Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Backward and Forward Compatibility for TOSCA Simple Profile in YAML Version 1.0: Concept and Modelling Tooling Support

Christoph Kleine

Course of Study:

Examiner:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Supervisor:	Dr. rer. nat. Oliver Kopp

Commenced:	November 1, 2016
Completed:	May 3, 2017

CR-Classification: K.4.3

// Abstract

Automatic provisioning and management of cloud applications is a major issue of enterprise IT. Topology and Orchestration for Cloud Application (TOSCA) is an extensible standard written in XML. TOSCA specifies a language, that automates deployment and management processes while focusing on portability, interoperability and vendor-neutral ecosystems. TOSCA Simple Profile in YAML is a rendering of TOSCA written in YAML, which aims to provide a more accessible syntax. This thesis proposes a meta model for TOSCA Simple Profile in YAML and compares the TOSCA elements from both specifications. Using the information from the comparison, a converter has been constructed as part of Winery, a web-based environment to graphically model TOSCA topologies.

Contents

1 Introduction 13

2 Related Work 15

3 Language Comparison 19
- 3.1 Overview on YAML 19
- 3.2 YAML compared to XML 20

4 Comparison of Main TOSCA Components 23
- 4.1 Comparison Proceeding 23
- 4.2 Abstract Elements 27
- 4.3 Artifacts 28
- 4.4 Application Topology Nodes 31
- 4.5 Requirements 35
- 4.6 Capabilities 37
- 4.7 Interfaces 39
- 4.8 Topology Template 41
- 4.9 Application Topology Relationships 42
- 4.10 Workflows / Plans 45
- 4.11 Policies 47
- 4.12 Definitions 49
- 4.13 Service Template 51
- 4.14 Import Definition 52
- 4.15 Repository Definition 53
- 4.16 Data Types 54
- 4.17 Groups 54
- 4.18 Property Definitions 56
- 4.19 Node Filter Definition and Property Filter Definition 58
- 4.20 Additional Specification Information 58

5 Mapping of Specification 59

6 Implementation 63

7 Summary and Outlook 69

Bibliography 71

List of Figures

4.1 TOSCA XML element information from TOSCA XML specification 23
4.2 TOSCA YAML element information from TOSCA YAML specification . . . 24
4.3 Simplification of TOSCA YAML and TOSCA XML elements 25
4.4 Simplified Grammar examples 26

6.1 Part of UML diagram for Policy Type and Entity Type 64
6.2 UML class diagram for winery module 66

List of Tables

5.1 Mapping of TOSCA YAML specification 59
5.2 Mapping of TOSCA YAML specification 60
5.3 Mapping of TOSCA XML specification 61

6.1 Part of Policy Type keyname table 64

List of Listings

2.1 YANG example 15
2.2 YANG example converted to TOSCA 16
2.3 Heat-Translator TOSCA Service Template example 18
2.4 Heat-Translator Heat Template example 18

3.1 Examples of TAG and YAML directives 21

4.1 Pseudo schema defining TOSCA XML Artifact Templates 26
4.2 Original grammar for TOSCA YAML Artifact Definition 26
4.3 Simplified grammar for TOSCA YAML Entity Type 27
4.4 Simplified grammar for TOSCA XML Entity Type 27
4.5 Simplified grammar for TOSCA XML Artifact Type 29
4.6 Simplified grammar for TOSCA YAML Artifact Type 29
4.7 Simplified grammar for TOSCA XML Artifact Templates 30
4.8 Simplified grammar for TOSCA YAML Artifact Definition 30
4.9 Simplified grammar for TOSCA XML Node Types and Node Type Implementation 32
4.10 Simplified grammar for TOSCA YAML Node Types 32
4.11 Simplified grammar for TOSCA XML Node Template 33
4.12 Simplified grammar for TOSCA YAML Node Template 33
4.13 Simplified grammar for TOSCA XML Requirement Definition 35
4.14 Simplified grammar for TOSCA YAML Requirement Definition 35
4.15 Simplified grammar for TOSCA XML Requirement Type 36
4.16 Simplified grammar for TOSCA XML Requirements 36
4.17 Simplified grammar for TOSCA YAML Requirements Assignment 37
4.18 Simplified grammar for TOSCA XML Capability Type 37
4.19 Simplified grammar for TOSCA YAML Capability Type 37
4.20 Simplified grammar for TOSCA XML Capability Definition 38
4.21 Simplified grammar for TOSCA YAML Capability Definition 38
4.22 Simplified grammar for TOSCA XML Capability Assignment 39
4.23 Simplified grammar for TOSCA YAML Capability Assignment 39
4.24 Simplified grammar for TOSCA XML Interface Definition 40
4.25 Simplified grammar for TOSCA YAML Interface Definition 40

