development capabilities for TOSCA definitions. While the Container is a TOSCA runtime responsible for running and managing a given service, the Vinothek enables users to trigger the provision and decommission of these applications with a single click on a button. Figure 2.5 schematically shows the dependencies between all three components in the ecosystem.

To run an application service, the OpenTOSCA Container requires a CSAR as its input. It first extracts all files and TOSCA definitions from the archive and stores them. Second, the TOSCA definitions are validated and referenced Implementation Artifacts (IAs) are deployed. Lastly, the management plans are bound to the IA's endpoints and are also deployed [BBH+13]. As a result, the service contained in the provided CSAR is available for instantiation which can be achieved using the self-service portal provided by the Vinothek [BBKL14b].

Since OpenTOSCA was introduced in 2013, it was enhanced in many ways. For example, it is capable of deploying cloud applications where the deployment is modeled declaratively and imperatively [EBF+17]. OpenTOSCA also supports the combination of declarative and imperative application provisioning, as declaratively modeled applications can be used to generate imperative provisioning plans [BBK+14].

To enable fast and frequent updates for cloud applications, the DevOps paradigm includes developers into the application's operation. This is achieved by automating the deployment process using specialized tools [WBL14]. The OpenTOSCA ecosystem is capable of transforming DevOps artifacts into Node Types and Relationship Types providing an approach for a seamless orchestration of different artifacts [WBKL16; WBL14]. Further, OpenTOSCA was successfully applied in an Industry 4.0 environment to automatically optimize production lines based on machine learning algorithms [FBK+16a].

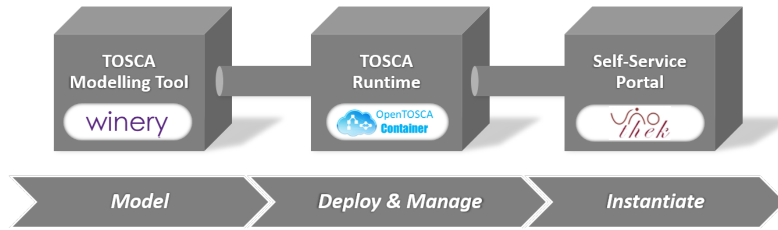

Figure 2.5: Schematic view of the OpenTOSCA Ecosystem, taken from http://opentosca.org

2.4 Versioning

This thesis intends to present a versioning approach for TOSCA definitions and uses the OpenTOSCA Ecosystem for a proof-of-concept implementation. Therefore, the following section introduces the idea of versioning (software) artifacts and gives an overview to version control.

In the field of software engineering, versioning is already a well established best-practice [Som16]. Source code can be versioned using Version Control System (VCS) tools such as Git[1] to simplify collaboration and tracking changes. Therefore, a VCS analyzes the files in a given directory and determines all changes a user made with respect to the current version. To create a version, all files have to be committed. By creating a commit, the VCS calculates all changes made to all files in the given directory and its subtree. The calculation of this list of differences, or delta, is further discussed in Section 2.5. However, different tools use different algorithms and methods to calculate their delta. A commit usually consists of an id, a message describing the changes, the delta itself and information about the author, and a timestamp. The id is, in most cases, a hash value which is calculated using the commit's content to ensure the uniqueness of every calculated identifier. Using this concept, a user is able to review all changes made over time to all files and can easily checkout different versions of each checked-in file.

As an example, Git calculates the deltas based on changes in a line of text files. If a line does not match the previous version of the line, even if only one single whitespace was changed, the whole line is considered different. Because of this, conventional VCSs are therefore not suited for complex model versioning. Changing the order of attributes in a model results in huge differences, even if there are no further changes. Therefore, model versioning, as for example presented by Cicchetti et al. [CDP07], Altmanninger et al. [AKK+08] and Könemann et al. [KKU09], try to solve this problem by calculating the differences between models on the model itself instead of their textual representation.

Despite that, versioning of XML documents can be achieved in different ways. One approach was suggested by Ogbuji [Ogb02]. Here, versions are specified by the namespace set in the `xmlns` attribute of an XML element. The namespace exactly defines the version of the XML vocabulary, either with a specific date or version identifier in it. `http://example.org/myXmlSchema/1.3` is an example namespace specifying a XML vocabulary in version 1.3.

[1] https://git-scm.com/

Another approach for introducing versioning in XML, and in particular in BPEL documents, was published by Juric et al. [JSR09]. The Buissness Process Execution Language (BPEL) is a language to describe business processes in a standardized way [LLN11]. Processes in BPEL are defined in XML. To create different versions of a process, Juric et al. [JSR09] developed an extension to the BPEL specification to declare version specific elements. Therefore, the available versions have to be specified atop of a process definition shown in Listing 2.1 from line 5 to 18. Then, inside the definition of a process, version specific parts are wrapped by the proposed version extension and are only executed, if the corresponding version is requested. However, because this extension interferes with the usual BPEL vocabulary, since elements may not appear in the expected places in the document, this approach does not conform to the BPEL standard [KGK+11].

Listing 2.1 Example for versions in BPEL according to Juric et al. [JSR09]

```
1  <process name="myProcess" targetNamespace="http://www.example.org/myProcess/"
2          xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/abstract"
3          xmlns:bpelx="http://www.uni-mb.si/bpel/versioning/">
4
5          <bpelx:versions mode="processLevel">
6                  <bpelx:version vid="1.0">
7                          <bpelx:initialVersion/>
8                          <bpelx:deprecated/>
9                          <bpelx:validUntil>2017-06-30T00:00:00Z</bpelx:validUntil>
10                         <bpelx:versionInfo>Short description of version 1.0.</bpelx:versionInfo>
11                 </bpelx:version>
12                 <bpelx:version vid="1.1" type="backward-compatible" intent="variant">
13                         <bpelx:previousVersion vid="1.0"/>
14                         <bpelx:validFrom>2017-05-01T00:00:00Z</bpelx:validFrom>
15                         <bpelx:validUntil>2019-01-01T00:00:00Z</bpelx:validUntil>
16                         <bpelx:versionInfo>Short description of version 1.1.</bpelx:versionInfo>
17                 </bpelx:version>
18         </bpelx:versions>
19
20         <bpelx:version vid="1.0">
21                 <!-- defines version 1.0 here -->
22                 <import .../>
23                 ...
24                 <sequence>
25                         ...
26                 </sequence>
27         </bpelx:version>
28
29         <bpelx:version vid="1.1">
30                 <!-- defines version 1.1 here -->
31                 <variables>...</variables>
32                 ...
33                 <sequence>
34                         ...
35                 </sequence>
36         </bpelx:version>
37 </process>
```

2.5 Difference Calculation

In the previous section, it was outlined that versioning approaches are calculating differences between the current state and the previous version. This section gives an overview to the more abstract problem: the calculation of differences between two versions of a given model. According to Brun and Pierantonio [BP08], the difference calculation between two models can be separated into the three phases described below. During this section consider the following example: a person named Peter used to live in Berlin in 2017 but moved to Stuttgart in 2018. Further, he replaced his car with a new bike. Both versions of Peter are shown as an UML Object Diagram in Figure 2.6. In the following, model A and model B always refer to two subsequent versions of the same model, whereas A is considered the "old" and B the "new" version. Peter's old model shown on the left side of Figure 2.6 represents model A, while the left side is an instance of model B.

Calculation Differences between two distinct models have to be calculated. This is done by a specialized algorithm. In general, during the calculation phase a transformation from model version A to version B must be found [CRGW96].

Representation The representation is the outcome of the calculation step. It represents the changes in an abstract way that they can be further processed. In the example, the representation would contain the operations required to transform Peter: 1.) move Peter from Berlin to Stuttgart, 2.) remove the car, and 3.) add his new bike.

Visualization To present the changes to a user, the representation needs to be processed to visualize all modifications. This can be achieved in various ways, for example, graphically or textually. Following the example, all changes Peter performed between 2017 and 2018 can be further processed into any from required. Listing 2.2 visualizes the differences in a textual way.

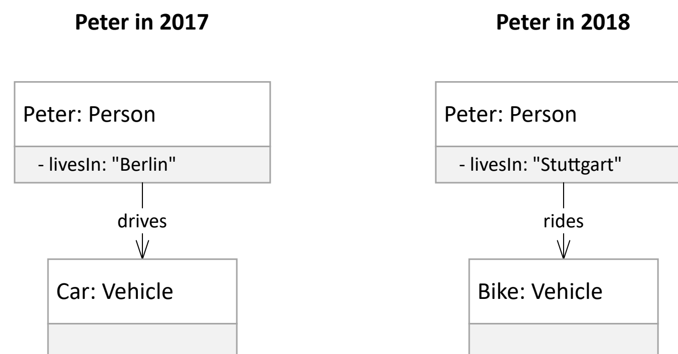

Figure 2.6: Example model to visualize differences

Listing 2.2 Textual difference visualization of the example model

```
1        Peter moved from Berlin to Stuttgart.
2        Peter sold his car.
3        Peter bought a bike.
```

2.5.1 Calculation of Differences

The calculation phase itself can be separated into two sub-phases: model matching and calculating differences between the corresponding elements [SC13]. During the matching phase, correspondences between the elements in model A and B are determined. For each element in the example above, all corresponding elements must be found. In this case, Peter is identified in both models whereas his car only exists in the 2017 version and his bike solely in 2018.

Further, because models can be represented as graphs [ASW09], model matching can be reduced to the subgraph isomorphism problem [Epp02]. Thereafter, model matching is more difficult to solve than the actual calculation of differences itself [KRPP09], which is straight forward given two matching elements [BP08]: 1.) if one element exists in the old model, but not in the new one, it is considered removed, 2.) if an element exists in the new version but not in the old, it was added, and 3.) if the new value of an element does not match its old value, it was changed.

The graph isomorphism problem states that two graphs G_1 and G_2 are considered isomorph, if and only if all nodes in both graphs are connected to the same neighboring nodes [GY04]. Similarly, the subgraph isomorphism problem finds isomorphic subgraphs in both graphs G_1 and G_2. Searching for corresponding elements between two models can be transformed to finding isomorphic subgraphs in both models as elements may have been added, deleted or moved. The outcome of a difference calculation must be a set of operations that transform G_1 into $G_1^{'}$ which is isomorph to G_2 [CRGW96].

In general, the subgraph isomorphism is a hard task to solve [Epp02]. To avoid solving the general problem, Chawathe et al. [CRGW96] stated, that the matching is trivial, if the nodes can be identified using a specific identification method. Kolovos et al. [KRPP09] found four identification strategies which are used to determine two matching elements in two given models.

Id-based Matching In an id-based matching algorithm, each model element has a document wide unique and static Identifier (ID). This identifier is created only once upon the element's creation and is used to uniquely identify an element.

Signature-based Matching In contrast to id-based matching, a signature is calculated every time a matching is performed between two elements. Therefore, it can be used on independently developed models but requires a signature function for every element. Further, universally unique identifiers (UUIDs) are a compound of id-based and signature-based matching, since the ids are generated using a signature or hashing strategy.

Similarity-based Matching Similarity-based algorithms employ a fundamentally different approach for matching two model elements. It relies on heuristics calculated for each element during the matching phase. These heuristics rely on weights which must be assigned to every element's attributes in order to introduce different levels of importance. If two elements share an specific amount of similarities, they are considered to be a match.

Custom Matching In custom matching algorithms, users can define their own matching techniques. The advantage of this approach is that meta-model dependent characteristics can be exploited to reduce processing time and improve the results.

Neither of the four approaches qualifies as the best solution because all matching methods have their up- and downsides, and should be chosen with the respective trade-off [KRPP09]. In the following, approaches using different matching techniques are presented.

In 1996 Chawathe et al. [CRGW96] published a change detection approach for hierarchically structured information with this information being available in ordered trees. Because these trees may not have keys or identifiers, the matching must be performed using the labels and values contained in each node. Therefore, similarities between the node's elements and properties are calculated. If the result of the similarity calculation exceeds a specific threshold, the nodes are considered to represent the same node in both models and are therefore a match [CRGW96].

An approach for matching model elements in a customizable way was presented by Kolovos [Kol09], where the Epsilon Comparison Language (ECL) (based on the Eclipse Epsilon[2] framework) has been introduced. ECL is an approach for defining specific criteria to achieve model matching. Therefore, the user needs to specify high-level Match Rules which are executed to decide whether the elements match or not. The author himself admits that standard similarity matching usually suffices considering the effort for creating the custom rules. Nevertheless, the Epsilon Comparison Language can be used to define a highly customized matching by utilizing the corresponding meta-model mostly used in corner cases.

Alanen and Porres [AP03] described an approach for calculating differences between two given UML models. First, all elements in both models M_{old} and M_{new} are mapped using any matching technique (in their particular case UUIDs were used). Afterwards, the actual differences calculation starts by searching for elements that exist in M_{new} but not in M_{old} to find the created elements. For every new element a `new` operation is added to a list C. Similarly, all deleted elements are collected in a second list D. Finally, every attribute of each element is checked for changes, additions and deletions, which are summarized in a third list F.

2.5.2 Representation of Differences

The result of the difference calculation must be stored in a way that it can be further processed. This is commonly known as difference representation. It represents all differences between two model versions and can be implemented in two ways: *directed deltas* or *symmetric deltas* [Men02]. Directed deltas define changes from one version to another as a sequence of operations that transform model A to model B [Men02]. Symmetric deltas describe the changes between two models as a complete set containing all elements of both models and emphasizes all changes [Men02].

As a result of the difference calculation presented by Alanen and Porres [AP03], the whole set of changes is represented as an array of atomic operations: $\Delta = [C, D, F]$. C hereby contains a list of all created, and D of all deleted elements. F contains changes, additions and deletions of an element's attributes. This list of atomic operations is commonly known as edit scripts and has been initially introduced by Chawathe et al. [CRGW96]. They provide a step-by-step instruction on how to modify the base version in order to create the new model version. In variations, the additional atomic action move is introduced [CRGW96; KKU08].

[2] https://www.eclipse.org/epsilon/

Cicchetti et al. [CDP07] propose an approach for calculating differences between models independent of their meta-models. They create a new Difference Meta-Model (MMD) derived from the original meta-model by extending each model element with the classes *Added*, *Deleted*, and *Changed*. According to the "everything is a model" paradigm [Béz05], differences between two models are thereafter represented as Difference Models. In the following the meta model shown in Figure 2.7 is considered. It represents a meta model describing the models shown in Figure 2.6. Following this approach, the MMD displayed in Figure 2.8 is generated to represent changes in a model conforming to the meta-model shown in Figure 2.7. The difference model resulting from comparing Peter in 2017 with his version in 2018 (cf. Figure 2.6) is depicted in Figure 2.9.

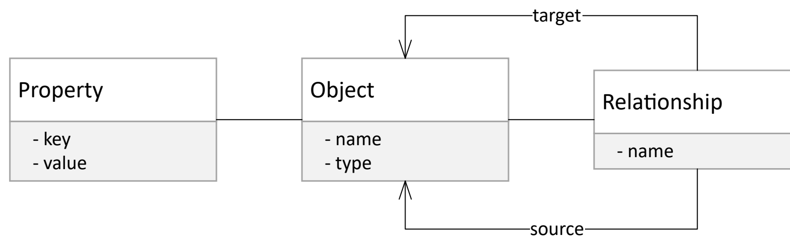

Figure 2.7: Meta model for the models shown in Figure 2.6

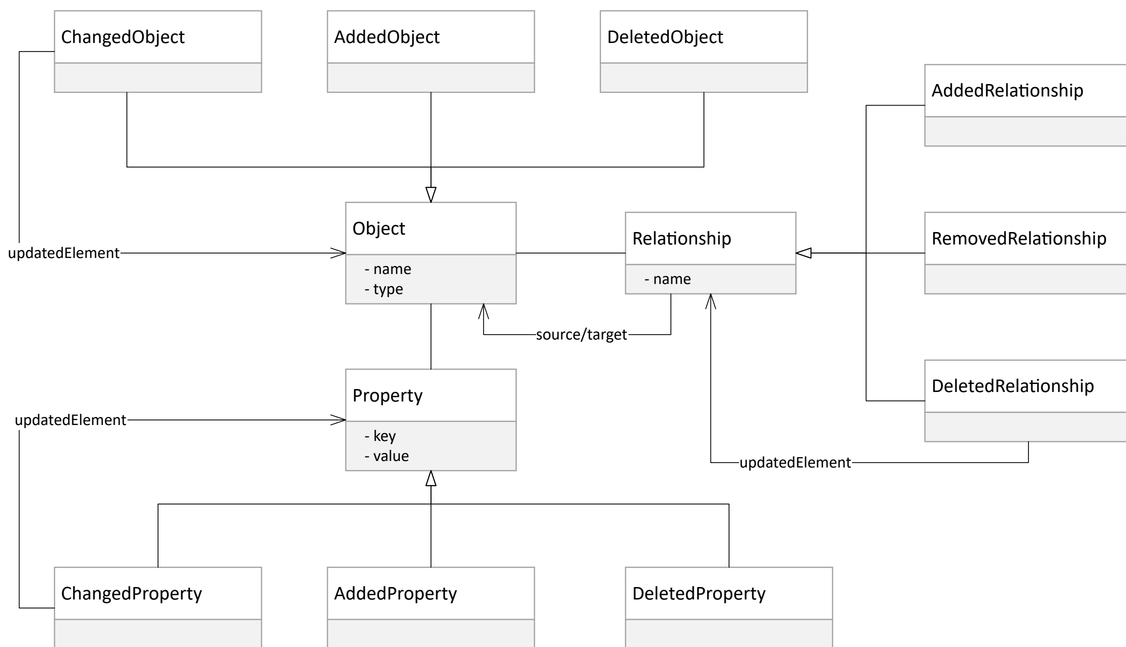

Figure 2.8: Difference Meta-Model (MMD) according to the meta model shown in Figure 2.7

Another method for calculating and presenting differences between models to the user is described by Ohst et al. [OWK03]. They use separate objects to describe the UML model elements in a "fine-grained data model". Each document, class, operation and attribute is defined as an independent object representing the elements of a given UML model. To compare both data models, the spanning trees the data objects of both models form are traversed simultaneously whereas matching sub-trees are located. The result is a symmetric delta representation containing all information for building a unified difference model.

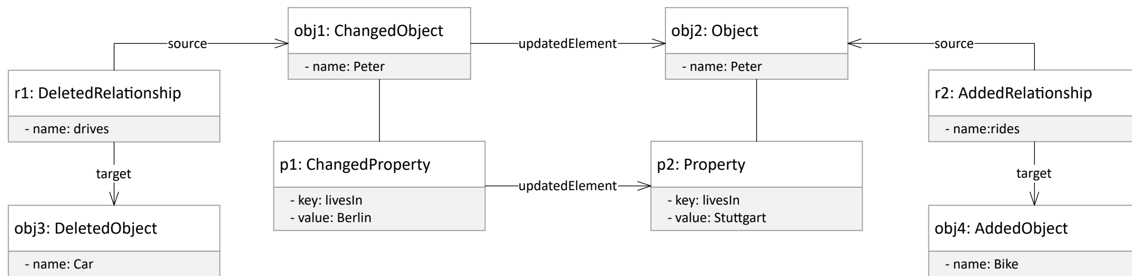

Figure 2.9: Difference model of the models shown in Figure 2.6

2.5.3 Visualization of Differences

The visualization is the presentation of differences between two objects, models or graphs to a human user [BP08]. It is usually very domain specific because the representation is applied to the base model along with a coloring technique [ASW09].

An example for color coding the model differences was presented by Ohst et al. [OWK03]. The authors argue that three colors suffice to signal added, removed and unchanged elements. However, it is also outlined that too many changes between both models generate a very colorful diagram. In this case, they only focus on desired information. Therefore, Ohst et al. added the possibility to restrict the coloration to relevant information only. All other changes are painted in grey [OWK03].

Könemann et al. [KKU09] published an approach called change lists. These change lists contain the automatically calculated differences between two models in a table-like structure. For each entry, the type (add, change, move, and remove), id of the changed element, description, as well as its old and new value are recorded. In contrast to Alanen and Porres [AP03] a fourth atomic action move was introduced to combine the actions add and delete of a single element into one action.