4.26 Simplified grammar for TOSCA YAML Interface Type 41
4.27 Simplified grammar for TOSCA XML Topology Template 41
4.28 Simplified grammar for TOSCA YAML Topology Template 41
4.29 Simplified grammar for TOSCA XML Relationship Type and TOSCA XML Relationship Type Implementation 43
4.30 Simplified grammar for TOSCA YAML Relationship Type 43
4.31 Simplified grammar for TOSCA XML Relationship Template 44
4.32 Simplified grammar for TOSCA YAML Relationship Template 44
4.33 Simplified grammar for TOSCA YAML Imperative Workflow Definition . 45
4.34 Simplified grammar for TOSCA YAML Precondition Definition 46
4.35 Simplified grammar for TOSCA YAML Step Definition 46
4.36 Simplified grammar for TOSCA YAML Activity Definition 47
4.37 Simplified grammar for TOSCA XML Policy Type 47
4.38 Simplified grammar for TOSCA YAML Policy Types 47
4.39 Simplified grammar for TOSCA XML Policy and TOSCA XML Policy Template 48
4.40 Simplified grammar for TOSCA YAML Policy Definition 48
4.41 Simplified grammar for TOSCA XML Definitions Document 49
4.42 Part 1 of simplified grammar for TOSCA YAML Service Template 50
4.43 Simplified grammar for TOSCA XML Service Template 51
4.44 Part 2 of simplified grammar for TOSCA YAML Service Template 52
4.45 Simplified grammar for TOSCA XML Import Definition 52
4.46 Simplified grammar for TOSCA YAML Import Definition 53
4.47 Simplified grammar for TOSCA YAML Repository Definition 53
4.48 Simplified grammar for TOSCA YAML Data Type 54
4.49 Simplified grammar for TOSCA YAML Constraint Clause 54
4.50 Simplified grammar for TOSCA YAML Group Type 55
4.51 Simplified grammar for TOSCA YAML Group Definition 55
4.52 Simplified grammar for TOSCA YAML Property Definition 56
4.53 Simplified grammar for TOSCA YAML Property Assignment 57
4.54 Simplified grammar for TOSCA YAML Attribute Definition 57
4.55 Simplified grammar for TOSCA YAML Attribute Assignment 57
4.56 Simplified grammar for TOSCA YAML Parameter Definition 58
4.57 Simplified grammar for TOSCA YAML Node Filter Definition 58
4.58 Simplified grammar for TOSCA YAML Property Filter Definition 58

6.1 Original grammar for Interface Definitions 65

1 Introduction

An application lifecycle consists of the operations: Provisioning and Management. For these, manual involvement is required, which leads to high costs and human caused errors [Ley09].

The open standard for cloud application portability: Topology and Orchestration Specification for Cloud Applications (TOSCA) from OASIS [PS13] specifies a language, that automates the deployment and management processes while focusing on portability, interoperability and vendor-neutral Ecosystems. Over 100 participants from over 40 companies [OAS17], including big industry leaders, support this standard.