A common approach used in many open source software projects is the changelog [Lac17]. It provides a kind of textual difference visualization based on Markdown[3]. In the changelog, each added, changed, or removed element or feature is shortly described for each release [Lac17]. An example for a chagelog can be found in Section 4.4.

2.6 Package and Version Naming

To run a software written in a language like Java, the compiled source code, along with all required libraries, are bundled into an executable package. Because multiple packages of the same software in different versions exist, they have to be distinguished from each other. Therefore, the version identifier must be added to the package name. For example, the runtime container for Java servelets Tomcat always contains its major version in its name. The currently supported versions are called `Tomcat 7`, `Tomcat 8.5`, and `Tomcat 9`.

[3] https://daringfireball.net/projects/markdown/

According to a blog post by Verma [Ver18], the Linux distributions Debian, Ubuntu, openSuse, and Fedora are most suitable for developers. Hence, there are many different software packages available for those operating systems which are required by software developers. To maintain uniformity in all package names, all distributions require the names and versions to conform to their naming policy [Red+17; Rus17; SUS+17]. Since Ubuntu also employs Debian packages, it is covered by the naming policy described by Debian.

The naming of packages in these three Linux distributions is mostly similar. Debian allows lowercase alphanumeric characters as well as pluses "+", dashes "-" and dots "." [Rus17]. Additionally, Fedora [Red+17] and OpenSuse [SUS+17] also allow uppercase letters and underscores "_". However, Fedora and openSuse both prefer dashes over underscores and even forbid dashes, pluses and dots as separators for name parts.

While the policies for naming packages are quite similar in Debian, Fedora, and OpenSuse, the policies regarding a package's version identifiers differ much more. Epochs, for example, can be used to correct mistakes in the version numbering [Rus17] and are allowed in Debian and should be used, if the version numbering scheme changes. In contrast to that, Fedora encourages developers not to use epochs and OpenSuse has not even implemented them. Similar, tilde versions are used to define pre-releases in OpenSuse and Debian but are forbidden in Fedora. A tilde version, for example `1.3~1`, is supposed to define a pre-rlease of the version `1.3`. The resulting order is defined as follows:

$$1.3.4 \sim 1 \quad < \quad 1.3.4 \sim 2 \quad < \quad 1.3.4$$

A common approach for assigning versions to software components or packages is semantic versioning [Pre13]. It defines three levels of a version identifier in the form `major.minor.patch`. Further, additional labels for describing pre-releases are also allowed after the patch level.

Besides the manual generation of version numbers, which is usually used in all the approaches mentioned before [BB09], Bauml and Brada [BB09] proposed an algorithm to automatically calculate the new version number for a software patch. In their approach they rely on a scheme similar to the semantic versioning approach [Pre13] to indicate a software's version. Depending on the changes' severity, they decide whether the patch level, the minor version, or the major version must be updated. It is outlined, that if the Application Programming Interface (API) is not changed at all, the patch level, if there are only specializations, the minor version and lastly, if there are generalizations or mutations, the major version will be increased by 1. This not only eases the finding for the correct version number, but also ensures that the version clearly indicates compatible and incompatible API changes.

2.7 Regular Expressions

Regular expressions offer a language to formally describe valid text or strings following a specific schema [Fri02]. To declare a schema as a regular expression, there are characters which are standing for specific elements. These characters are 1.) a dot "." which represents any character, 2.) a caret "^" marking the start of a line, 3.) a dollar "$" stands for the line end, 4.) brackets "[ ]" define a matching class, and 5.) a backslash "\" is used to escape the special characters [Fri02]. Further, it is possible to define quantifiers, boolean expressions, and groups [Fri02].

Characters such as “*”, “+”, and “?” represent quantifiers matching either zero or any number of elements (*), at least one element (+), or at most one element (?) [Fri02]. For example, the expression `A+B*C?D` requires a matching string to contain at least one A which is followed immediately by any number of Bs, a subsequent but optional C, and exactly one D. The strings `AD`, `AAABBBBD`, or `ACD` are conforming to this expression whereas `BCD` or `ABC` are invalid since the required A, or D respectively, is missing.

It is possible to define elements that can be used interchangeably or must not be contained. A vertical bar “|” defines an option to either match one of the defined elements. `apple|banana|peach` is an example to only match stings containing apple, banana, or peach. Similarly, every character stated inside the brackets but after the caret “[ˆ]” must not be included in a matching string. For instance, the expression [ˆ`abc`] matches any characters except for a, b and c.

Additionally, parenthesis are grouping expressions. Consider the following expression matching a x followed by any digit from 0 through 9: `x([0-9])`. The parenthesis surrounding the digits expression can be used to directly refer to the contained number only. For example, the string `x4` matches the expression. While group 0 refers to the whole matching string (x4), group 1 directly points to 4.

3 An Approach to Version TOSCA Definitions

This chapter presents concepts for a versioning approach for TOSCA. A TOSCA definition hereby refers to all element types defined in TOSCA (*Service Templates*, *Node Types*, *Relationship Types*, etc.). It first starts with requirements a versioning approach should fulfill (Section 3.1) and continues with a detailed description about the version identification (Section 3.2) and all other actions and concepts required to ensure a proper versioning: support for already existing definitions (Section 3.3), detecting version states (Section 3.4), applying changes and bug fixes (Section 3.5), and how a definition can be released in a specific version (Section 3.6).

During this section, tables are summarizing all considered alternatives solving a specific problem. They are stated along with their respective up- and downsides. Because Markdown Architectural Decision Records (MADR) is intended to enrich source code repositories [KAZ18], the tables follow a similar but condensed approach for summarizing alternatives.

The running example presented in Section 1.3 is now extended to contain version identifiers introduced during this section. Therefore, the `Webshop` Node Type is now named `Webshop_1.2-w2`, `Tomcat_8` becomes `Tomcat_8-w2`, `Java_8` becomes `Java_8-w1`, `Ubuntu_16.04` becomes `Ubuntu_16.04-w3` and `Amazon_EC2` stays the same. The service topology is defined in the `WebshopService_-w1` Service Template. The topology is shown in Figure 3.1.

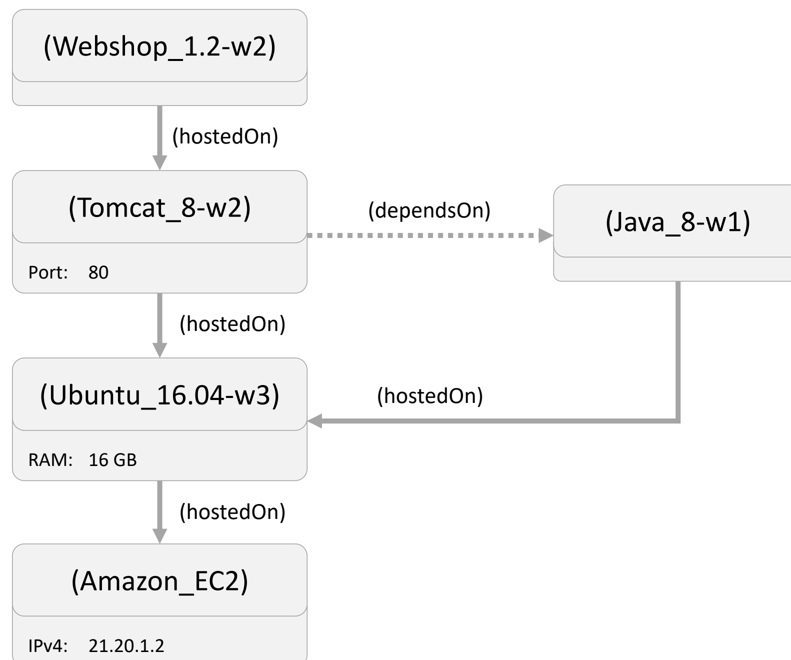

Figure 3.1: Extended example topology including version identifiers.

3.1 Requirements

In this section, all requirements are listed and described. The abbreviation VR hereby stands for Versioning Requirement followed by its id.

Requirement 1 (VR1): Conformity to the TOSCA specification. A versioning approach for TOSCA definitions must conform to the standard specification given in [OAS13b]. It is required to enable all TOSCA implementation to understand the approach without adaptations.

Requirement 2 (VR2): Detectable version identifier inside the definition. It must be possible to detect the current version of a TOSCA definition while reading it. This is required because TOSCA definitions are defined in XML files.

Requirement 3 (VR3): Support component versions. Node Types, for example, represent external components such as a Tomcat web server or a Java virtual machine. In case such a component has its own version identifier, it must be represented in the management layer as well.

Requirement 4 (VR4): Support a version identifier. The foundation of the approach is to version TOSCA definitions. Hence, it must be possible to define an identifier to differentiate the versions. This represents the core of this approach. It is named winery version because the Winery is used to implement the approach prototypically.

Requirement 5 (VR5): Support work-in-progress versions. During the development of definitions, it is necessary to allow the users to create work in progress (WIP) versions which can be used for testing purposes. After a work in progress (WIP) version has been successfully tested, the definition should be released in the corresponding winery version.

Requirement 6 (VR6): Support existing definitions. A versioning approach must support already existing definitions in a way that they can be displayed, reused and new versions can be added. This ensures that previous work is still valid and can still be maintained.

Requirement 7 (VR7): Committed and released versions must not be changed. If a version was committed, regardless of whether it is a winery or WIP version, any user must be prevented from further changing the TOSCA definition. Otherwise, the concept of versioning is bypassed. For example, if the Webshop in the WebShopService is updated to use Java 9 features but the Java Node Template remains on version 8, the stack will not start anymore since the application is incompatible with the Java 8 runtime.

3.2 Version Identification

This section discusses possible options for realizing the identification of a version in TOSCA definitions in Section 3.2.1, and describes the chosen option in Section 3.2.2.

3.2.1 Considered Alternatives

The requirement VR2 states that the version should be easily detected just by scanning through the definition. Hence, saving the version identifiers in a separate file per TOSCA definition does not meet this requirement. It is listed in Table 3.1 as option 4. In the following, three additional approaches solving this problem are described and summarized in Table 3.1 along with their advantages and disadvantages.

Similar to BPEL, the TOSCA specification follows the idea of the Extensible Markup Language (XML) and allows extensions to the language in the sense of implementing custom elements coming from a different namespace at certain places. The versioning approach published by Juric et al. [JSR09] slices each BPEL definition into several parts (cf. Section 2.4 and Listing 2.1). Therefore, the interpreter must understand the additional versioning elements in order to interpret a process. This leads to process definitions which cannot be executed by nodes implementing the standard definition only. Extending TOSCA with a similar approach was considered as option 1, but it would violate VR1 (“Conformity to the TOSCA specification”).

Appending the version identifier in the namespace as described earlier in Section 2.4 was investigated as option 2. Namespace provides a way to avoid naming conflicts, hence the name of a TOSCA definition stays the same over all versions. However, there are two drawbacks. First, appending the version identifier of a TOSCA definition to its namespace suggests that all components defined in it have the corresponding version. Because all TOSCA definitions are mostly developed independently from each other this approach introduces a huge number of namespaces. Second, namespace versioning is mainly intended to specify the XML’s schema rather than its content’s version.

The third option, embedding the version identifier into the definition’s name fulfills the requirement of inserting a version into the definition VR2. It not only allows the version to be detectable, but is also a standard compliant solution VR1. As an investigation of Tomcat Node Types developed by the University of Stuttgart showed, all of them already include their component’s version in its id. For example, there are the Node Types with ids “Tomcat-7”, “Tomcat8” and “Tomcat_9”. The only inconsistency found in this pattern was the separation of name and version. To cope with this, a naming schema must be enforced.

A naming schema does not only introduce constraints but also comes with the benefit of uniform definition ids following a common structure. However, applying this approach to TOSCA definitions comes with another drawback. By explicitly defining one version with a specific id per TOSCA definition, the whole definition and definitions referencing it[1] need to be duplicated every time a new version is created. For example, to create a new version of the Webshop Node Type, it must be duplicated to contain the current configuration of the Node Type. If only a new TOSCA definition is added, already defined elements in the previous version are lost.

All four approaches are summarized in Table 3.1. Option 1, a similar approach as proposed by [JSR09] [JSR09] is not working for an versioning approach in TOSCA, because it is violating VR1 (“Conformity to the TOSCA specification”). Further, option 2 states a namespace versioning as suggested by [Ogb02] [Ogb02]. Since namespaces are used for grouping specific TOSCA definitions, this option was also rejected. In contrast to option 3, option 4 is interfering with VR2 (“Detectable version identifier inside the definition”). However, option 3 needs to duplicate

[1] How the elements are linked and referencing each other is discussed in Section 3.5.

TOSCA definitions to create new versions. Considering the small size of a TOSCA definition (e.g., the `Tomcat-7` Node Type has 86KB, already including additional files), and the downsides other approaches bring along, this inconvenience was accepted.

#	Alternative	Advantages	Disadvantages
1	Employing an approach similar to the one presented by Juric et al. [JSR09]	Version specific parts can be detected easily in the XML document	Extension must be understood by any interpreter When exporting a definition, it must be converted into a standard compliant format
2	Appending the version to the namespace as suggested by Ogbuji [Ogb02]	Easy and well established method in XML Definition's name stays intact	Implies that all elements in the corresponding namespace have the same version Used to specify the version of the XML's vocabulary only
(3)	Embedding the version in the component's name	Identification of the version inside the TOSCA definitions file Non-disruptive and standard compliant	Introduces a naming convention Requires to create (multiple) new definitions for each new version[1]
4	Saving the version in an external file	Less disk space required	Does not meet all requirements

Table 3.1: Alternatives for saving the version. Chosen alternative is circled.

3.2.2 Concept for Version Identifiers in TOSCA

In Section 3.2.1 it was decided to append the version identifier to the TOSCA definition's id (option 3).Therefore, a naming schema must be enforced to ensure consistent separation between its id and version. As outlined in the requirements above, the naming schema must support an optional component version VR3 (e.g., Tomcat in version 8), a required winery version VR4, and an optional work in progress (WIP) version VR5. Since TOSCA provides a way to declare components of a cloud application on a management level, the combination of winery and WIP version is named management version.

$$namingSchema = definitionsId_(componentVersion)? - (wineryVersion) - (wipVersion)?$$

$$managementVersion = -(wineryVersion) - (wipVersion)?$$

To be consistent with other open source naming schemes, it was decided to use a schema based on the one employed in Debian's package naming policy (see Section 2.6). The version must be separated from the name using an underscore "_" followed by the component version which can

be a string conforming to the EncName defined in the XML specification [BPS+08]. Because the EncName allows Latin characters only, the tilde ("~") symbol is replaced by a dash ("–"). If the name itself has underscores in it, the part after the last occurrence is considered the version string. Therefore, the component version string must not contain underscores. Additionally, an EncName must start with a letter [BPS+08]. However, since the version identifier must stand after the id, this only applies to the TOSCA definition's id. All these constraints result in the following regular expression describing the TOSCA definition's id and component version:

$$definitionsId = \texttt{[a-zA-Z]}\ \texttt{[a-zA-Z0-9\textbackslash._-]}*$$

$$componentVersion = \texttt{[a-zA-Z0-9\textbackslash.-]}*$$

The winery version is defined by a unsigned number. To set the version number into more context, a lowercase "`w`" is prepended. The w was chosen because the approach is implemented in the Winery and therefore indicates the version inside Winery. However, another implementation may choose a different prefix for the winery version since this does not change the concept. Nevertheless, a separator must be introduced to divide the component version from the winery version. Similar to Debian, a dash "–" is used to separate those two.

$$wineryVersion = \texttt{-w[0-9]}+$$

Similar to the winery version, the WIP version is also defined by a unsigned number after a leading `wip` label. It is separated from the winery version by a dash. Following these rules, the work in progress version is defined as follows:

$$wipVersion = \texttt{-wip[0-9]}+$$

Combining all version expressions into one expression, the requirements must also be followed. Because the component version, as well as the WIP version must be optional, the following regular expression describes the naming schema which applies to TOSCA definition ids.

$$(\texttt{[a-zA-Z]}\ \texttt{[a-zA-Z0-9\textbackslash._-]}*)\ _\ (\texttt{[a-zA-Z0-9\textbackslash.-]}*)?\ (\texttt{-w[0-9]}+)\ (\texttt{-wip[0-9]}+)?\ \$$$

On creation of a new TOSCA definition, the winery and the WIP versions are set to 1 as the default. If there is no winery version then there is no WIP version allowed. Further, if there is no winery or WIP version, their respective version number is defined to be zero and vice versa: if the winery or WIP version is defined to be zero, they are considered to be not existent.

To present all versions of a given TOSCA definition to a user, they must be ordered to ease their readability. WIP versions are considered to be predecessors of the actual winery version. Additionally, a component version is considered to be newer than another, if a common string comparison considers it to be higher. A TOSCA definition in component version `1.3.5` is newer than version `1.3.4`. For example, a WIP version is newer than a winery version, if its winery version is greater. Therefore, the versions are ordered according to the following scheme:

$$\texttt{1.3.4-w1} < \texttt{1.3.4-w2-wip1} < \texttt{1.3.4-w2-wip2} < \texttt{1.3.4-w2} < \texttt{1.3.5-w1}$$

Adding a new component, or a new component version of a definition, is as follows: after the id of a TOSCA definition, its component version is appended with an leading underscore as well as the winery and work in progress version with their initial values of 1. For example, adding a new Node Type `Tomcat` in the version `9.1` will result in the final name `Tomcat_9.1-w1-wip1`. Although the component version is optional, the winery and WIP versions are required and must be appended upon the creation of a new version. A winery version of 1 and a WIP version of 1 is also referred to as a "new basic version"

Table 3.2 shows TOSCA definition ids and their corresponding version in ascending order. The ids are ordered according to their names, and versions. In the first line, the id `Tomcat` is listed. Since it does not specify any version identifier, the default values are employed. These are 1.) an empty string for its component version, 2.) a winery version of 0, as well as a work in progress (WIP) version of 0.

In the second line of Table 3.2, the id `Tomcat_-w3` is shown. According to the naming schema, it defines a valid version identifier without a component version but a winery version of 3. Similarly, `Tomcat_6` listed in the third line only defines a component version. While lines 4 and 6 (`Tomcat_7-w1` and `Tomcat_7-w2`) define a component version and winery version only, `Tomcat_7-w2-wip168` (line 5) and `Tomcat_8.3.5-beta1-w1-wip4` (line 7) also have WIP versions. Because `Tomcat_9-wip3` in line 8 defines the WIP version at an incorrect place, its component version is `9-wip3`.

Definitions id	**Name**	**Component**	**Winery**	**WIP**
Tomcat	Tomcat		0	0
Tomcat_-w3	Tomcat		3	0
Tomcat_6	Tomcat	6	0	0
Tomcat_7-w1	Tomcat	7	1	0
Tomcat_7-w2-wip168	Tomcat	7	2	168
Tomcat_7-w2	Tomcat	7	2	0
Tomcat_8.3.5-beta1-w1-wip4	Tomcat	8.3.5-beta1	1	4
Tomcat_9-w1-wip3	Tomcat	9	1	3

Table 3.2: Valid TOSCA definition ids in ascending order including their resulting versions

While Table 3.2 only defines valid names, Table 3.3 shows special cases. The id defined in line 1 (`Tomcat_w1`) is not correctly following the naming schema. Instead of the winery version 1, the component version is considered to be `w1` because the separator is not prepended. A similar problem arises in line 2 with `Tomcat_7-wip6`. WIP versions are only allowed, if a winery version is placed in front. Therefore, the component version results in `7-wip6`. `Tomcat_8_w1-wip2` (line 3) has two underscores in its id. Since the last underscore is considered to separate the version from the id, the name of this TOSCA definition is `Tomcat_8` with the component version `w1-wip2`.

Definitions id	Name	Component	Winery	WIP
Tomcat_w1	Tomcat	w1	0	0
Tomcat_7-wip6	Tomcat	7-wip6	0	0
Tomcat_8_w1-wip2	Tomcat_8	w1-wip2	0	0

Table 3.3: Valid TOSCA definition ids which incorrectly define versions

3.3 Support for Existing TOSCA Definitions

Requirement VR7 ("Committed and released versions must not be changed") leads to the conclusion that committed definitions must not be changed to ensure runnable Service Templates. However, this was a common case developers reported using the OpenTOSCA ecosystem. In addition, TOSCA definitions developed in previous work, for example by Franco da Silva et al. [FBK+16b], must still be supported and interpreted correctly, requirement VR6 ("Support existing definitions").

Considering this, three alternatives were under investigation to support existing TOSCA definitions. The following options were considered: 1.) enforce the naming schema for all TOSCA definitions, 2.) fully support TOSCA definitions which are not following the schema, and 3.) a combination of both variants where new versions must follow the naming schema, and TOSCA definitions which do not can be displayed but not edited, imported TOSCA definitions for example. Table 3.4 summarizes all options along with their advantages and disadvantages.

First, all TOSCA definitions must always have a valid version. While this approach makes it easy to deal with all definitions (e.g., for updating them), it requires a huge overhead upfront. Since not every component version of each definition follows the semantic versioning approach (introduced in Section 2.6), the handling of separating the name parts must be performed manually in most cases. Further, if the component version is not appended at the end of the name, it is very difficult to detect the elements belonging to the version identifier and those which belong to the name. For instance, if a TOSCA definition has the following id: `Webshop_1.3-test-definition_Peter`, it is difficult to automatically extract the actual version of this TOSCA definition.

The second option is to fully support definitions without any version. If they are fully supported, it must be possible to edit them. This results in the issue that requirement VR7 ("Committed and released versions must not be changed") is violated.

By combining both approaches, a third, hybrid solution was considered, where components without a version can be imported without any renaming. If they need to be edited, a new version and therefore a new definition must be added. All alternatives along with their corresponding advantages and disadvantages are summarized in Table 3.4.

In general, if components are not following the naming convention described in Section 3.2.2, an automatism or the user must ensure that the definition's name is valid. Otherwise, if a name incorrectly uses underscores, for example to separate name parts, and does not end with a valid version, the version identifier may contain the last part of the name instead of the version. This is the case for the Node Type `Amazon_EC2`. "Amazon" is considered the Node Type's id while "EC2" is interpreted as its component version. This issues will be present in all listed alternatives and needs to be handled by the user.

#	Alternative	Advantages	Disadvantages
1	Force all TOSCA definitions to have a convention compliant id and version	Naming convention is enforced	On importing TOSCA definitions, user needs to ensure that all follow the convention
		Easy to update all TOSCA definitions, with and without versions	Creates a huge overhead upfront
2	Fully support definitions without any version	TOSCA definitions without a version are fully supported	Non-versioned TOSCA definitions can still be edited and cause unexpected failures (interferes with VR7)
③	Allow to display TOSCA definition without a version, edits are only allowed on versioned TOSCA definitions	Components without a version are supported	If the naming convention is violated, the automated detection of version identifiers is difficult
		Committed versions cannot be edited anymore, changes require a new version	

Table 3.4: Alternatives to support existing TOSCA definitions. Chosen alternative is circled.

3.4 TOSCA Definition Version States

A TOSCA definition is at any time in one of four different version states. A version state indicates the types of changes which can be applied to a TOSCA definition. All four states Editable, Committed, Releasable and Released were identified during a discussion on how to categorize versions of TOSCA definitions by Kopp and Harzenetter [KH18]. They are shortly explained in the following.

Editable A TOSCA definition in the editable state can be edited and committed.

Committed In the committed state, a TOSCA definition cannot be changed anymore.

Releasable A TOSCA definition which can be released as a stable management version. It is a refinement for the committed state.

Released The released state indicates a stable management version. It is a special case of the committed state.

3.4.1 State Transitions

Figure 3.2 describes the transitions from one TOSCA definition version state into another. If a version of a TOSCA definition is in the editable state, it can be committed. The committed state is refined by two sub states, releasable and released. A TOSCA definition version is only transfered into the releasable state, if its management version is the latest of its component version. This is discussed in more detail in Section 3.4.2

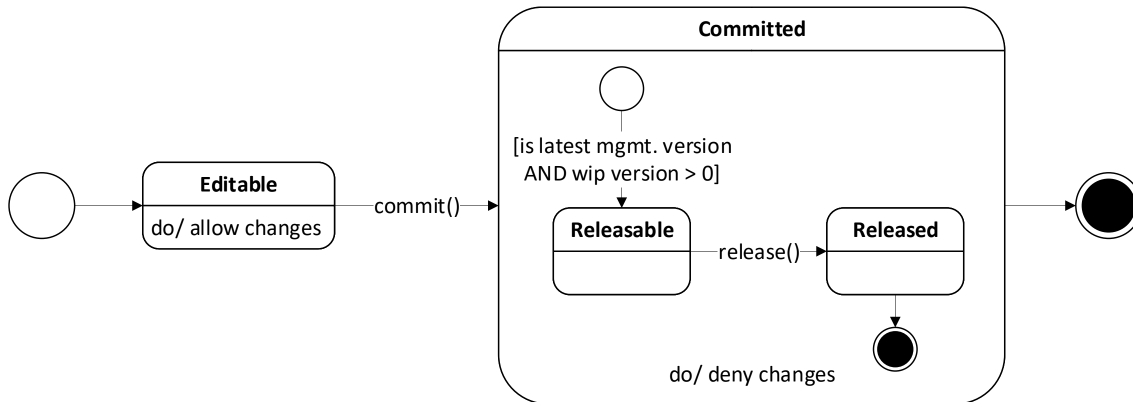

Figure 3.2: State diagram defining TOSCA definition version state transitions

3.4.2 Determining Initial States

Determining the initial TOSCA definition version state is always required, if a TOSCA definition is loaded. The reason for this is, that the states are not saved in the XML files. Figure 3.3 illustrates how the initial state of a TOSCA definition version can be determined. The steps can be summarized into the following five steps: 1.) parse the version from the id, 2.) check, if the winery version is greater zero, 3.) determine, if the current TOSCA definition is the latest management version, 4.) decide, whether the TOSCA definition was already committed, 5.) check, if the WIP version is greater than zero.

First the version has to be extracted from its id. This is done by validating the TOSCA definition's id against the naming schema. If it does not conform to the schema, the version contains default values: an empty string for the component version, and 0 for both management versions (see Table 3.2 for an example in line 1). Afterwards, the winery version is investigated. If it is greater than zero, the TOSCA definition has a valid management version. In the case that no valid management version was found (the winery version is equal to zero), the definition is considered to be in the released state which prevents it from further changes. The released state was chosen over the committed state because an imported TOSCA definition without a valid version should be treated as a released TOSCA definition [KH18].

If the winery version is greater than zero, it must be decided whether the version is the latest management version of its component version. For example, consider the following versions of the Tomcat Node Type: `7-w1-wip1`, `7-w1-wip2`, `7-w1`, `7-w2-wip1`, and `8-w1-wip1`. If version `7-w1-wip2` would be in the releasable state and a release is triggered, a conflict would occur since `7-w1` already exists.[2] Similarly, version `7-w1-wip1` must not be in the releasable state either because it was replaced by the newer version `7-w1-wip2`. Therefore, only the latest management version may be allowed to be either editable, or releasable. In the case of Tomcat in component version 7, the management version `-w2-wip1` is meeting this requirement and is further examined.

[2] For more details on releasing a TOSCA definition, see Section 3.6.

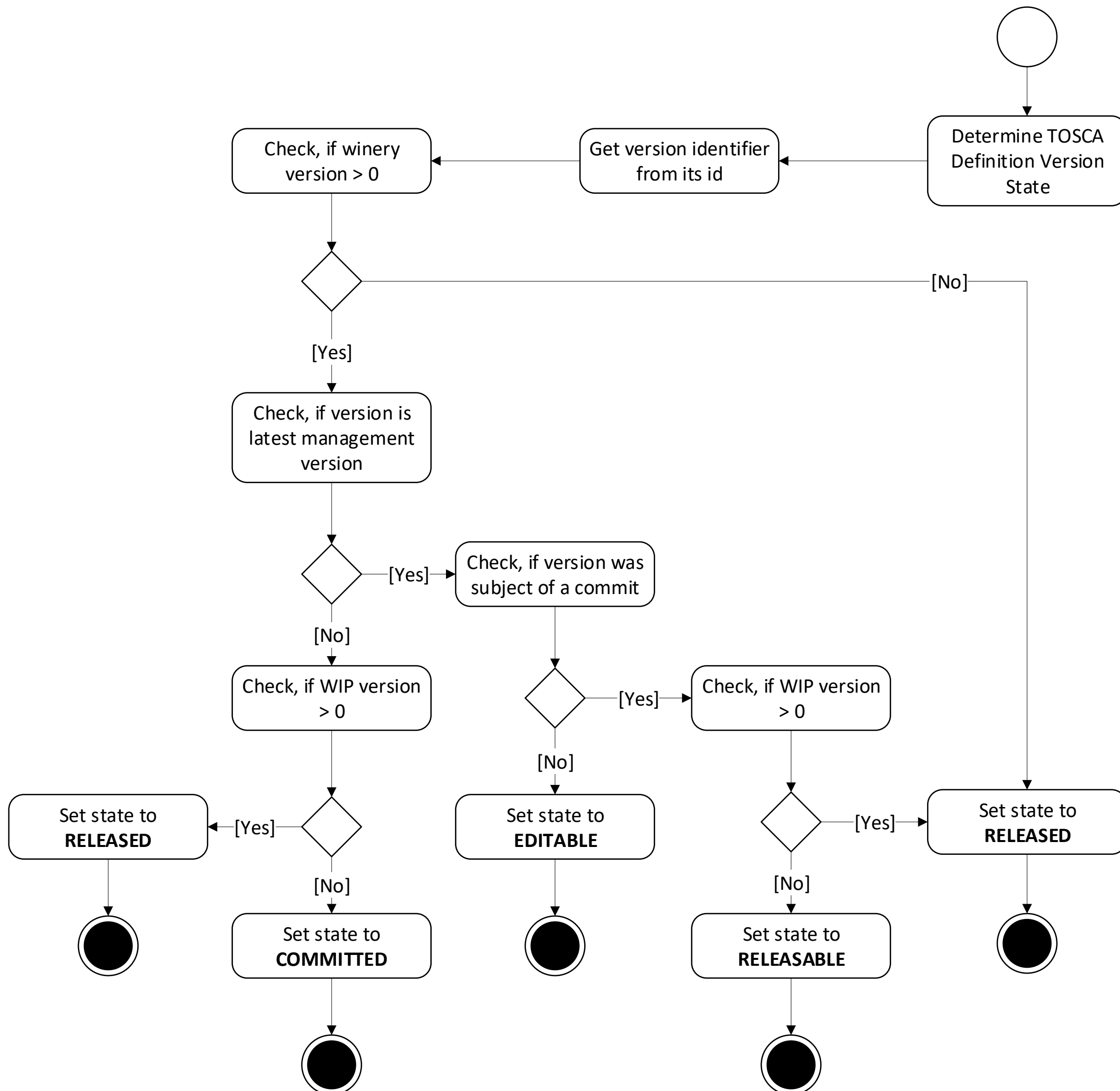

Figure 3.3: Activity diagram to decide the initial TOSCA definition version state

The Tomcat versions `1.2-w1-wip1` and `1.2-w1-wip2` have a WIP version greater zero which is why they are classified to be in the committed state. In contrast, version `1.2-w1` is a stable release and is thereafter ranked in the released state. The example Node Types are also listed in Table 3.5.

After a TOSCA definition was successfully identified as the latest management version of its component version, it must be checked, if it was committed. If this is not the case, changes are allowed and its version state is editable. Otherwise, its work in progress version is evaluated. Similar to TOSCA definitions which do not represent the latest management version, a WIP version equal to zero puts the TOSCA definition version state into the released state. If a WIP version exists, the state is defined to be releasable.

Table 3.5 shows examples for each version state. It is assumed that no other Tomcat Node Types are defined and `Tomcat_7-w2-wip1`, as well as `Tomcat_9-w1-wip1` were not committed. A checkmark in the respective column identifies the TOSCA definition's version state. According to the naming

schema, the Node Types in the first two lines, `Tomcat5` and `Tomcat-6`, do not define valid versions. Therefore, they are considered to be in the released state. Similarly, the TOSCA definitions listed in lines three and four, `Tomcat_7-w1-wip1` and `Tomcat_7-w1-wip2`, are marked committed. They do not represent the latest management versions of Tomcat version 7 and they are work in progress versions. In contrast, `Tomcat_7-w1` in line five is a stable release and marked as released. According to the assumptions described above, `Tomcat_7-w2-wip1` in line six is in the editable state. Since both management versions defined for Tomcat 8 are committed, `Tomcat_8-w1-wip1` in line seven is in the committed state, while `Tomcat_8-w1-wip2` in line eight is releasable. It represents the latest management version of Tomcat in the component version 8. Lastly, `Tomcat_9-w1-wip1` in the last line is also editable, as it is a WIP version which was not committed.

Definitions id	**Committed**	**Released**	**Editable**	**Releasable**
Tomcat5	–	✓	–	–
Tomcat-6	–	✓	–	–
Tomcat_7-w1-wip1	✓	–	–	–
Tomcat_7-w1-wip2	✓	–	–	–
Tomcat_7-w1	–	✓	–	–
Tomcat_7-w2-wip1	–	–	✓	–
Tomcat_8-w1-wip1	✓	–	–	–
Tomcat_8-w1-wip2	–	–	–	✓
Tomcat_9-w1-wip1	–	–	✓	–

Table 3.5: TOSCA definitions and their derived version states

3.5 Applying Changes to TOSCA Definitions

To change a TOSCA definition a new version has to be added first. This is achieved by duplicating and renaming the TOSCA definition which represents the latest management version of the current component version as explained in Section 3.4.2. Figure 3.4 presents the required steps to update a TOSCA definition. The steps are 1.) the generation of the updated id, 2.) saving the definition, and 3.) if the old version of the TOSCA definition defined external files, those are also copied to the new destination.

If the latest management version of the requested component version is editable, no new version must be created. Instead, the user should be redirected to this version. This avoids the creation of unfinished or uncommitted definitions.

To update the id of a TOSCA definition, several steps are involved: 1.) check, if a new component version is provided, 2.) update the winery version, and 3.) update the WIP version. All steps are shown in Figure 3.5.

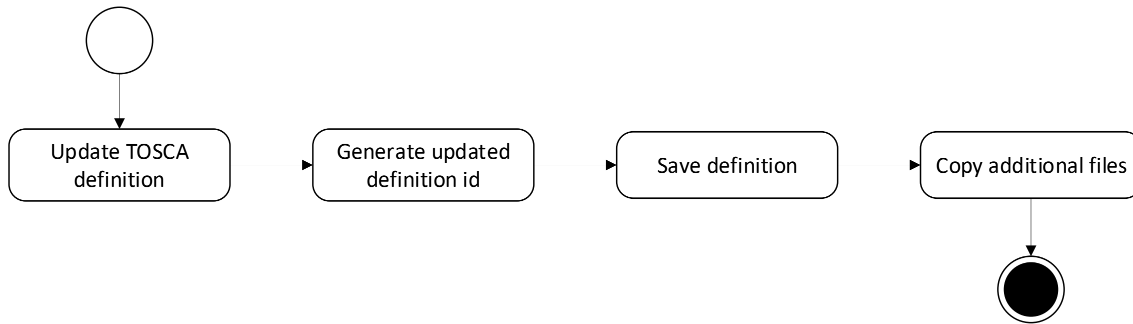

Figure 3.4: Steps required to update a definition

In the first step, it must be checked, if a component version was provided as input. In this case, the generation of the new id is done without further actions because the provided component version replaces the old one, and both management versions are set to 1. For example, the Webshop Node Type is defined in version `1.2-w2`. The resulting id is `Webshop_1.2-w2`. If this TOSCA definition is updated to component version 2.0, the new id is as follows: `Webshop_2.0-w1-wip1`.

If no component version is given, the WIP version must be examined. If it exists, in other words its value is greater than zero, the value is increased by one. Otherwise, the winery version is increased by one and a WIP version of 1 is added. For instance, if the `Webshop_1.2-w2` is updated, the winery version is increased to 3 and WIP version 1 is added: `Webshop_1.2-w3-wip1`. After `Webshop_1.2-w3-wip1` was committed, a new version can be added by updating the WIP version to 2: `Webshop_1.2-w3-wip2`.

If a new management version of a definition is added, other elements may be referencing it and should therefore be considered for an consecutive version update. There are three options how consecutive updates of referencing TOSCA definitions can be achieved: 1.) do not update referencing TOSCA definitions, but warn if outdated versions are referenced, 2.) update all referencing TOSCA definitions, and 3.) a user selects all TOSCA definitions which must be updated. All options are listed along with their advantages and disadvantages in Table 3.6.

The first option listed in Table 3.6 does not change referencing TOSCA definitions. It uses a conservative approach where the currently selected TOSCA definition is updated only. To update TOSCA definitions referencing an outdated version of another TOSCA definition, the user is warned to update the corresponding reference while editing it. This "pulling changes" strategy enables the user to stay in control of all references in the currently edited TOSCA definition. For example, the Requirement Definition of the `Tomcat_8-w2` Node Type references the Requirement Type `JavaRuntime_8-w1`. If `JavaRuntime_8-w1` is updated to component version 9, `Tomcat_8-w2` is not affected. However, while editing the Tomcat Node Type, a warning must be presented, that a new version of the referenced Requirement Type is available. This way, it is also possible to also warn the user about breaking changes. For instance, if a property of the `WAR` Artifact Type was removed in a new version but the currently edited Artifact Template defines a value for this deleted property, the property should be removed from the Artifact Templates to prevent incompatibilities.

Instead of pulling the changes into a TOSCA definition, options 2 and 3 in Table 3.6 present "pushing changes" strategies. Whereas option 2 always updates all TOSCA definitions which are referencing an updated TOSCA definition, option 3 allows the user to select the TOSCA definitions which must be updated automatically while updating the referenced TOSCA definition. For example, if the

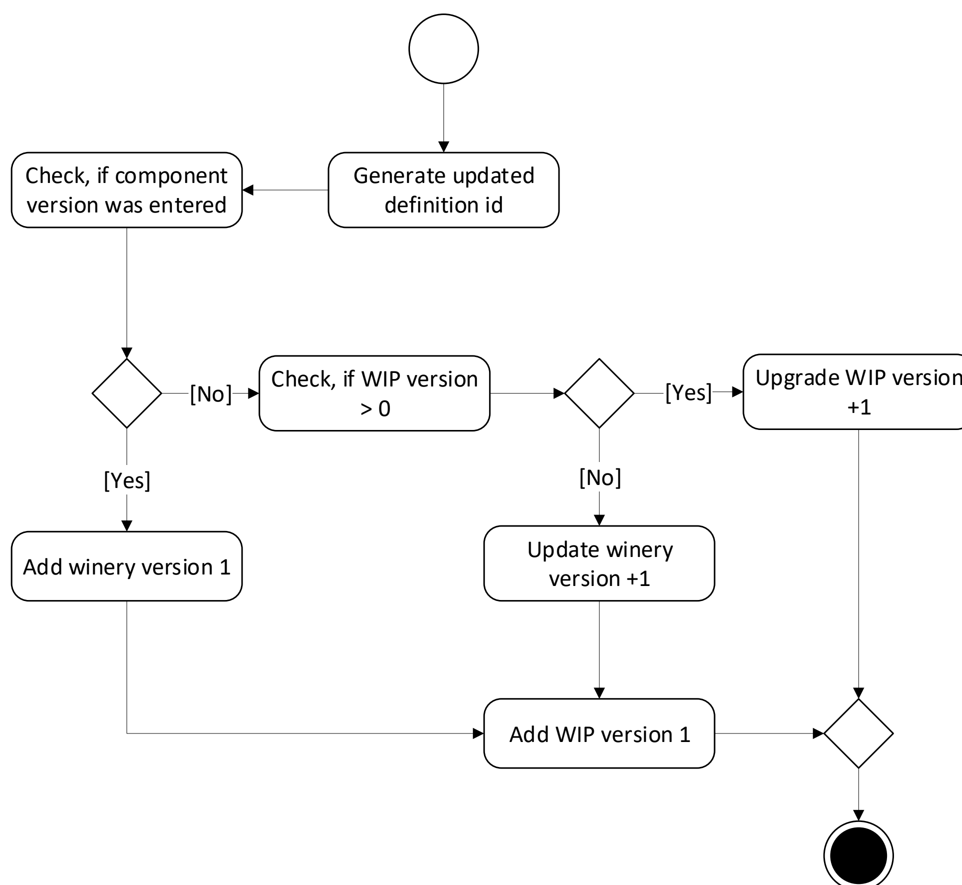

Figure 3.5: Steps to update a TOSCA definition's id

Requirement Type `JavaRuntime_8-w1` is referenced by the Webshop and Tomcat Node Types, a user may only want to update the Tomcat Node Type to reference the new Requirement Type. By employing option 3, this is possible by selecting Tomcat to be updated along with the Requirement Type. As option 2 always updates all referencing TOSCA definitions, this is not possible.