TOSCA has three Building Blocks: Application topologies, Workflows and Policies. Application Topologies describe services as a set of node and relationships templates. Each node template has a node type, properties, deployment artifacts, implementation artifacts, capabilities and requirements, that are used to install, configure and deploy the node. Relationship templates have a type and define the relation between two nodes. E.g. basic relationship types are: "`hostedOn`", "`connectsTo`" and "`dependsOn`". Workflows are defined as portable, reusable and automated plans. They manage parallel execution, error handling, traceability, auditability, and other workflow related tasks. Policies add the monitoring ability to the orchestration and allow ongoing evaluation of Rules, enforcing of Service Level Agreements and invocation of other processes.

TOSCA version 1.0 utilizes XML Schema 1.0 and is extensible, therefore it allows the use of attributes and elements from other namespaces. The next version TOSCA 2.0 [TY16] is based on YAML. The differences of the two versions can be sorted into three categories. The first category contains renaming differences, for example properties that have a different names in TOSCA version 2.0. The second category contains missing or additional properties. The last category contains relocated properties, which describes information that has moved from one element into a different element. These differences will be the main topic of this master thesis.

The implementation section of this master thesis describes a translator between TOSCA version 1.0 and TOSCA version 2.0. This translator is developed as a sub module for the Winery [ESF17a], which is a modeling tool for TOSCA-based cloud applications. The modeling tool is developed with Java and offers a Web-based environment for graph-based modeling of TOSCA version 1.0 [PS13].

2 Related Work

Searching in Google with the queries “tosca yaml converter” and “tosca yaml transformer” results in very few reasonable results, which this chapter examines.

The first result is yttc, a YANG to TOSCA converter [yttc16]. Yet Another Next Generation (YANG) is a modular data modeling language for the Network Configuration (NETCONF) protocol. The NETCONF protocol is defined by IETF [EBBS11] and provides mechanisms to install, manipulate, and delete the configuration of network devices. Operations are realized as remote procedure calls. YANG files contain YANG modules, which specify the hierarchy of data that can be used by NETCONF operations. Listing 2.1 presents an example of a YANG module with a container, which exists only to organize the hierarchy of data nodes (Line 3) and the module definition statements (Lines 4–9).

Listing 2.1 YANG example

```
1  module my-system {
2     /* namespace, prefix, imports, includes, revision ... */
3     container system {
4        leaf host-name { type string; description "Hostname for this system"; }
5        leaf-list domain-search { type string; description "List of domain names to
             search"; }
6        list interface { key "name"; description "List of interfaces in the system";
7           leaf name { type string; }
8           leaf type { type string; }
9           leaf mtu { type int32; }
10       }
11    }
12 }
```

Yttc takes header information (yang-version, namespace, prefix), linkage statements (import, include), meta information (organization, contact, description, reference), revision history and converts them into TOSCA Metadata. The module and the module definition are converted into TOSCA Node Types and TOSCA Data Types. The result of the conversion is represented in Listing 2.2.

The second relevant search result is Heat-Translator [Ope16a], an OpenStack project licensed under Apache 2. Heat is an OpenStack Orchestration program, which implements an engine to launch cloud applications based on text file templates. The Heat-Translator

Listing 2.2 YANG example converted to TOSCA

```
1  data_types:
2    _config: { properties: { system: { required: false, type: system_type }}}
3    _rpc: { properties: {}}
4    interface_type:
5      properties:
6        mtu: { required: false, type: integer }
7        name: { required: false, type: string }
8        type: { required: false, type: string }
9    system_type:
10     properties:
11       domain-search: { description: '[List_of: string_type]', required: false }
12       host-name: { required: false, type: string }
13       interface: { description: '[List_of: interface_type]', required: false }
14 node_types:
15   my-system:
16     derived_from: cloudify.netconf.nodes.xml_rpc
17     properties:
18       config:
19         required: false
20         type: _config
21       metadata:
22         _: # namespace URI
23         # revision, ...
24       rpc:
25         required: false
26         type: _rpc
```

can process non-Heat templates, e.g. TOSCA YAML templates or TOSCA Cloud Service Archives and create an in memory graph, which is mapped to Heat resources and produces a Heat Orchestration Template (HOT).