Automatically updating TOSCA definitions can lead to long execution times, if the updated TOSCA definition is referenced by many other TOSCA definitions, or the updated referencing TOSCA definition is also referenced by other TOSCA definitions. For example, if a Tomcat Node Type is derived from a Web-Server Node Type, and a SLA Policy Type, which itself is referenced by another Policy Type, applies to the Tomcat Node Type, there are 4 TOSCA definitions which must be updated in option 2. If the Policy Type is the parent of multiple other Policy Types, this number rapidly increases.

In addition, it is possible, that automatically updated TOSCA definitions become invalid, for example, if a defined property is deleted as explained above for option 1. The difference in options 2 and 3 to option 1 is that it is more difficult to warn the user about these incompatibilities because the warnings cannot be displayed at the appropriate places.

Since option 3 offers optional automated updates, option 1 should be operated in parallel to ease updating references of TOSCA definitions at any time. Therefore, both options were chosen. Which one applies during an update is outlined throughout the remainder of this section for each type of TOSCA definitions.

#	Alternative	Advantages	Disadvantages
(1)	Do not update referencing TOSCA definitions, rather pull updates.	User stays in control of referenced TOSCA definitions. Breaking changes and incompatibilities can be displayed in the respective places.	TOSCA definitions may reference outdated versions of TOSCA definitions.
2	Update all referencing TOSCA definitions.	Automatically updates all TOSCA definitions.	Update may involve changing many other TOSCA definitions. Automatically updated TOSCA definitions may break or contain invalid configurations.
(3)	User selects TOSCA definitions which must be updated.	User stays in control of updated referencing TOSCA definitions. Automatically updates only selected TOSCA definitions. Option 1 can be used in addition.	Update may involve changing many other TOSCA definitions. Automatically updated TOSCA definitions may break or contain invalid configurations.

Table 3.6: Alternatives to update referencing TOSCA definitions. Chosen alternatives are circled.

Besides the TOSCA definition specific updates resulting from references in other TOSCA definitions (explained in this chapter's sequel), Node Types, Relationship Types, Node Type Implementations, Relationship Type Implementations, Artifact Types, Capability Types, Policy Types and Requirement Types can be derived from another TOSCA definition of the same type. Consequently, if one of the named TOSCA definition types is updated, it may also involve updating potential children. However, because children are also referencing the updated TOSCA definition, children only represent a special case of referencing TOSCA definitions.

3.5.1 Automated Update of Referencing TOSCA Definitions

Automatically updating referencing definitions while creating a new version of the referenced TOSCA definition can save many redundant steps: 1.) creating a new version of the referencing TOSCA definition, 2.) update the corresponding reference (see Section 2.1 for details), and 3.) repeat this for all referencing TOSCA definitions. In the following “reference update” is used as an umbrella to refer to the steps 1 and 2.

To find all definitions referencing a given TOSCA definition, every TOSCA definition in a repository is examined. If it contains a reference to the given TOSCA definition, it is called a “referencing TOSCA definition”.

However, applying updates to all referencing TOSCA definitions may not be allowed considering the version states described in Section 3.4. Therefore, all referencing TOSCA definitions must be filtered if they are not representing their latest management version. Otherwise, illegal versions might be created from already outdated versions. For example, given two versions of the Tomcat Node Type, `Tomcat_8-w1` and `Tomcat_8-w2-wip1`, whereas `Tomcat_8-w1` defines a Requirement Definition of type `JavaRuntime_8-w1`. Further, assume that `Tomcat_8-w2-wip1` does not specify this requirement anymore. If an user creates a new version for the `JavaRuntime_8-w1` Requirement Type and selects all referencing TOSCA definitions to be updated automatically (including the `Tomcat_8-w1`), an invalid version of the Tomcat Node Type would be created as illustrated by Figure 3.6. Instead of building atop of the currently newest management version (`Tomcat_8-w2-wip1`, the newly created version for the Tomcat Node Type would depend on the previous version `Tomcat_8-w1`. This must be prevented to ensure a traceable and linear version development. Therefore, referencing TOSCA definitions must always be the latest management version.

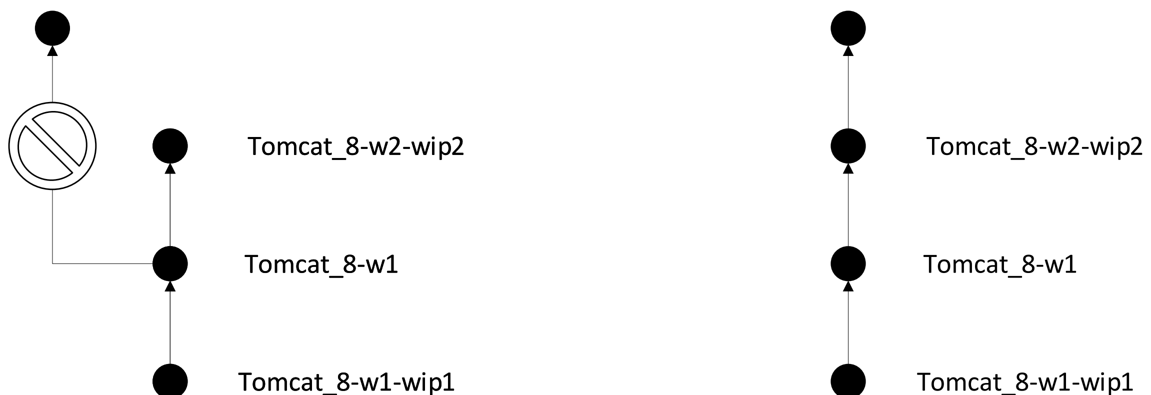

Figure 3.6: Invalid and valid version advancement. Points represent versions of a TOSCA definition, while arrows point to their successors

.

If a referencing TOSCA definition is selected for an reference update, a new management version must be added. This is done by the steps explained above in Figure 3.5. However, if the referencing TOSCA definition is in the editable state (see Section 3.4), the according changes must be performed to the TOSCA definition itself.

3.5.2 Updating Service Templates

If a new version of a Service Template is added, no reference updates are required as it is not referenced by other TOSCA definitions [OAS13b]. However, every update made to a TOSCA definition referenced in a Service Template affects it. To minimize the effect changes may have on Service Templates, it was decided to use option 1 from Table 3.6: pull changes into a Service Template instead of pushing them. This way, the user stays in control of the referenced TOSCA definitions and can decide whether an element will be updated or should remain in its current version while editing the service. Further, one change often requires multiple other changes as well which could lead to semantical conflicts inside a Service Template.

For example, if the `Webshop` Node Type is updated to a new version where it depends on Java 9 instead of Java 8, the stack will not start anymore without further changes to the stack. To avoid this kind of conflicts and other unforeseen side effects, updates to Service Templates are only available while editing it. Another advantage is, that the user can be warned about potential side effects an update would bring along.

3.5.3 Updating Node Types

Node Types are referenced by 1.) Node Type Implementations as their corresponding type, 2.) Relationship Types to define valid sources and targets, as well as by 3.) Policy Types to specify Node Types the respective policy can be applied to. These referencing elements (see also Figure 2.2) can be updated according to option 3 presented in Table 3.6.

If a Node Type is updated to any new version, a Node Type Implementation can be generated, if one was present in the old version. The id must be generated using the Node Types id, including its version, an `-Implementation` string and a new basic version string. For instance, the generated id for an implementation for a `Tomcat_8-w1-wip2` Node Type is `Tomcat-8-w1-wip2-Implementation_-w1-wip1`, where the underscore in the Tomcat's id is replaced with a dash to avoid confusion.

Optionally, it is possible to also update all TOSCA definitions which are referencing a Node Type. For example, if a Policy Type, which applies to the currently updated Node Type, should also apply to the new version, the Policy Type has to be updated to contain the new Node Type in its applies-to list. Therefore, its winery version must be increased.

Relationship Types which define the current Node Type as a valid source or target should also be updated, if the user requests it. Similar changes as described for Policy Types are performed: the corresponding source or target element must be updated to contain the new QName. For example, assuming a `Webshop-hostedOn-Tomcat_-w1` Relationship Type exists where the `Webshop_1.2-w2` Node Type is defined as valid source and `Tomcat_7-w1` as valid target. If the Webshop is updated to the new component version 2, the valid source element is updated to reference the new TOSCA definition, and its version is increased (according to Section 3.5.1) resulting in the new Relationship Type `Webshop-hostedOn-Tomcat_w2-wip1`.

3.5.4 Updating Relationship Types

Relationship Types are only referenced in Service Templates and Relationship Type Implementations. Since changes affecting a Service Template are pulled, only a new Relationship Type Implementation is created, if there exists an implementation for the currently updated Relationship Type.

Similar to the generated id for Node Type Implementations (see Section 3.5.3), the id of the new Relationship Type Implementation is generated. For instance, the generated Relationship Type Implementation id for the `hostedOn_-w2-wip1` Relationship Type is `hostedOn-w2-wip1-Implementation_-w1-wip1`.

3.5.5 Updating Node Type Implementations and Relationship Type Implementations

In contrast to all other TOSCA definitions, Node Type Implementations and Relationship Type Implementations are not referenced by any TOSCA definition (see Section 2.1). Hence, if a new version of an implementation is created, no further updates are required.

3.5.6 Updating Requirement Types

Requirement Types are referenced by 1.) Node Types inside a Requirement Definition, 2.) Relationship Types as a valid source, and 3.) Service Templates as they can expose requirements defined by its Node Templates. However, since changes which are affecting a Service Template must be pulled into it (see Section 3.5.2), an automated update is available for Node Types and Relationship Types only.

While updating a Requirement Type, the user can select reference updates for referencing Node Types and Relationship Types. For example, if the `JavaRuntime_8-w1` Requirement Type is updated the `Tomcat_8-w2` Node Type referencing it will be available for selection to perform a reference update. During the update, a new version is created according to the rules described in Section 3.5.1. Afterwards, the Requirement Definition inside the new `Tomcat_8-w3-wip1` Node Type is updated to reference the new version of the Requirement Type.

Similarly, a Relationship Type referencing the Requirement Type as a valid source can also be selected for an automated update. This results in a new version of the Relationship Type referencing the new version of the Requirement Type as its valid source.

3.5.7 Updating Capability Types

Complementary to Requirement Types, Capability Types can be referenced by 1.) Node Types as a Capability Definition, 2.) Relationship Types as a valid target, 3.) Service Templates to expose capabilities defined by Node Templates, and 4.) Requirement Types where they can be defined as their required Capability Type.

Node Types, Relationship Types, and Requirement Types are available for an reference update, if they meet the requirements described in Section 3.5.1. The automatism will update all references to the old Capability Type version to point to the newly created one.

3.5.8 Updating Artifact Templates

Both implementations, Node Type Implementations and Relationship Type Implementations, are referring to Artifact Templates as they contain the actual files instantiating the respective Implementation Artifact (IA) and Deployment Artifact (DA). Therefore, if an Artifact Template is updated, implementations conforming to the requirements described in Section 3.5.1 can be updated to reference the new Artifact Template's version.

3.5.9 Updating Artifact Types

Artifact Types, are referenced by Node Type Implementations and Relationship Type Implementations, as well as Artifact Templates as their instances. However, if an Artifact Type is updated, implementations referencing it can only be updated to refer to its new version, if there is no Artifact Template specified. Otherwise, the defined Artifact Template may not be of the specified type which invalidates the TOSCA definition because the referenced Artifact Template must be of the declared Artifact Type [OAS13b].

For example, the Node Type Implementation `Webshop-1.3-Implementation_-w1` defines a Deployment Artifact (DA) of type `WAR_-w1`. The artifact reference of the DA points to the Artifact Template `Webshop-1.3-DA_-w1` which is an instance of the Artifact Type `WAR_-w1`. If `WAR_-w1` is updated to a new management version (`WAR_-w2-wip1`), the Node Type Implementation `Webshop-1.3-Implementation_-w1`, and the Artifact Template `Webshop-1.3-DA_-w1` are both available for an automated reference update. In the case that the user only selects the Node Type Implementation (`Webshop-1.3-Implementation_-w1`) for the reference update, the DA is becoming invalid because the updated Artifact Type (`WAR_-w2-wip1`) does not match the Artifact Type of the referenced Artifact Template (`WAR_-w1`).

The example shows that Node Type Implementations and Relationship Type Implementations must not be available for selection for an automated reference update, if a DA or IA defines an artifact reference. However, if multiple DAs or IAs reference the same Artifact Type and at least one does not specify an artifact reference, the whole Implementation can be updated automatically. For example, if the `Webshop-1.3-Implementation_-w1` contains a second Deployment Artifact which is also of type `WAR_-w1` but does not specify an artifact reference, a reference update can be performed on `Webshop-1.3-Implementation_-w1`. Nevertheless, it must be ensured that all DAs and IA defining an artifact reference must not be changed.

3.5.10 Updating Policy Types and Policy Templates

Both, Policy Types and Policy Templates can only be referenced by Service Templates which must not be updated from outside (see Section 3.5.2). Nevertheless, if a Policy Type is updated, the user must be able to select Policy Templates following the requirements explained in Section 3.5.1 for an automated reference update.

3.6 Freezing and Releasing a TOSCA Definition

As already mentioned in Section 3.4, there are two ways to release a TOSCA definition. First, a TOSCA definition is committed, which freezes the current WIP version for testing purpose and changes the TOSCA definition version state to committed. Second, a release can be made to state that the development of a TOSCA definition's management version has been completed, the TOSCA definition version state is updated to be released. In the following, both releases are described in detail, whereas the first one is referred to as "Freeze" and the second one as "Release".

3.6.1 Freeze WIP Versions

A TOSCA definition can only be committed, if it is in the editable version state. In all other cases, it is not editable and must have been committed before (see Section 3.4 for details). Figure 3.7 illustrates the steps required to freeze a TOSCA definition. First, the TOSCA definition version state is retrieved. If the TOSCA definition is in the editable state, it is committed along with all its files. Otherwise, if the TOSCA definition is in any other version state, the action ends immediately.

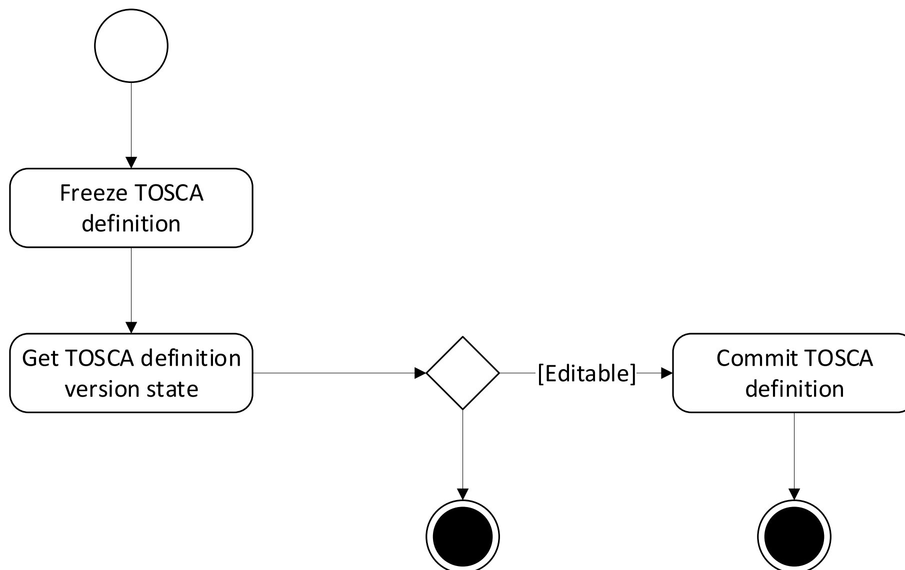

Figure 3.7: UML activity diagram to freeze a TOSCA definition

3.6.2 Release Winery Version

If a developer decides that the current definition is stable, it can be released as the current winery version. Therefore, the definition must be in either the editable or releasable version state (see Section 3.4). If it is in the committed or released version state, the release cannot be performed. Next, if the TOSCA definition is editable it is first committed and duplicated. The additional commit was introduced to clarify the definition's development. Afterwards, the work in progress (WIP) label is removed from the new definition's id to signal a release (see Section 3.2). Finally, the newly created management version is saved and committed. All steps are summarized in Figure 3.8.

Usually, a released definition should not be referencing WIP versions of other definitions to ensure dependencies on stable elements only. Enforcing released definition dependencies during the release itself may lead to an untested and therefore unstable, released definition. If a TOSCA definition references other unreleased TOSCA definitions, the user must be warned about these dependencies or even prevented from releasing the TOSCA definition to either release them first or update the current TOSCA definition to depend on already released dependencies. However, the concepts required to achieve this are not discussed in this thesis.

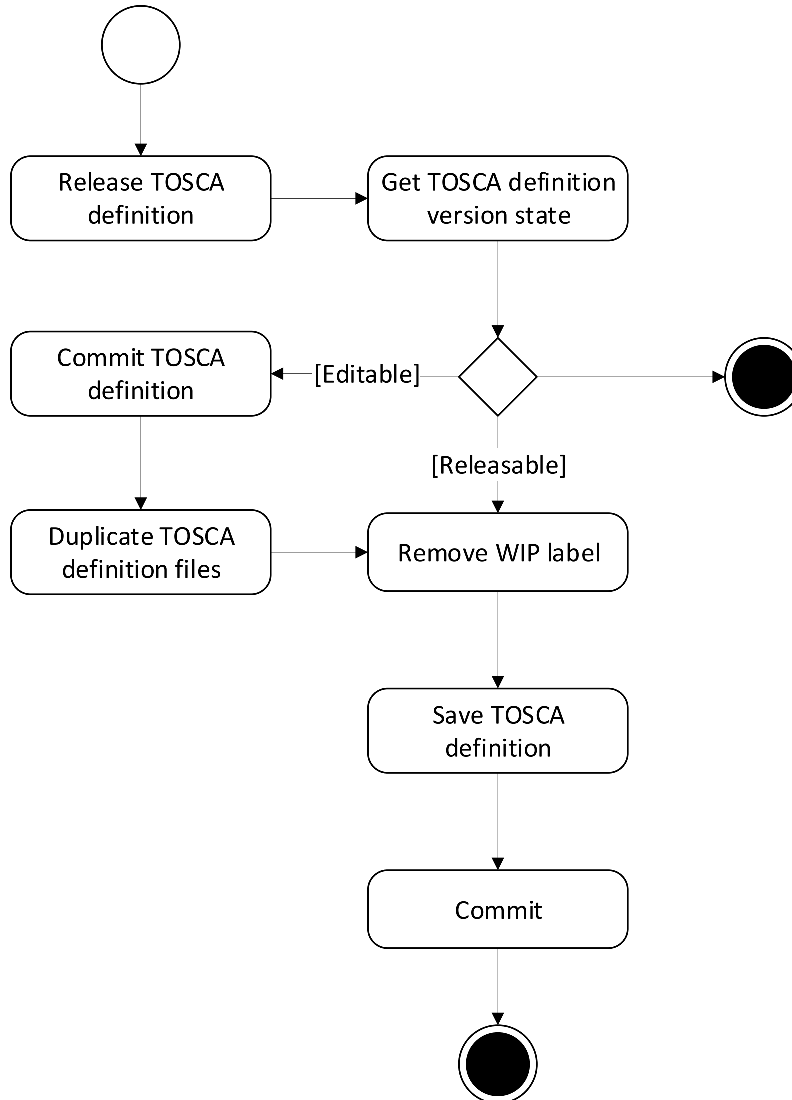

Figure 3.8: UML activity diagram to release a TOSCA definition

4 An Approach for Differences between TOSCA Definitions

A key element extending a versioning approach is the calculation, representation, and visualization of differences between two TOSCA definitions. If there are multiple versions available for a definition, it is only a matter of time until the advancement of a specific version needs to be reconstructed. Therefore, it is necessary to provide an approach which is capable of presenting differences to a human user in multiple forms as argued in the following. To support different kinds of visualizations a generic representation must be calculated and provided for further processing.

In this chapter, requirements (Section 4.1) and concepts for calculating differences between two TOSCA definitions are presented (Section 4.2). A TOSCA definition hereby refers to all element types defined in TOSCA (*Service Templates*, *Node Types*, *Relationship Types*, etc.). Further, solutions for presenting the differences to a human user are presented in a graphical way (Section 4.3), as well as in a generic, textual way (Section 4.4).

Similar to the tables shown in Chapter 3, this section also uses tables to summarize considered alternatives as proposed by Kopp et al. [KAZ18].

In Chapter 3, the example was extended to contain version identifiers in their ids. This chapter extends the example by introducing a new version of the service's topology. The Service Template `WebshopService_-w2-wip1` shown in Figure 4.1 employs new versions of the Webshop and the Java Node Types. Further, the `Ubuntu_16.04-w3` Node Template was updated to use 32GB of RAM instead of 16. The previous version `WebshopService_-w1` is displayed in Figure 3.1

4.1 Requirements

Users usually prefer graphics over text to gain an overview but prefer textual representations, if they are interested in details [Pet95]. Therefore, both methods should be supported by a visualization approach to represent differences in TOSCA definitions. A Topology Template inside a Service Template can be rendered as a graph [KBBL13]. Building atop of already existing technologies provided by Winery, a graphically visualization should outline differences between two versions of a Topology Template, whereas differences should also be visualized in a textual form to present differences between two versions of any TOSCA definition type.

Requirement 1 (DR1): Visualization of changes inside a topology. A similar approach as presented by Ohst et al. [OWK03] (see Section 2.5.3) should be applied where differences in a graph can be displayed by an extended graph including changed, added and removed elements.

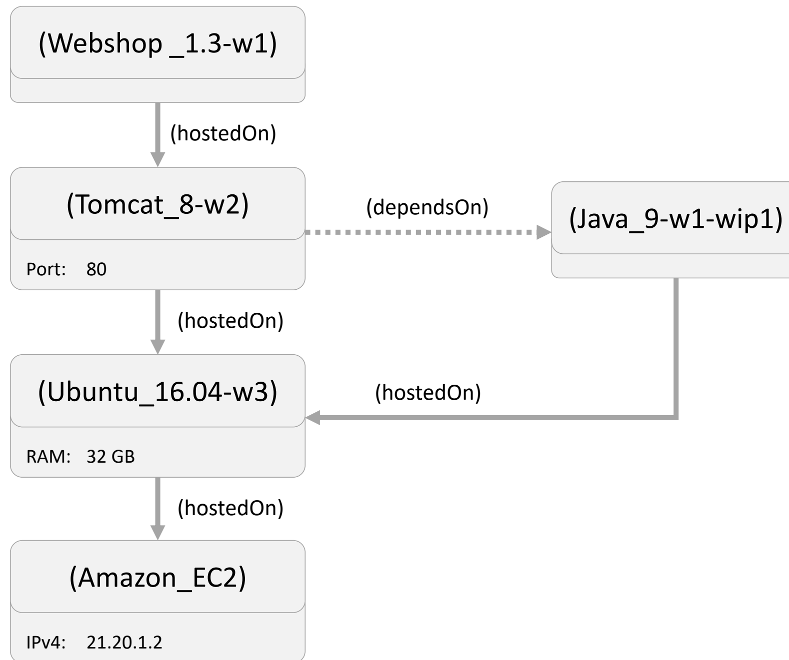

Figure 4.1: Second version of the example topology.

Requirement 2 (DR2): Difference visualization for all definitions. To render differences in a textual form, a similar approach as described by Lacan [Lac17] should be employed because it groups added, changed, and removed elements.

Requirement 3 (DR3): Generic representation. A generic representation of the changes between two versions of a TOSCA definition is required to enable further processing into the graphical and textual visualizations. Additionally, it increases the extendability, for example, to add more visualization approaches.

4.2 Determining Differences

As described in Section 2.5, the calculation of differences between two models consists of two phases: matching and determining differences. This section describes approaches to match (Section 4.2.1), and determine differences between TOSCA definitions (Section 4.2.2).

4.2.1 Matching TOSCA Definitions

In Section 2.5 it was outlined that the matching of model elements can be difficult and time consuming unless an id- or similarity-based algorithm is used (cf. Section 2.5). In TOSCA each definition can be identified by its QName. For example, the id of the Webshop is `{http://example.org/tosca/versioning/nodeTypes}Webshop_1.2-w1`. However, matching a TOSCA definition itself is not required because comparing a Node Type with a Service Template for example, does not make sense. In the following, only the matching of versions of the same TOSCA

definition is discussed. For instance, it is only allowed to compare two versions of the Webshop Node Type. Comparing a Tomcat Node Type definition with a Webshop Node Type definition may also work, but it is not explicitly discussed in this thesis.

To match elements of a TOSCA definition ids are used. However, because not every element defines an id, the concept of a primary key is introduced. It identifies an element by different means. If two elements define the same primary key, the elements are considered a match. If ids are available, the primary key is equal to the id. In case that no id attribute is available, the name is used for identifying the elements. For example, the `Amazon_EC2` Node Type defines a life cycle interface with two operations `startUp` and `shutdown`. As interfaces and operations only define a name attribute, they are considered matched, if their names are equal.

In other cases, if an element is allowed only once or only contains one attribute, it is considered matched if the same element or attribute respectively is present in both models. Consequently, the element or attribute itself is its primary key. For example, an Node Type's instance state only declares a string defining the state. A state in two Node Type definitions is considered a match, if both models define an instance state with the same string. Hence, there are no changed states. They are either matched, or they were added or removed respectively.

However, there are exceptions where neither an id, nor a name can be used as the primary key. These elements are Property Definitions, Property Constraints, Property Mappings, and Implementation Artifacts (IAs). Each Property Definition defines the attributes `element` and `type`. Its primary key is the `element` attribute since its type may change over time. A Property Constraint defines a `property` as well as a `constraintType`. The primary key of Property Constraints is the combination of both attributes because one property may be subject of multiple constraints. Property Mappings inside a Service Template are identified by its three attributes `serviceTemplatePropertyRef`, `targetOjectRef`, and `targetPropertyRef` because it is possible to define a mapping of a Service Template's property to multiple properties of a Node Template or Relationship Template. Lastly, Implementation Artifacts are strictly bound to an interface or operation. Therefore, the primary key of IAs is represented by the the combination of the `interfaceName` and `operationName`.

Finally, a fall back method uses all attributes defined in an element for building its primary key. Table 4.1 summarizes the elements and their resulting primary keys.

Element	**Primary Key**
Defines an id attribute	`id` attribute
Defines a name attribute	`name` attribute
Defines only one attribute	The defined attribute
Property Definition	`element` attribute
Property Constraints	`property` and `constraintType` attributes
Property Mapping	`serviceTemplatePropertyRef`, `targetOjectRef`, and `targetPropertyRef` attributes
Implementation Artifact (IA)	`interfaceName` and `operationName` attributes
None of the above	All attributes defined inside the element

Table 4.1: Primary keys to identify elements inside TOSCA definitions. The primary key column contains the attributes building the primary key to identify the element.

4.2.2 Calculating Differences

Since the matching phase traverses both model trees to find matching elements, added and removed ones can be collected simultaneously. If no matching element is found in the base model M_{base} (old version) for an element in the working model $M_{working}$ (new version), the element was added in $M_{working}$. Equivalently, if no match is found for an element in $M_{working}$ which is present in M_{base}, it has been removed (see also Section 2.5).

If a match between both models (M_{base} and $M_{working}$) was found, it is further investigated to detect changes in its properties. If a property of an element was added, changed or removed, the element itself and all of its parents are flagged changed. This is performed recursively until all nested elements are processed.

Similar to added and removed elements, the phase of determining changes between two elements can be combined with the matching operation. Therefore, instead of matching the models first and calculate the changes between them afterwards, the whole change detection can be one optimized algorithm as it is described in Algorithm 4.1.

The function shown in Algorithm 4.1 gets two elements (`oldElement` and `newElement`) as its input. For example two Node Type definitions. The first statement defines a `hasChanges` boolean variable to indicate, whether the currently processed element was changed. If both elements are null or define an empty set, the function ends immediately. Otherwise, the differences are calculated.

If the `oldElement` is null or an empty set, the `newElement` was added. Hence it gets added to the list of a added elements and a change was found resulting in a true value for the `hasChanges` variable. Similarly, if the `newElement` is empty, it has been removed. Therefore, it is added to the removed elements list and the `hasChanges` flag is set to true.

To determine whether any of the elements has children, the method `hasChildren` is called. If this is the case, the primary key is retrieved from both elements. If they are equal, a match is found and all their child elements are investigated for changes. The `children` function hereby collects all children defined in both elements and returns it as a list of tuples containing the child elements as tuples. If there are changes in any of their children, the element is considered changed.

Otherwise, if the elements do not have children, they are directly compared with each other. If they are not equal, they must have been changed. Finally, if the `hasChanges` variable is set to true, both elements are added to the changed elements list and the `hasChanges` is returned.

4.2.3 Representing Differences

The result of a difference calculation is the representation. It must store all necessary information to create human readable representations, commonly referred to as visualizations [BP08] (see also Section 2.5). For later processing, the representation of model differences in a directed delta includes the following elements:

- the path to each changed element for later access,
- the element's state (added, changed or removed), and
- in the case of changed attributes their old and new values.

Algorithm 4.1 Difference algorithm

```
function detectDifferences(oldElement, newElement)
    hasChanges ← false
    if oldElement ≠ ∅ and newElement ≠ ∅ then
        if oldElement = ∅ then
            addElementToAddedList(newElement)
            hasChanges ← true
        else if newElement = ∅ then
            addElementToRemovedList(oldElement)
            hasChanges ← true
        else
            if hasChildren(oldElement, newElement) and
              getPrimaryKey(oldElement) = getPrimaryKey(newElement) then
                for all (oldChild, newChild) ∈ children(oldElement, newElement) do
                    hasChanges ← hasChanges ∨ detectDifferences(oldChild, newChild)
                end for
            else
                if oldElement ≠ newElement then
                    hasChanges ← true
                end if
            end if
            if hasChanges then
                addElmentToChangedList(oldElement, newElement)
            end if
        end if
    end if
    return hasChanges
end function
```

To represent changes combined with unchanged elements in the graph, a symmetric delta is required (see Section 2.5.2). A symmetric delta contains all elements of the graph, including added and removed ones. Therefore, a fourth element state "unchagend" must be introduced to describe matched but unchanged elements.

4.3 Topology Difference Visualization

To visualize the changes graphically in a Topology Template, a similar approach as proposed by Ohst et al. [OWK03] is employed (see also Section 2.5.3). They use colors to encode added, changed and removed elements in a model. Similarly, this approach uses a red color to display removed nodes and relationships, yellow indicates changes were made to an element, whereas a green color represents added Node and Relationship Templates.

In addition to the color codes, annotations should be used to clearly signal the performed operations. The reason is that screens may have different brightness, and colors are displayed differently. Further, if the user is colorblind, the annotations help to interpret the graph correctly.

As an example, the two versions of the Service Template defined in the running example are compared. The old version (`WebshopService_-w1`) is described in Figure 3.1, whereas the new version's topology (`WebshopService_-w2-wip1`) is shown in Figure 4.1.

The changes performed between `WebshopService_-w1` and `WebshopService_-w2-wip1` are described in Table 4.2. Between the two versions, the Webshop was updated to version `1.3-w1`, the Java runtime was upgraded to version 9, and the operating system's random access memory (RAM) was changed from 16GB to 32GB.

Node	**Changes**
Webshop	Updated version from `1.2-w2` to `1.3-w1`
Tomcat	–
Java	Updated version from `8-w2` to `9-w1-wip1`
Ubuntu	Changed RAM property from 16GB to 32GB and added a DA
Amazon EC2	–

Table 4.2: Changes in the topology between the Service Templates `WebshopService_-w1` and `WebshopService_-w2-wip1`.

By comparing these two topologies, a result similar to the one shown in Figure 4.2 should be generated. According to the procedure described above, removed Node Templates and Relationship Templates are marked red, added elements green, and changed ones yellow. Further, rather than displaying nodes and relationships in the "unchanged" state with their color defined by their type, all unchanged elements must be painted in an neutral color to avoid confusions (grey in this case).

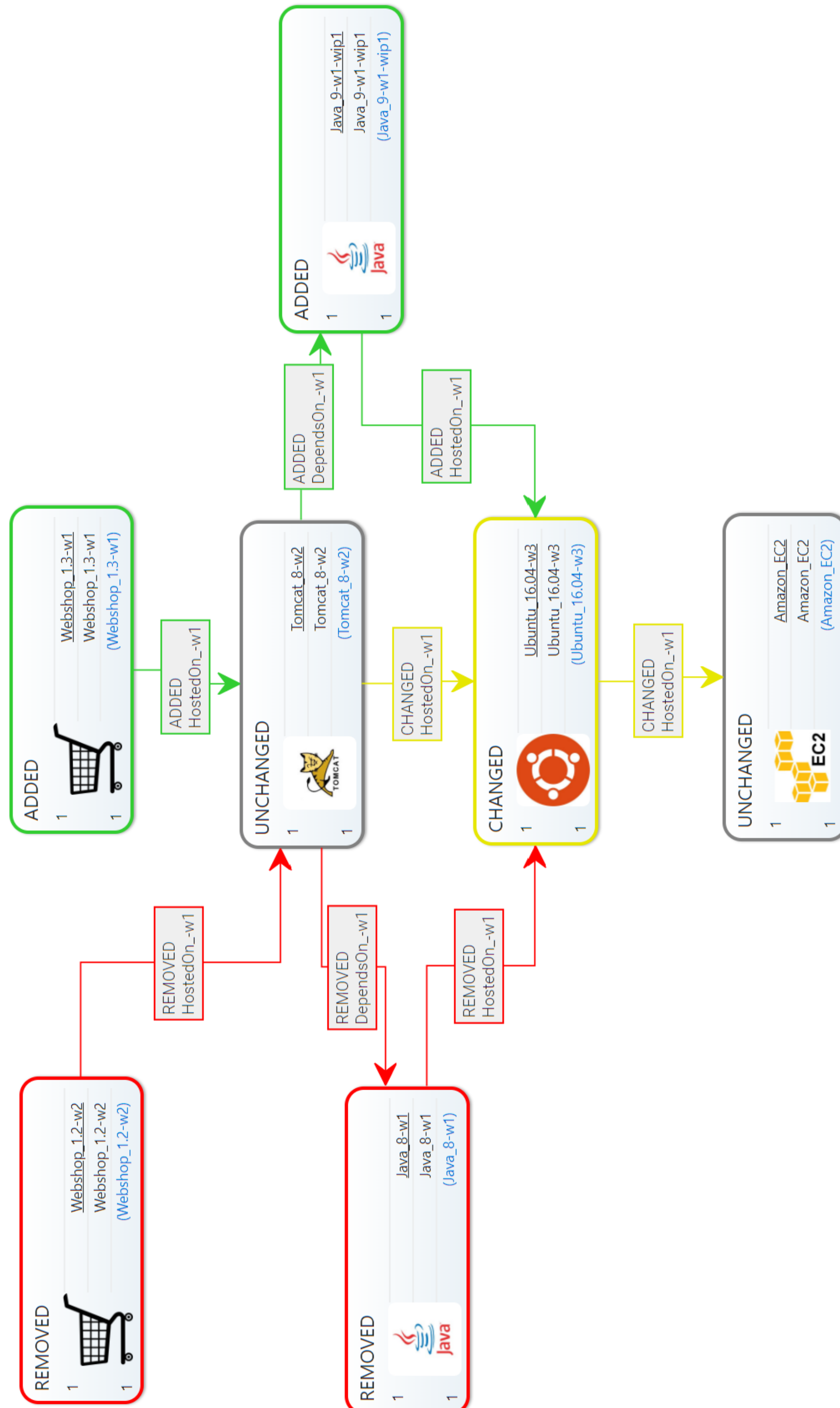

Figure 4.2: Visualization of changes in a topology template
Visualization of changes made in the topology template between `WebshopService_-w1` and `WebshopService_-w2-wip1`

4.4 Textual Difference Visualization

In addition to the visualization of differences in a topology, all other changes must also be presented to a user (DR2). Because a textual visualization is more detailed than a graphical one (in this particular scenario) and TOSCA definitions usually do not have a graphical representation, differences should be visualized in a text-based way.

Therefore, a textual difference visualization should group the elements according to their states in a directed delta into the added, changed, and removed group as required by DR2. Inside a group, each element is stating its name, parent elements up to the root, and if it is in the changed state, its old and new value. For example, if the name of the `Java_8-w1` Node Template inside a Topology Template defined in a Service Template was changed from `Java 8` to `Java 8 Runtime`, the resulting text appearing in the textual difference visualization is `topologyTemplate/nodeTemplates/Java_8-w1/name changed from "Java 8"to "Java 8 Runtime"`.

Continuing the running example which was detailed in Section 4.3, the resulting text difference is shown in Listing 4.1. According to DR2, the text uses the changelog format proposed by Lacan [Lac17] (see Section 2.5.2. In contrast to the changelog by [Lac17], the changelog to represent changes in TOSCA contains only the changes between two given versions shown in line 3. Afterwards, the added, changed and removed elements are listed in their respective blocks.

The added block (lines 5 to 11) describes the added Node Templates `Webshop_1.3-w1` and `Java_9-w1-wip1` in lines 6 and 7. Line 8 describes a change which is not visible in Figure 4.2. While the Ubuntu Node Template is only shown changed in the topology difference, the added Deployment Artifact appears in the added list. Additionally, three new Relationship Templates were added: the new Webshop (line 9) and Java (line 10) versions are in a relation with the Tomcat web-server, as well as the dependencies between the new Java version and the Ubuntu operating system (line 11).

Similarly, all changed elements are listed in the changed block (lines 13 to 19). Both, the id (lines 14 and 15) and the name (lines 16 and 17) of the Service Template changed from `WebshopService_1.2-w1` to `WebshopService_1.2-w2-wip1`. Lastly, the changed RAM property defined in the Ubuntu Node Template is shown in lines 18 and 19.

The last block (lines 21 to 26) lists all elements which have been removed in the Service Template from `WebshopService_1.2-w1` to `WebshopService_1.2-w2-wip1`. A Relationship Template is deleted, if a Node Template is deleted that it points to. Hence, since the `Webshop_1.2-w2` (line 22) and `Java_8-w2` (line 23) Node Templates were deleted, the Relationship Templates describing their connections were also deleted (lines 23 to 26).