The project uses a TOSCA Simple Profile in YAML parser written in python, which generates object oriented TOSCA python classes. The translator translates TOSCA Node Templates, Inputs, Outputs, and Node Types to HOT resources. The Heat-Translator has a set of python classes, which extend the generic HotResource class and offer the translation of specific TOSCA Types [Ope16b]:

- `tosca.nodes.BlockStorage` represents a local block storage device offering evenly sized data blocks.
- `tosca.nodes.BlockStorageAttachment` (not a default Node Type) represents "`AttachesTo`" relationship for "`Compute`" and "`BlockStorage`".
- `tosca.policies.Scaling.Cluster` (not a default Policy Type) is used to govern scaling of a cluster of nodes.

- `tosca.nodes.Compute` represents processors of software applications or services.
- `tosca.nodes.Database` is a logical database.
- `tosca.nodes.DBMS` represents a relational SQL Database Management System.
- `tosca.nodes.network.FloatingIP` (not a default Node Type but part of the definition of TOSCA types used by the tosca-parser [Ope17b, TOSCA_definition_1_0.yaml]) is a node with a floating IP that can associate to a Port.
- `tosca.nodes.network.Network` represents a simple, logical network service.
- `tosca.nodes.network.Port` is a logical entity that associates between Compute and Network types.
- `tosca.nodes.ObjectStorage` represents a storage used to store data as objects.
- `tosca.policies.Placement` is a Policy Type used to govern the placement of TOSCA nodes.
- `tosca.policies.Scaling` represents a Policy Type used to govern the scaling of TOSCA nodes.
- `tosca.nodes.SoftwareComponent` is a generic software component.
- `tosca.nodes.WebApplication` represents a software application that can be run by a web server.
- `tosca.nodes.WebServer` is an abstract software component that is able to host and manage web applications.

The above list from the Heat translator contains a huge part of the default TOSCA YAML Node Types, but lacks the Node Types corresponding to Runtime Container, Application Container and Load Balancer. Missing Policy Types are Update and Performance Policies. Default Data Types, Artifact Types, Capability Types, Requirement Types and Interface Types are not present in the Heat-Translator as well.

For example, the definition of a server (Listing 2.3) in TOSCA YAML is shown to be converted to a Heat Template with a resource named “`server`” (Line 8 in Listing 2.4). The TOSCA YAML Service Template defines a Compute Node (Line 8ff. in Listing 2.3) with a host capability (Line 10ff. in Listing 2.3) and an OS capability (Line 15ff. in Listing 2.3). The first capability is converted to a flavor property (Line 11 in Listing 2.4) and the second to a resource type (Line 9 in Listing 2.4).

The rest of the entries retrieved by the Google search were either about the previously discussed converters, or YAML to JSON converters or other irrelevant topics.

Listing 2.3 Heat-Translator TOSCA Service Template example

```
1  tosca_definitions_version: tosca_simple_yaml_1_0
2
3  description: Testfile
4
5  topology_template:
6    node_templates:
7      server:
8        type: tosca.nodes.Compute
9        capabilities:
10         host:
11           properties:
12             disk_size: 10 GB
13             num_cpus: 4
14             mem_size: 8 GB
15         os:
16           properties:
17             architecture: x86_64
18             type: Linux
19             distribution: Fedora
20             version: 27.0
```

Listing 2.4 Heat-Translator Heat Template example

```
1  heat_template_version: 2013-05-23
2
3  description: >
4   Testfile
5
6  parameters: {}
7  resources:
8   server:
9     type: OS::Nova::Server
10    properties:
11      flavor: m1.large
12      user_data_format: SOFTWARE_CONFIG
13 outputs: {}
```

3 Language Comparison

This chapter describes the differences between the two specification languages: YAML and XML. Section 3.1 presents the language YAML. Since XML is widely known, Section 3.2 directly compares YAML with XML.