Listing 4.1 Textual difference visualization example

```
# Changelog

## Changes from version 1.2-w1 to 1.2-w2-wip1

### Added
- topologyTemplate/nodeTemplates/Webshop_1.3-w1
- topologyTemplate/nodeTemplates/Java_9-w1-wip1
- topologyTempltae/nodeTemplates/Ubuntu_16.04-w3/deploymentArtifacts/ubuntuInstallable
- topologyTemplate/relationshipTemplates/Webshop_1.3-w1_HostedOn_-w1_Tomcat_8-w2
- topologyTemplate/relationshipTemplates/Java_9-w1-wip1_HostedOn_-w1_Ubuntu_16.04-w3
- topologyTemplate/relationshipTemplates/Tomcat_8-w2_DependsOn_-w1_Java_9-w1-wip1

### Changed
- id
  changed from "WebshopService_1.2-w1" to "WebshopService_1.2-w2-wip1"
- name
  changed from "WebshopService_1.2-w1" to "WebshopService_1.2-w2-wip1"
- topologyTemplate/nodeTemplates/Ubuntu_16.04-w3/properties/KVProperties/{RAM}
  changed from "16GB" to "32GB"

### Removed
- topologyTemplate/nodeTemplates/Webshop_1.2-w2
- topologyTemplate/nodeTemplates/Java_8-w2
- topologyTemplate/relationshipTemplates/Webshop_1.2-w2_HostedOn_-w1_Tomcat_8-w2
- topologyTemplate/relationshipTemplates/Tomcat_8-w2_DependsOn_-w1_Java_8-w2
- topologyTemplate/relationshipTemplates/Java_8-w2_HostedOn_-w1_Ubuntu_16.04-w3
```

5 Prototype and Validation

In this chapter, a proof of concept implementation of the Versioning (Chapter 3) and Differencing (Chapter 4) approaches is explained (Section 5.1) and validated against the requirements defined for both approaches (Section 5.2).

5.1 Extension to Eclipse Winery

The versioning approach presented in Chapter 3 along with the differencing approach described in Chapter 4 were implemented as an extension to Eclipse Winery (see Section 2.2).[1] The main contribution was made in the web-based front-end to fully support the versioning and differencing approaches with respect to the corresponding requirements outlined in Sections 3.1 and 4.1. Major advancements in the current implementation have been the possibility to correctly manage and display versioned definitions (see Sections 3.2, 3.5 and 3.6) as well as already existing ones which do not follow the naming convention (see Section 3.3), the enforcement of the definition's editability state (Section 3.4), the graphical visualization of differences in topologies (Section 4.3), as well as in a text-based form (Section 4.4). Further, the business functionality was extended to support the management of versions, including the calculation and representation of differences among two TOSCA definitions (see Section 4.2).

5.1.1 Back-End Extension: Version Identifiers

Extensions in the back-end were developed according to the Test Driven Development (TDD) paradigm [JS05]. The benefit of writing tests upfront helped to increase the stability of newly developed code and ensured that applied changes are resulting in the expected outcome. Simultaneously, they ensure that other functionalities were not affected by the currently performed changes.

To support version identifiers in the id field of a TOSCA definition a utility class providing the required functionality, for example to get a version from a given id or the name without its version, was developed. It also implements the functionality to calculate the differences between two given model elements (`TExtensibleElements`). The actual calculation of differences is performed by a specialized open-source library called *Java-Object-Diff*[2]. Because the difference representation provided by Java-Object-Diff does not reference the old and new values of changed attributes,

[1]The implementation is available via GitHub in the OpenTOSCA fork of the Eclipse Winery repository: https://github.com/OpenTOSCA/winery/pull/59

[2]Java-Object-Diff is available via GitHub under https://github.com/SQiShER/java-object-diff

a custom representation was added in the form of the *ToscaDiff* class. It not only provides the functionality to convert Java-Object-Diff's difference representation, but also the generation of the textual visualization.

Combined with a class to represent the version of a TOSCA definition, both classes introduced above are shown in an UML class diagram in Figure 5.1. On the right hand side of Figure 5.1, already existing classes are shown for reference. Each subclass of the `DefinitionsChildId` class represents the id (and name) of a specific TOSCA definition. Only the three most common elements are shown here for readability reasons, but all other definitions are indicated by the three dots. Similarly, the actual model elements are represented by subclasses of `TExtensibleElements`. At the bottom of the diagram, an enumeration is shown which is used to indicate the state of an difference element. All elements were added to the `org.eclipse.winery.common` module.

5.1.2 Back-End Extension: Handling Commits

As described in Section 3.4, there must be a way to perform commits to indicate that a definition cannot be changed anymore. There are multiple ways how this can be achieved.

Winery saves all TOSCA definitions separately in a well-defined directory structure (see Section 2.2). Based on this structure where each TOSCA definition has its own directory, it is easy to do commits, if the repository is set up using tools like Git (Section 2.4). Since the Winery already uses a Git-based repository, creating commits and determining whether a TOSCA definition has been the subject of a commit is done by using the JGit[3] library.

5.1.3 Back-End Extension: Find Referencing Definitions

Section 3.5 outlines that it should also be possible to update other TOSCA definitions which are referencing the current definition. Therefore, all referencing TOSCA definitions must be found first (Section 3.5.1). Figure 2.2 shows the dependencies between all elements defined in TOSCA. To collect all TOSCA definitions referencing, for example, a given Node Type, all *Node Type Implementations*, *Relationship Types*, *Service Templates*, *Policy Types*, and *Node Types* must be collected and inspected whether they are referencing the given definition.

The implementation in Eclipse Winery offers methods for all eleven TOSCA definitions where all referencing TOSCA definitions are collected and returned in a filtered list containing only those TOSCA definitions representing the latest management version (see Section 3.5.1). It further provides the possibility to find definitions referencing imported XML artifacts which can be used to define property definitions as an extension to TOSCA. Property definitions can be declared in *Node Types*, *Relationship Types*, *Artifact Types*, *Capability Types*, *Requirement Types*, and *Policy Types*. Similar to the collection providing the referencing TOSCA definitions, each of these definitions is returned in the collection, if the given import is referenced by it.

To collect all TOSCA definitions referencing a given TOSCA definition, all available TOSCA definitions have to be investigated. First, they are filtered to only contain TOSCA definitions representing the management versions as described in Section 3.5.1. Next, all remaining TOSCA

[3] `https://www.eclipse.org/jgit`

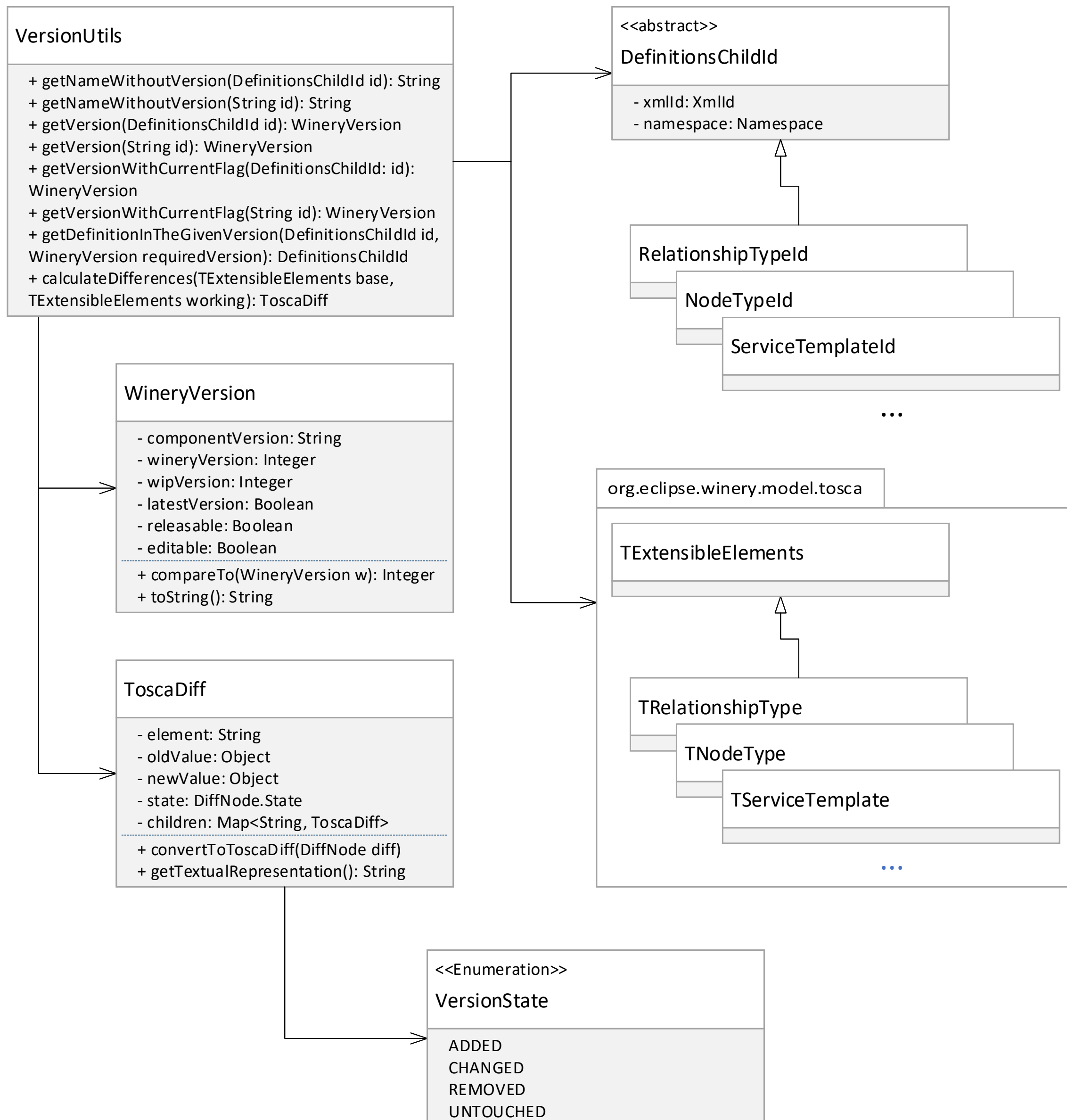

Figure 5.1: Simplified class diagram showing the functionality of the new `VersionUtils`.

definitions are processed by scanning through all of its elements for references. If a reference points towards the given TOSCA definition, the respective TOSCA definition is added to the referencing list. For example, to find referencing TOSCA definitions of the `Java_9-w1-wip1` Node Type, all TOSCA definitions defined in the repository must be loaded. Afterwards, all TOSCA definitions are removed which do not fulfill the requirement of a latest management version. All references the remaining elements define are investigated whether they point to the `Java_9-w1-wip1` Node Type. In this case, the list of referencing TOSCA definitions contains the `WebserviceService_-w2` Service Template, as well as the `Java-9-Implementation_-w1`.

5.1.4 Front-End Extension: Overview

In the following, adjustments made in the User Interface (UI) are described and explained. Figure 5.2 shows all Node Types which are defined in a repository. An orange border around an item indicates, that there are multiple versions available for a TOSCA definition. In Figure 5.2, the lower four Node Types (`Java`, `Tomcat`, `Ubuntu`, and `Webshop`) are surrounded by an orange border, whereas the border of the first Node Type (`Amazon_EC2`) is not colored, indicating that there is only one version available for this TOSCA definitions.

Accordingly, Figure 5.3 shows available versions for the `Java` Node Type in a tree-like structure. Each winery version highlighted with #2 in Figure 5.3 (indicated by the `w1`, see Section 3.2) is listed underneath the common name (#1) and every WIP version along with the management release is shown as their children (#3). Further, it is possible to display the differences among two consecutive versions which is visualized textually (#5) (see Section 4.4) by expanding the "`+ Differences`" (#4) element between them.

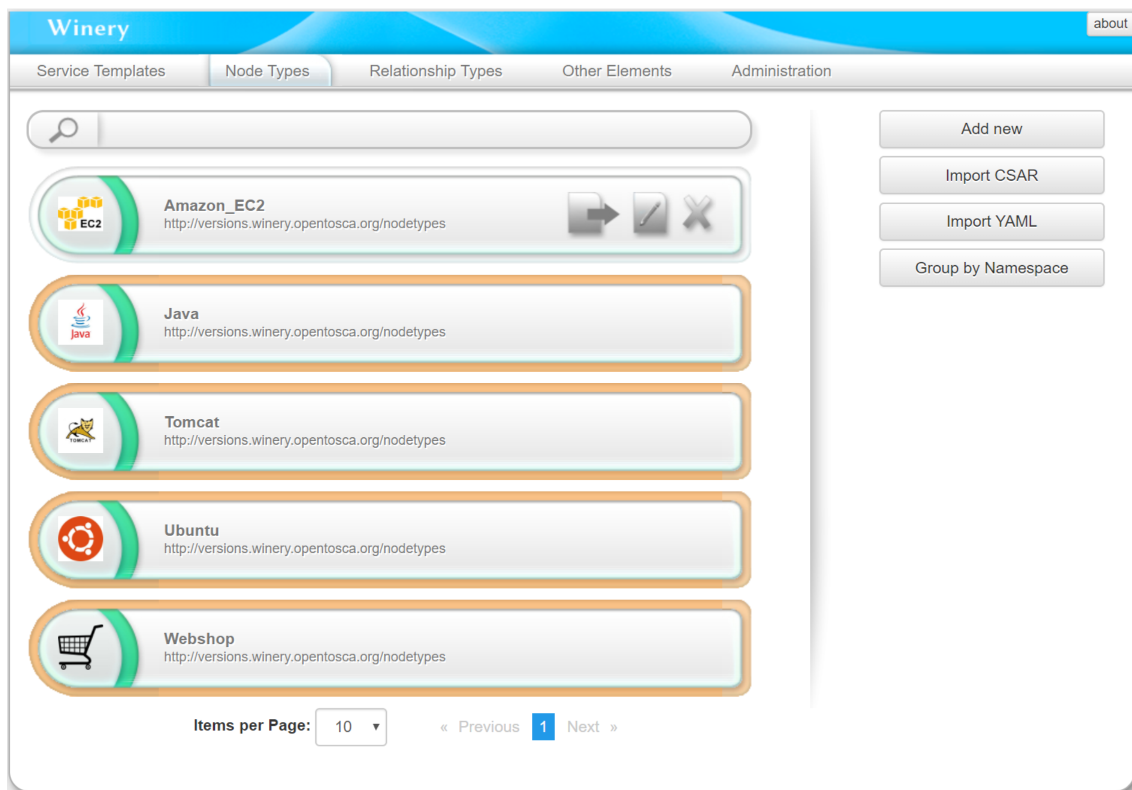

Figure 5.2: Overview over all Node Types defined in the current repository

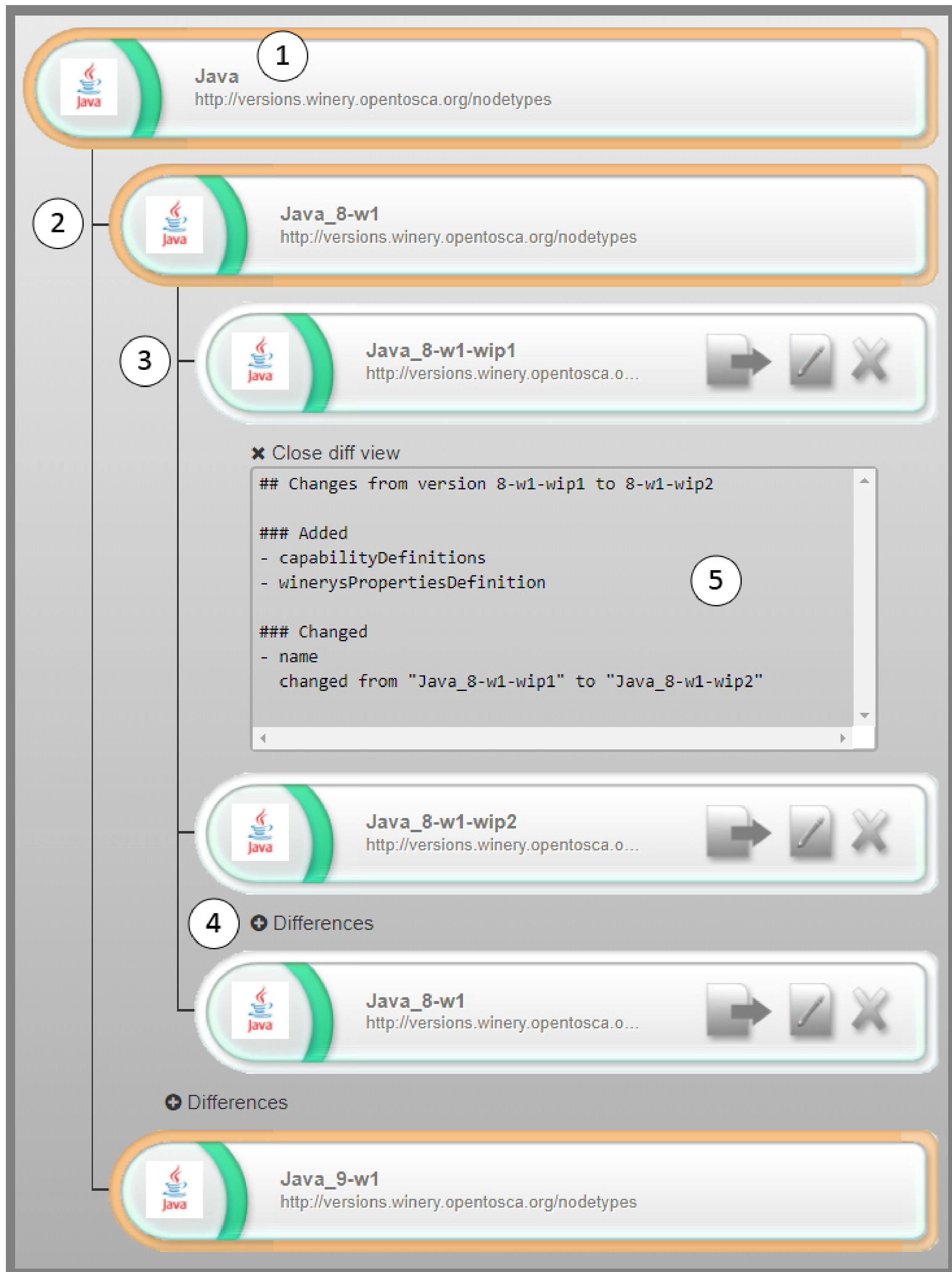

Figure 5.3: Overview of multiple versions of a Node Type

5.1.5 Front-End Extension: Detailed View

After a version has been selected from the overview in Figure 5.3, the user is directed to a detailed view of the selected definition. Figure 5.4 shows the version `9-w1-wip1` of the `Java` Node Type. At the top, two information banner indicate, that the currently selected version was already committed (expressed by the yellow warning banner), and that there is a newer version available for this TOSCA definition (indicated by the blue information banner). On the right side, a version drop-down displays all versions of this TOSCA definition and available operations. In this case, “Add a new version” is the only option available since the TOSCA definition is in the committed TOSCA definition version state (see Section 3.4).

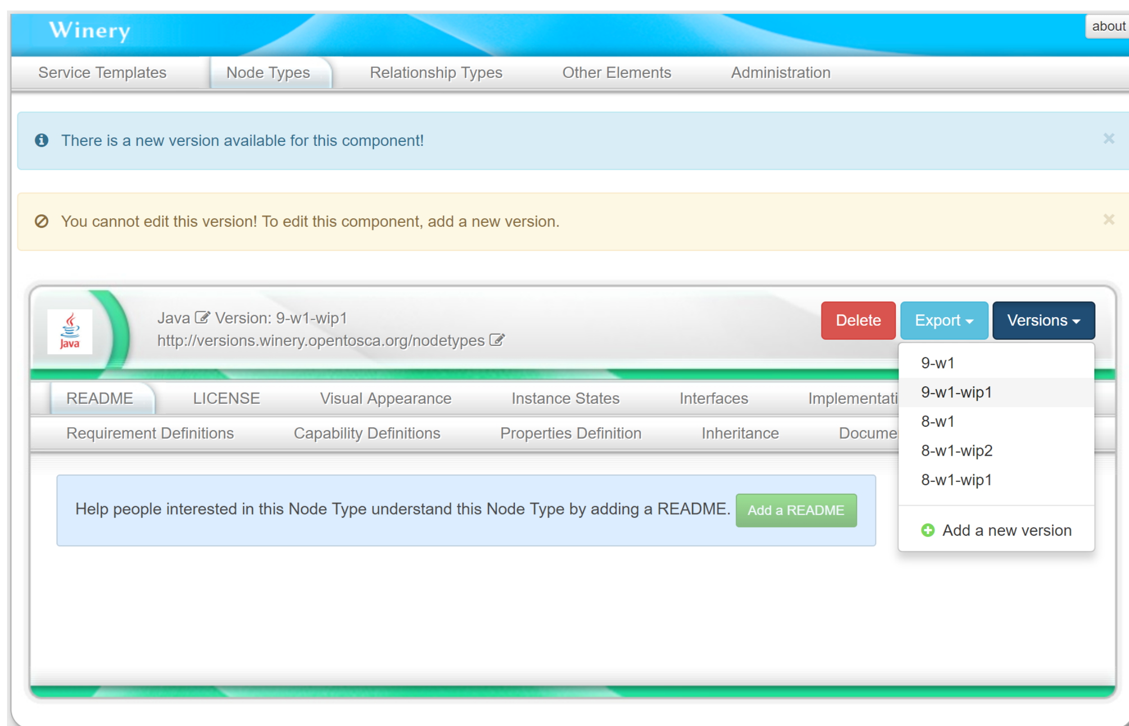

Figure 5.4: View of a single TOSCA definition

To add a new management version to the definition shown in Figure 5.4, a pop-up appears (see Figure 5.5) where the desired version can be added, after the “Add a new version” element has been clicked. Once a specific version has been selected, the new name of the definition is displayed at the pop-up’s bottom. Finally, after the new version has been added, the user gets redirected to it as shown in Figure 5.6.

Since this `Java_9-w2-wip1` has just been created, the TOSCA definition is in the editable TOSCA definition version state and changes can be applied. Further, the version drop-down offers more options than in Figure 5.4. In contrast to `Jav_9-w1-wip1` which already is in the committed state, `Java_9-w2-wip1` can be frozen to prevent it from further changes, or released as the new stable winery version of `Java_9` (see Section 3.6).

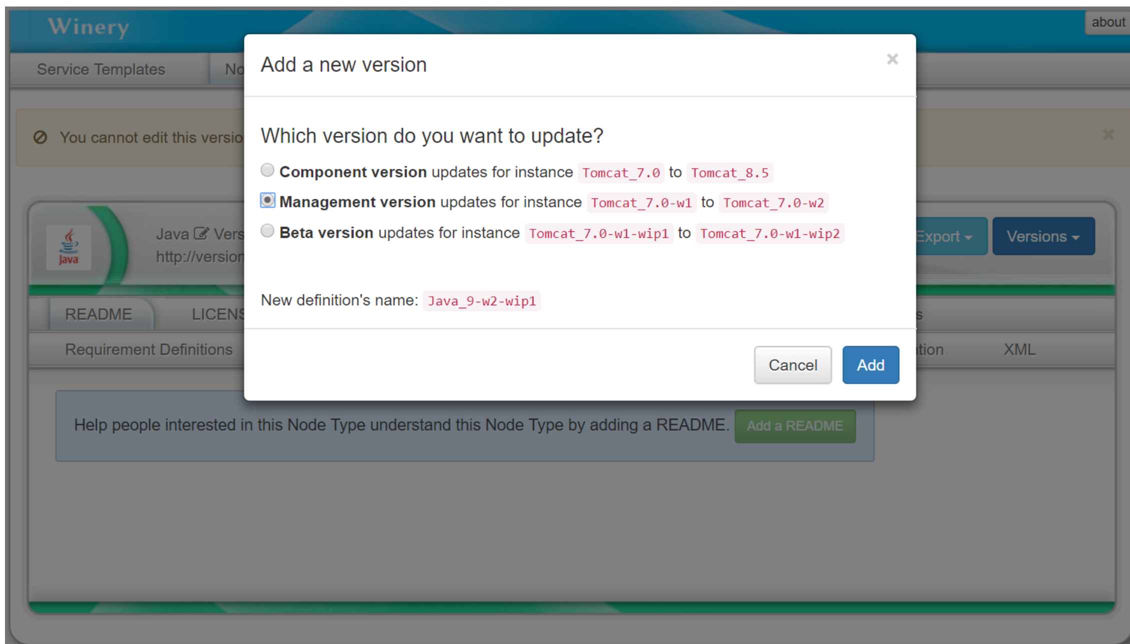

Figure 5.5: Dialog to add a new version

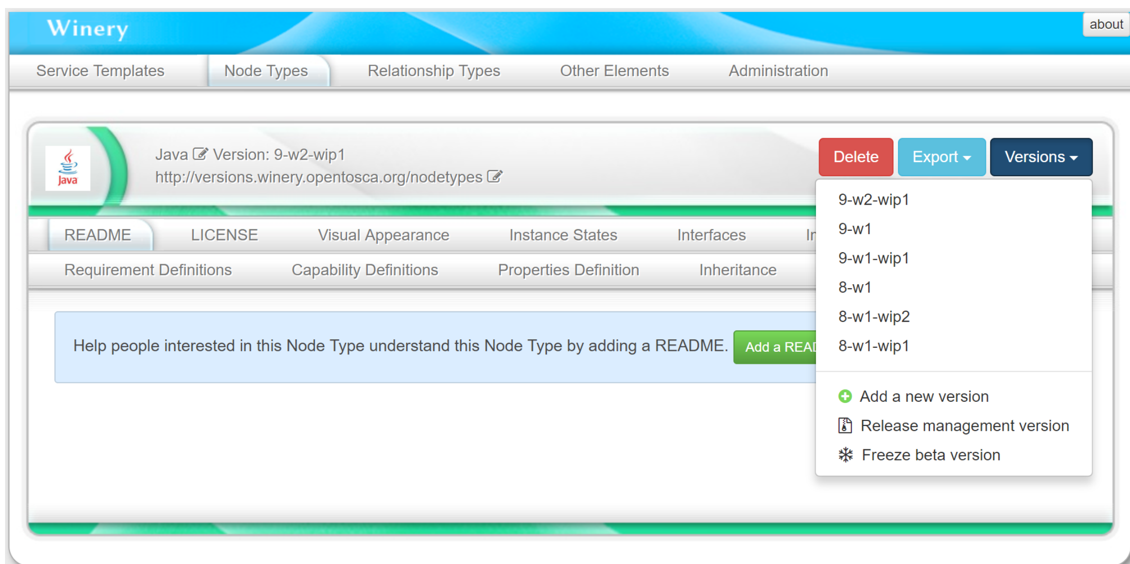

Figure 5.6: View of a editable definition version

5.1.6 Front-End Extension: Topology Differences

Differences between two service topologies can be visualized graphically in the Topology Modeler as shown before in Section 4.4. Since the `ToscaDiff` object represents a directed delta from one version to another, both topologies have to be merged first. This is implemented in the `TopologyRenderComponent` inside the Topology Modeler. It iterates over all changes and adds removed elements to the loaded model and flags them accordingly. Similarly, all other elements

(Node and Relationship Templates) get their corresponding version state assigned. Afterwards the merged model's layout is optimized to avoid overlaps between nodes and relationships. Finally, it is painted by the Topology Modeler's canvas component which uses a JavaScrip library called *jsPlumb*[4] to render the topology as a graph.

5.2 Validation

This section validates the implementation (see Section 5.1 of both approaches against their requirements defined in Sections 3.1 and 4.1 (Section 5.2.1), and describes the applicability of the implementation by solving a use case (Section 5.2.2).

5.2.1 Requirement Validation

In Section 3.1 and Section 4.1, requirements were defined which must be satisfied by the proposed versioning, as well as the difference representation and visualization approaches. In the following, both are validated against their respective requirements. The results are summarized in Table 5.1, whereas the number in front of each requirement indicates the respective chapter it is defined in along with its id. While two check-marks represents a fully satisfied requirement, a single check-mark hereby stands for a satisfied element with room for improvements.

First, the versioning approach is examined. Since the approach does not interfere with any of elements defined in the TOSCA specification [OAS13b], it is completely standard conform an satisfies requirement VR1. However, since it is standard conform and does not introduce any new elements or properties, the version identifier is somehow "hidden" inside the name. If an extra element or property would have been introduced, a version would be declared more clearly. However, VR2 is satisfied because the version identifier is appended at the id of a TOSCA definition.

As outlined in Section 3.1, three version identifiers must be supported to define 1.) a component version (VR3), 2.) a winery version (VR4), and 3.) a WIP version (VR5). The approach defines identifiers of three different kinds: `componentVersion`, `wineryVersion`, and `work in progressVersion`. A component version allows an user to optionally specify external versions for a TOSCA definition, for example to reference a MySQL database in a specific version. Following a component version, the winery, or management version in general, defines the version of a given definition. If bug fixes or other advancements are added to it, the management version is incremented. Further, to distinguish stable versions form unstable ones, a third version identifier was introduced: the beta, or WIP version. Hence, all three requirements regarding the version identifier VR3, VR4, and VR5 are fully satisfied.

In Section 3.3, it is described how definitions without version identifiers must be handled. They are always considered to be not editable but should be displayed to the users. If fixes have to be applied, a version identifier is appended to its name (and id) which satisfies requirement VR6. Therefore, the implementation of the approach must ensure, that non-editable definitions cannot be changed. The

[4]The jsPlumb framework is available under `https://jsplumbtoolkit.com/`.

Requirement	Satisfied
VR1 (“Conformity to the TOSCA specification”)	✓✓
VR2 (“Detectable version identifier inside the definition”)	✓
VR3 (“Support component versions”)	✓✓
VR4 (“Support a version identifier”)	✓✓
VR5 (“Support work-in-progress versions”)	✓✓
VR6 (“Support existing definitions”)	✓✓
VR7 (“Committed and released versions must not be changed”)	✓
DR1 (“Visualization of changes inside a topology”)	✓✓
DR2 (“Difference visualization for all definitions”)	✓✓
DR3 (“Generic representation”)	✓✓

Table 5.1: Requirement validation using check-marks. Two check-marks indicate fully satisfied requirements, whereas one check-mark refers to a mainly satisfied requirement.

Winery UI ensures that only TOSCA definitions in the editable version state are editable. However, since Winery directly saves all TOSCA definitions in XML files, it cannot be assured that a user is editing these TOSCA definition files directly. Therefore, requirement VR7 is “mainly satisfied”.

All requirements applying to a difference calculation and visualization approach were defined in Section 4.1. The visualization displayed in Figures 5.9 and 5.10 shows the differences between two topology templates satisfying requirement DR1. Similarly, differences between two TOSCA definitions are shown in textual form in Figure 5.3 (DR2). Since both visualizations are calculated using the representation introduced in Section 5.1, it satisfies the requirement of a generic difference representation (DR3).

5.2.2 Use Case Validation

As an use case to version the components of a cloud application, the project Digital Romansh Chrestomathy (DRC)[5] is used. It took place from 2009 to 2011 at the University of Cologne. During this project an application was developed which provides a tool for collaboratively digitalize and correct the most important work describing the romansh language: “Rätoromanischen Chrestomathie” by Caspar Decurtins. The DRC project ended in 2011 but the developed application is still in use. However, during the last seven years, software updates had to be performed, for example, for security reasons. Many of them required further changes as versions of the employed frameworks were incompatible with each other. Due to the fact that the application is still in use, it is required to always maintain a runnable service version. Further, after a new service version is successfully developed, the differences between both should be outlined. The approach presented in this thesis is capable of resolving these issues.

[5] http://www.crestomazia.ch

Component	First Version	Current Version
Eclipse	3.7	Neon
Java	6	8
Ubuntu	16.04	16.04
DRC	b2	b7
DRC-Portal	b2	b7
eXist-DB	1.4.2	2.2
Java	6	8
Debian	7	9.3

Table 5.2: Component versions of the DRC's topology.

The topology describing the DRC's application stack in the 2011 version is shown in Figure 5.7. It can be divided into a client side shown at the bottom of Figure 5.7, and a server side illustrated at the top. While the client side only requires a Java-based Eclipse IDE running on an operating system (Ubuntu in this case), the server is more complex. It also requires a Java runtime hosted on a Debian operating system. The actual DRC software is split into two parts: 1.) the back-end service (`DRC_b2-w1`) which is used by the Eclipse IDE, and 2.) a web service `DRC-Protal_b2-w1` giving access to the data via a web browser. All data is stored and retrieved by both DRC components by connecting to an eXist database.

Over the last seven years the DRC's topology structure stayed the same. However, the versions of the components employed in the application stack have changed significantly. Figure 5.8 shows the DRC's current topology in version 2018. The versions of all Node Templates of both, 2011 and 2018, topologies are listed in Table 5.2.

The proposed versioning approach ensures that a running application stack is always available since a new version must be created in order to change the service's topology. Hence, it is possible to create multiple versions of one service whereas changes to already committed or released versions are prohibited, while it is ensured that a service can always be deployed. Simultaneously, it is also possible to calculate the differences between two versions and visualize them in a topology.

In Figures 5.9 and 5.10, the differences between the 2011 and 2018 versions of the DRC service are illustrated. However, because the versions of all components in the application stack have changed, the difference in the topology is huge as outlined by Figures 5.9 and 5.10. If many changes were applied between two versions, it is difficult to find correspondences. Therefore, an improved matching for topologies could include a version and context aware matching, where elements are matched, if they are in the same neighborhood and are of the same "base" type.

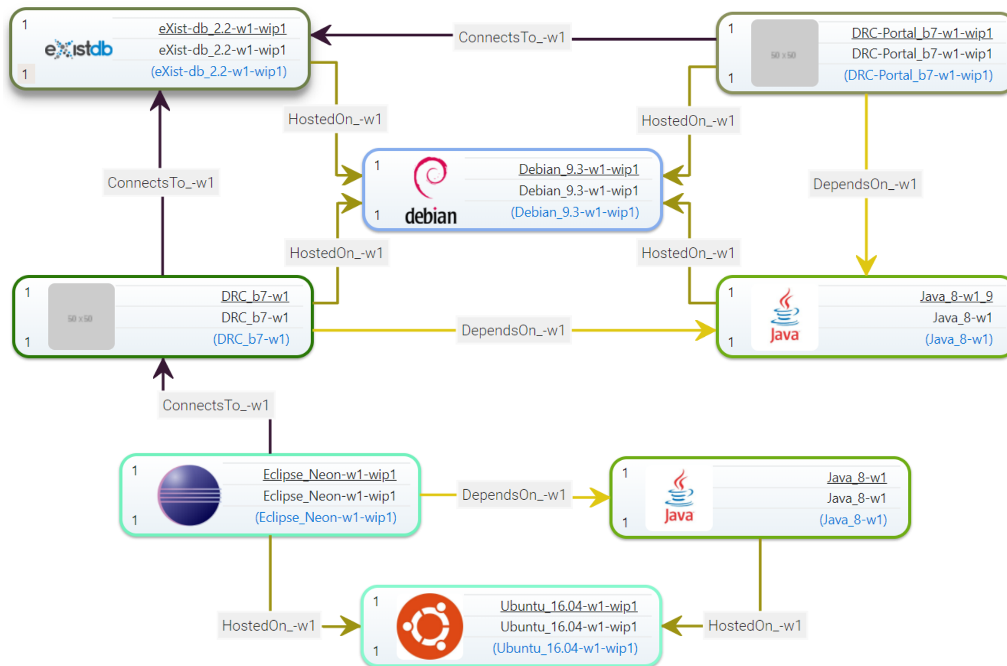

Figure 5.7: DRC topology in the 2011 version.

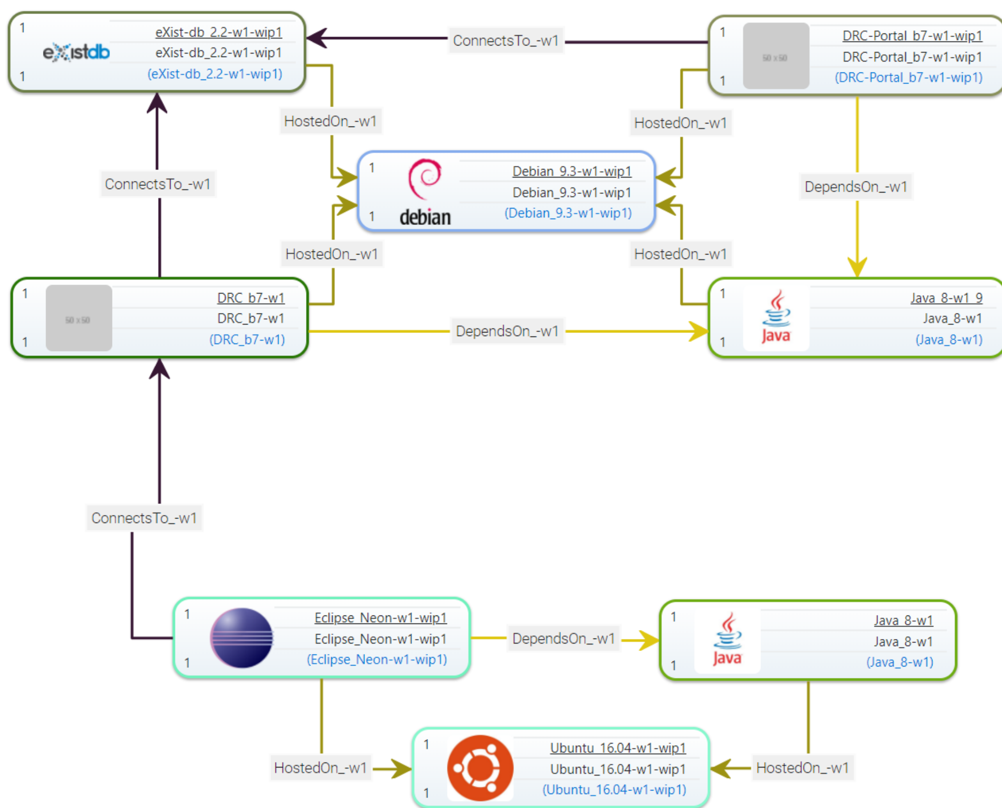

Figure 5.8: DRC topology in the 2018 version

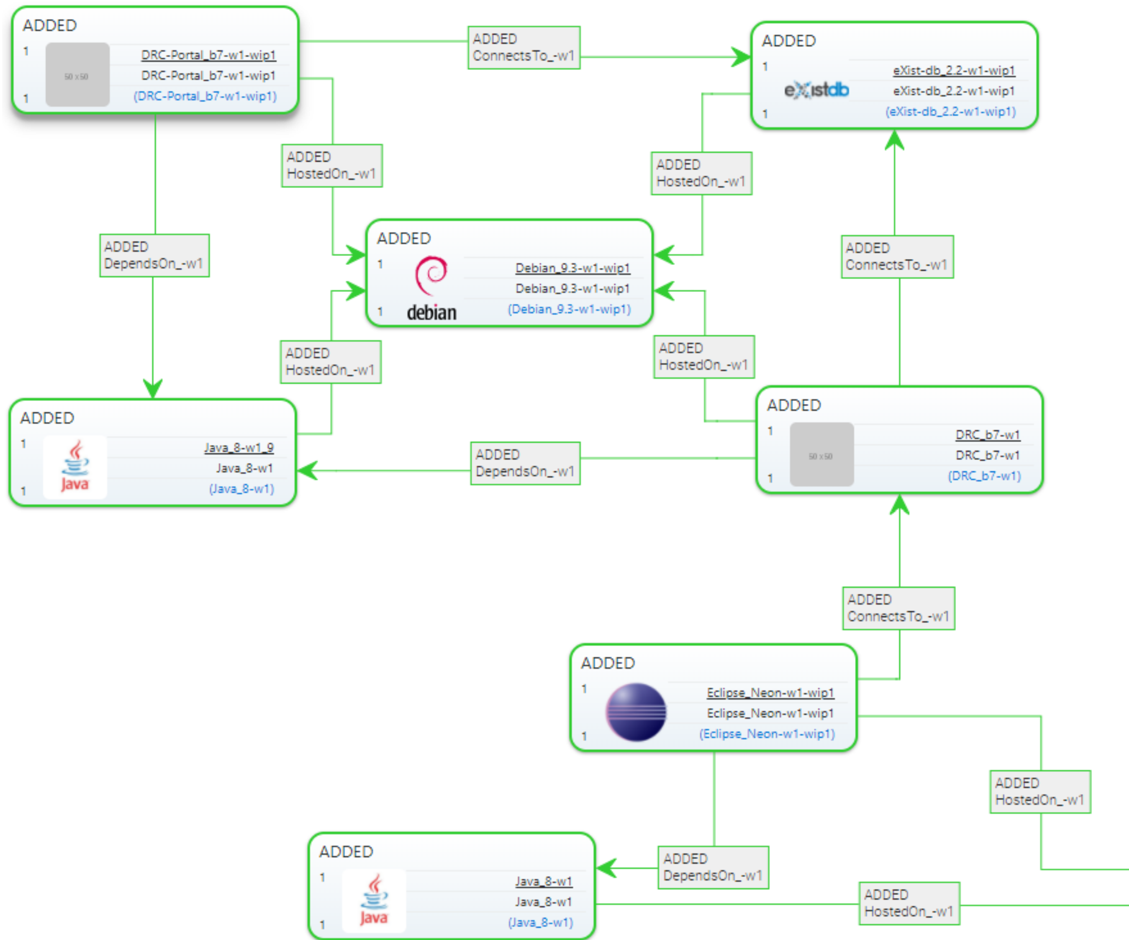

Figure 5.9: Difference visualization between two DRC topologies. Left part.