3.1 Overview on YAML

YAML is the short form of "YAML Ain't Markup Language" and focuses on human readability and interoperability. The YAML specification [YamlSpec09] defines seven design goals for the language:

- Easily human readable
- Data portable between programming languages
- Match native data structure of agile languages
- Consistent model to support generic tools
- Supports one-pass processing
- Expressive and extensible
- Easy to implement and use

The translation between a native data structure and the presentation character stream is done in three steps.

1. The first step takes the data structure and converts it to a representation node graph by adding tags that represent type information and by converting the data structure to a combination of sequences, mappings and scalars.

   Scalar nodes contain zero or more Unicode characters. Sequences contain zero or more ordered nodes, which can include the same node more than once, and Mappings are unordered sets of key value pairs.

Tags can either be global tags “`URI`” and unique across all applications or local tags “`!Name`”. In addition to the expected YAML node type (scalar, sequence, mapping), tags can provide other metadata information: E.g. allowed content values or a mechanism for tag resolution.

2. The next step is to serialize the representation node graph to a serialization event tree. This is done by ordering unordered mapping nodes and by resolving aliases and anchors.

   Aliases and anchors are used when one node appears multiple times. The first use will be identified by an anchor and any further occurrence of the node is serialized as an alias.

3. The last step is to present the serialization event tree as a presentation stream, which means choosing node styles and scalar formats.

   The node style specifies that a node is represented as a block or a flow, plain or quoted, in-line or next-line, while the format defines the presentation, for example integers can be written as decimal or hexadecimal numbers.

   The presentation character stream optionally contains comments and directives which are not part of the event tree or the node graph. Comments are presentation detail and not bound to any node.

   Directives have a name and an optional set of parameters. The YAML specification [YamlSpec09] defines two directives, “`YAML`” and “`TAG`”. The first directive specifies the version of the YAML document which is set by default to “`%YAML 1.2`”. The second directive can be used to specify shorthand notations for node tags. Listing 3.1 shows how both directives are used: There are three different possibilities to define the type for a value. A single “!” indicates a local tag, a double “!!” indicates the default global tag “`tag:yaml.org,2002:int`”. Thus, the authority “`yaml.org`” (owned by the YAML authority since 2002) defines the integer language-independent type “`int`” together with other default types for scalars and collections [BEI05].

   The third possibility is to use a tag handle, for example “`!e!str`” (Line 5) or “`!l!task`” (Line 8), which both must have an associated “`TAG`” (Line 2–3) directive.

3.2 YAML compared to XML

YAML has no direct correlation to XML [YamlSpec09, Section 1.4]. While YAML is primarily designed as a serialization language, XML is backwards compatible to Standard

Listing 3.1 Examples of TAG and YAML directives

```
1 %YAML 1.2
2 %TAG !e! tag:example.org,2017:types/
3 %TAG !l! !local-
4 ---
5 forename: !e!str "foo" #global schme: "tag:example.org,20017:types/str"
6 surename: !<tag:example.org,2017:types/str> "bar"
7 taskA: !task "foo_bar" #local tag: "!task"
8 taskB: !l!task "foo_bar[]" #local tag : "!local-task"
9 age: !!int "27" #global scheme: "tag:yaml.org,2002:int
```

Generalized Markup Language (SGML), which was designed to support structured documentation.

The XML Information Set (Infoset) [CT04] specifies a standardized, abstract data set, whose purpose is to provide a consistent set of definitions to be reused by other specifications. Each XML document can be converted to this data set if the document is well-formed and satisfies the namespace constraints. The XML Infoset consists of several information items:

- Document Information Item: Only once in the information set and all other items can be accessed through its properties.
- Element Information Item: Represents an XML element.
- Attribute Information Item: Represents a single XML attribute inlcuding namespace declarations.
- Processing Instruction Information Items: Represent XML Processing instructions allowing them to contain instructions for application.
- Unexpanded Entity Reference Information Items: Placeholders for unparsed external entities.
- Character Information Items: Represent characters to appear as character references or within a CDATA section.
- Comment Information Items: Represent comments appearing in the XML document.
- Document Type Declaration Information Item: Represents an XML document type declaration.
- Unparsed Entity Information Items: Represent unparsed general entities declared in a DTD.
- Notation Information Item: Represents notations declared in a DTD.