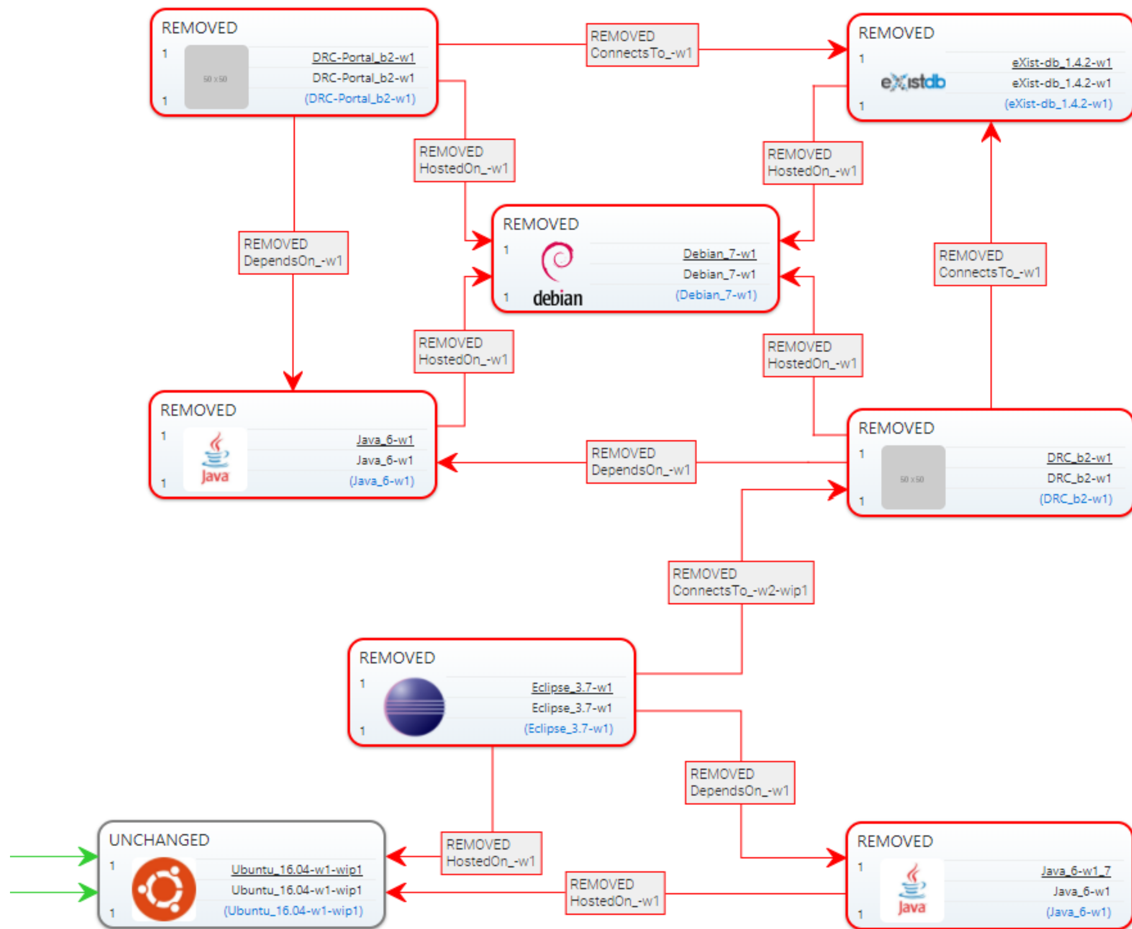

Figure 5.10: Difference visualization between two DRC topologies. Right part.

6 Summary and Outlook

This thesis presented an approach for version identifiers into the Topology Orchestration Specification for Cloud Applications (TOSCA). It tackles the issue of breaking service topologies after changes were made to any of the referenced TOSCA definitions. By introducing a versioning approach, it is always ensured that committed TOSCA definitions cannot be edited anymore.

To achieve versioning of Topology Orchestration Specification for Cloud Applications (TOSCA) definitions, it appends the version identifier of a TOSCA definition at the end of its name separated with an underscore. For example, a Node Type defining a `tomcat` web server in version `8` has the name `Tomcat_8-w3-wip2`. `w3-wip2` at the end represents the version inside TOSCA, whereas `w3` identifies the management version and `wip2` a development (or beta) version of the definition. This enables an user to differentiate between stable and unstable versions. If there is no development version, the definition is considered stable.

An approach was proposed which enables the calculation of differences between to versions. It generates a generic difference representation which can be further processed to create a graphical and textual visualization. While the graphical visualization of changes is limited to Topology Templates defined in Service Templates, the textual one generically visualizes differences between any two versions of one TOSCA definition.

The validation of both approaches was shown by a proof-of-concept implementation as an extension to the TOSCA modeling tool Eclipse Winery. It handles the version management and difference visualization in a way that an user does not need to know the approaches. The Winery presents versions detached from the name and offers the required management functionality. However, as described in Section 5.2, improvements can be made during the matching phase.

Future work could also include an automated update and test of Service Templates, where different Node Types are exchanged and deployments are performed automatically in order to test, whether the updated Stack still starts and works as expected. Further, if a well working stack is found, the respective elements could be tagged with a group name identifying them to work well together. This could enable a suggestion functionality providing propositions such as “other developers used...”.

An option regarding the visualization of changes over time could be a time line inside the Topology Modeler. While sliding up and down the time line, the topology adjusts itself accordingly. Starting from the first version up to the latest one, this method would present the service’s evolution in an interactive way.

Bibliography

[AKK+08] K. Altmanninger, G. Kappel, A. Kusel, W. Retschitzegger, M. Seidl, W. Schwinger, M. Wimmer. "AMOR–towards adaptable model versioning". In: *1st International Workshop on Model Co-Evolution and Consistency Management, in conjunction with MODELS*. Vol. 8. 2008, pp. 4–50 (cit. on p. 27).

[AP03] M. Alanen, I. Porres. "Difference and Union of Models". In: *«UML» 2003 - The Unified Modeling Language. Modeling Languages and Applications: 6th International Conference*. Springer Berlin Heidelberg, 2003, pp. 2–17. DOI: 10.1007/978-3-540-45221-8_2 (cit. on pp. 31, 33).

[ASW09] K. Altmanninger, M. Seidl, M. Wimmer. "A survey on model versioning approaches". In: *International Journal of Web Information Systems* 5.3 (2009), pp. 271–304. DOI: 10.1108/17440080910983556 (cit. on pp. 30, 33).

[BB09] J. Bauml, P. Brada. "Automated Versioning in OSGi: A Mechanism for Component Software Consistency Guarantee". In: *2009 35th Euromicro Conference on Software Engineering and Advanced Applications*. Aug. 2009, pp. 428–435. DOI: 10.1109/SEAA.2009.80 (cit. on p. 34).

[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: *11th International Conference on Service-Oriented Computing*. Springer Berlin Heidelberg, 2013, pp. 692–695. DOI: 10.1007/978-3-642-45005-1_62 (cit. on p. 26).

[BBK+12] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, D. Schumm. "Vino4TOSCA: A Visual Notation for Application Topologies Based on TOSCA". In: *OTM 2012, Part I*. Vol. 7565. Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2012, pp. 416–424. DOI: 10.1007/978-3-642-33606-5_25 (cit. on p. 18).

[BBK+14] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, J. Wettinger. "Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA". In: *Proceedings of the IEEE International Conference on Cloud Engineering (IEEE IC2E 2014)*. IEEE Computer Society, Mar. 2014, pp. 87–96. DOI: DOI10.1109/IC2E.2014.56 (cit. on p. 26).

[BBKL14a] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. "TOSCA: Portable Automated Deployment and Management of Cloud Applications". In: *Advanced Web Services*. Springer New York, 2014, pp. 527–549. DOI: 10.1007/978-1-4614-7535-4_22 (cit. on p. 21).

[BBKL14b] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. "Vinothek-A Self-Service Portal for TOSCA." In: *ZEUS*. 2014, pp. 69–72 (cit. on p. 26).

[Béz05] J. Bézivin. "On the unification power of models". In: *Software & Systems Modeling* 4.2 (2005), pp. 171–188. DOI: 10.1007/s10270-005-0079-0 (cit. on p. 32).

[BP08] C. Brun, A. Pierantonio. "Model Differences in the Eclipse Modeling Framework". In: *UPGRADE*. Vol. 9. 2008, pp. 29–34 (cit. on pp. 29, 30, 33, 60).

[BPS+08] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau. *Extensible markup language (XML) 1.0*. 2008 (cit. on p. 41).

[CDP07] A. Cicchetti, D. Di Ruscio, A. Pierantonio. "A metamodel independent approach to difference representation." In: *Journal of Object Technology* 6.9 (2007), pp. 165–185 (cit. on pp. 27, 32).

[CRGW96] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, J. Widom. "Change detection in hierarchically structured information". In: *ACM SIGMOD Record*. Vol. 25. 2. ACM. 1996, pp. 493–504 (cit. on pp. 29–31).

[EBF+17] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, J. Wettinger. "Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications". In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services (XPS), 2017, pp. 22–27 (cit. on p. 26).

[Epp02] D. Eppstein. "Subgraph isomorphism in planar graphs and related problems". In: *Graph Algorithms And Applications I*. World Scientific, 2002, pp. 283–309 (cit. on p. 30).

[FBK+16a] M. Falkenthal, U. Breitenbücher, K. Kepes, F. Leymann, M. Zimmermann, M. Christ, J. Neuffer, N. Braun, A. W. Kempa-Liehr. "OpenTOSCA for the 4th Industrial Revolution: Automating the Provisioning of Analytics Tools based on Apache Flink". In: *Proceedings of the 6th International Conference on the Internet of Things*. ACM, 2016, pp. 179–180. DOI: 10.1145/2991561.2998463 (cit. on p. 26).

[FBK+16b] A. C. Franco da Silva, U. Breitenbücher, K. Képes, O. Kopp, F. Leymann. " Open-TOSCA for IoT : Automating the Deployment of IoT Applications based on the Mosquitto Message Broker". In: *Proceedings of the 6th International Conference on the Internet of Things*. ACM, 2016, pp. 181–182. DOI: 10.1145/2991561.2998464 (cit. on p. 43).

[Fri02] J. E. Friedl. *Mastering regular expressions*. O'Reilly Media, Inc., 2002 (cit. on pp. 34, 35).

[GY04] J. L. Gross, J. Yellen. *Handbook of graph theory*. CRC press, 2004 (cit. on p. 30).

[JS05] D. Janzen, H. Saiedian. "Test-driven development concepts, taxonomy, and future direction". In: *Computer* 38.9 (Sept. 2005), pp. 43–50. DOI: 10.1109/MC.2005.314 (cit. on p. 67).

[JSR09] M. B. Juric, A. Sasa, I. Rozman. "WS-BPEL Extensions for Versioning". In: *Information and Software Technology* 51.8 (2009), pp. 1261–1274. DOI: 10.1016/j.infsof.2009.03.003 (cit. on pp. 28, 39, 40).

[KAZ18] O. Kopp, A. Armbruster, O. Zimmermann. "Markdown Architectural Decision Records: Format and Tool Support". In: *ZEUS*. 2018 (cit. on pp. 37, 57).

[KBBL12] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. "BPMN4TOSCA: A Domain-Specific Language to Model Management Plans for Composite Applications". In: *Business Process Model and Notation*. Vol. 125. Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, 2012, pp. 38–52. DOI: 10.1007/978-3-642-33155-8_4 (cit. on p. 22).

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. "Winery – A Modeling Tool for TOSCA-Based Cloud Applications". In: *Service-Oriented Computing: 11th International Conference*. Springer Berlin Heidelberg, 2013, pp. 700–704. DOI: 10.1007/978-3-642-45005-1_64 (cit. on pp. 18, 25, 57).

[KBF+17] K. Képes, U. Breitenbücher, M. P. Fischer, F. Leymann, M. Zimmermann. "Policy - Aware Provisioning Plan Generation for TOSCA - based Applications". In: *Proceedings of The Eleventh International Conference on Emerging Security Information, Systems and Technologies*. Xpert Publishing Services, Sept. 2017, pp. 142–149. ISBN: 978-1-61208-582-1 (cit. on p. 22).

[KE11] T. Kauppinen, G. M. de Espindola. "Linked Open Science-Communicating, Sharing and Evaluating Data, Methods and Results for Executable Papers". In: *Procedia Computer Science* 4 (2011), pp. 726–731. DOI: 10.1016/j.procs.2011.04.076 (cit. on p. 17).

[KGK+11] O. Kopp, K. Görlach, D. Karastoyanova, F. Leymann, M. Reiter, D. Schumm, M. Sonntag, S. Strauch, T. Unger, M. Wieland, et al. "A classification of BPEL extensions". In: *Journal of Systems Integration* 2.4 (2011), p. 3 (cit. on p. 28).

[KH18] O. Kopp, L. Harzenetter. *Discussion on Tosca Definition Version States*. 2018 (cit. on pp. 44, 45).

[KKU08] E. Kindler, P. Könemann, L. Unland. *Diff-based model synchronization in an industrial MDD process*. Tech. rep. Technical University of Denmark, DTU Informatics, Building 321, 2008 (cit. on p. 31).

[KKU09] P. Könemann, E. Kindler, L. Unland. "Difference-based model synchronization in an industrial mdd process". In: *Model Driven Tool and Process Integration* (2009) (cit. on pp. 27, 33).

[Kol09] D. S. Kolovos. "Establishing Correspondences between Models with the Epsilon Comparison Language". In: *Model Driven Architecture - Foundations and Applications: 5⁵ European Conference*. Springer Berlin Heidelberg, 2009, pp. 146–157. DOI: 10.1007/978-3-642-02674-4_11 (cit. on p. 31).

[KRPP09] D. S. Kolovos, D. D. Ruscio, A. Pierantonio, R. F. Paige. "Different models for model matching: An analysis of approaches to support model differencing". In: *Workshop on Comparison and Versioning of Software Models*. May 2009, pp. 1–6. DOI: 10.1109/CVSM.2009.5071714 (cit. on pp. 30, 31).

[Lac17] O. Lacan. *Keep a changelog*. 2017. URL: http://keepachangelog.com/en/1.0.0/ (cit. on pp. 33, 58, 64).

[Ley09] F. Leymann. "Cloud Computing: The Next Revolution in IT". In: *Photogrammetric Week '09*. Wichmann Verlag, 2009, pp. 3–12 (cit. on p. 17).

[LLN11] T. v. Lessen, D. Lübke, J. Nitzsche. *Geschäftsprozesse automatisieren mit BPEL*. 1. Edition. Dpunkt-Verl., 2011. ISBN: 978-3-89864-670-3 (cit. on p. 28).

[Men02] T. Mens. “A state-of-the-art survey on software merging”. In: *IEEE Transactions on Software Engineering* 28.5 (May 2002), pp. 449–462. DOI: 10.1109/TSE.2002.1000449 (cit. on p. 31).

[OAS07] OASIS. *Web Services Business Process Execution Language Version 2.0*. Apr. 11, 2007. URL: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html (cit. on p. 22).

[OAS13a] OASIS. *Topology and orchestration specification for cloud applications (TOSCA) Primer Version 1.0*. 2013. URL: http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.html (cit. on p. 21).

[OAS13b] OASIS, ed. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. Nov. 25, 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html (cit. on pp. 17, 21, 23, 38, 52, 54, 74).

[OAS17] (OASIS). *TOSCA Simple Profile in YAML Version 1.2*. Ed. by L. B. Matt Rutkowski, C. Lauwers. 2017. URL: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.2.html (cit. on p. 21).

[Ogb02] U. Ogbuji. *Namespaces and versioning*. Ed. by IBM. 2002. URL: https://www.ibm.com/developerworks/library/x-tipnamsp/ (cit. on pp. 27, 39, 40).

[OMG11] OMG. *Business Process Model and Notation (BPMN) Version 2.0*. 2011. URL: http://www.omg.org/spec/BPMN/2.0/ (cit. on p. 22).

[OWK03] D. Ohst, M. Welle, U. Kelter. “Differences Between Versions of UML Diagrams”. In: *SIGSOFT Softw. Eng. Notes* 28.5 (Sept. 2003), pp. 227–236. DOI: 10.1145/949952.940102 (cit. on pp. 32, 33, 57, 61).

[Pet95] M. Petre. “Why looking isn’t always seeing: readership skills and graphical programming”. In: *Commun. ACM* 38.6 (1995), pp. 33–44. DOI: 10.1145/203241.203251 (cit. on p. 57).

[Pre13] T. Preston-Werner. *Semantic Versioning 2.0.0*. 2013. URL: https://semver.org (cit. on p. 34).

[Red+17] RedHat Inc et al. *Guidelines for Naming Fedora Packages*. 2017. URL: https://fedoraproject.org/wiki/Packaging:Naming?rd=Packaging:NamingGuidelines#Release_Tag (cit. on p. 34).

[Rus17] S. W. Russ Allbery Bill Allombert Andreas Barth. *Debian Policy Manual*. Nov. 30, 2017. URL: https://www.debian.org/doc/debian-policy/#document-ch-archive (cit. on p. 34).

[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. DOI: https://doi.org/10.1016/j.jss.2015.12.025 (cit. on p. 17).

[SC13] M. Stephan, J. R. Cordy. “A Survey of Model Comparison Approaches and Applications.” In: *Modelsward*. 2013, pp. 265–277 (cit. on p. 30).

[Som16] I. Sommerville. *Software engineering*. 10. ed., global ed. Always learning. Pearson, 2016. ISBN: 978-1-292-09613-1 (cit. on p. 27).

[SUS+17] SUSE LLC et al. *openSUSE:Package naming guidelines*. 2017. URL: https://en.opensuse.org/openSUSE:Package_naming_guidelines (cit. on p. 34).

[Ver18] A. Verma. *9 Best Linux Distros For Programming And Developers (2018 Edition)*. Jan. 9, 2018. URL: https://fossbytes.com/best-linux-distros-for-programming-developers/ (cit. on p. 34).

[WBKL16] J. Wettinger, U. Breitenbücher, O. Kopp, F. Leymann. "Streamlining DevOps automation for Cloud applications using TOSCA as standardized metamodel". In: *Future Generation Computer Systems* 56.Supplement C (2016), pp. 317–332. DOI: 10.1016/j.future.2015.07.017 (cit. on p. 26).

[WBL14] J. Wettinger, U. Breitenbücher, F. Leymann. "Standards-based DevOps Automation and Integration Using TOSCA". In: *Proceedings of the 7th International Conference on Utility and Cloud Computing (UCC 2014)*. IEEE Computer Society, 2014, pp. 59–68 (cit. on p. 26).

[WWB+13] T. Waizenegger, M. Wieland, T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, B. Mitschang, A. Nowak, S. Wagner. "Policy4TOSCA: A Policy-Aware Cloud Service Provisioning Approach to Enable Secure Cloud Computing". In: *On the Move to Meaningful Internet Systems: OTM 2013 Conferences*. Springer Berlin Heidelberg, 2013. DOI: 10.1007/978-3-642-41030-7_26 (cit. on p. 22).

All links were last followed on April 3, 2018.

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