- Namespace Information Item: Represents a namespace existing in the scope of an elements.

XML Infosets can be serialized to XML, Binary XML, CSV, JSON, and other formats including YAML [Gud02, Section Mapping to APIs].

YAXML [BEI06] defines an XML Binding for YAML, which takes a subset of each language and defines a mapping. The conversion of default YAML elements is straight forward:

- Mappings are converted to elements.
- Lists are expressed as sequences.
- Scalars are modeled as single text node child.

YAML does not need a root element, which must be chosen for XML. Anchors and Aliases can be modelled by anchor attributes and empty elements with alias attributes matching a previous anchor. Tags can be converted to namespaces, but XML namespaces cannot be converted in general to Tags since YAML tags allow only limited tag URIs.

The fundamental difference between YAML and XML is that the YAML processor needs knowledge about the used type – if not the default types – in order to read a YAML document. Thus, the processor must have implemented the types specified. XML, however, uses schemata specifying the document type and the data types and allowing a Unmarshaller to automatically check and deserialize XML documents. A YAML document can be written with less effort than a XML document, but the deserialization process becomes complicated when the document contains complex types.

4 Comparison of Main TOSCA Components

There are currently two TOSCA specifications: One is the “Topology and Orchestration Specification for Cloud Applications Version 1.0” from [PS13]. It uses XML as serialization format and thus it is called “TOSCA XML” in the following. The other TOSCA specification is the “TOSCA Simple Profile in YAML Version 1.1” from [TY17]. It uses YAML as serialization format and thus is called “TOSCA YAML” in the following.

This chapter compares elements from TOSCA XML with elements from TOSCA YAML. This is done by establishing a simplified grammar comparing elements from both specifications. Section 4.1 describes the proceeding of the comparison. The following sections present the actual comparison of the main TOSCA Components.

4.1 Comparison Proceeding

This section describes how the information about main TOSCA components is obtained from both specification and transformed into a comparable simplified grammar.

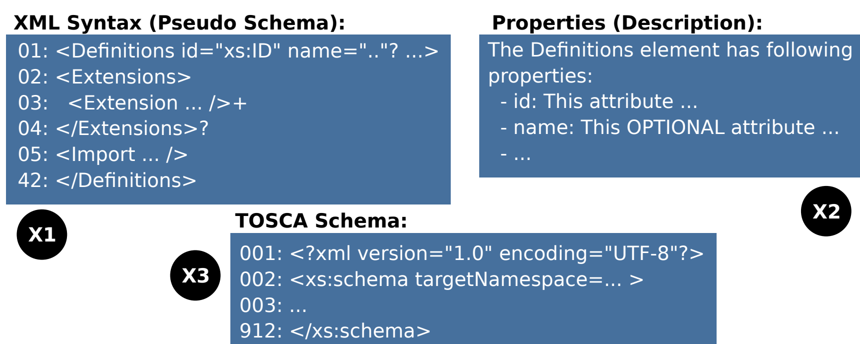

Figure 4.1: TOSCA XML element information from TOSCA XML specification. (X1) [PS13, Section 4.1], (X2) [PS13, Section 4.2] and (X3) [PS13, Appendix D. TOSCA Schema].

Grammar:(Example for Service Template)

```
01: tosca_definitions_version: #Required TOSCA ...
02: metadata:
03:    template_name: <value>
04:    ...
05: description: <template_type_description>
06: imports: # ordered list of import definitions
07: ...
```

Y1

Keynames table:

```
Keyname | Required | Type | Description
- artifact_types | no | list of Artifact Types |
  This section contains an optional ...
- metadata | no | map of string | Defines ...
- tosca_definitions_version | yes | string | ...
```

Y2

Figure 4.2: TOSCA YAML element information from TOSCA YAML specification

The information the TOSCA XML specification is providing for a TOSCA element is presented in Figure 4.1. Thus, the main TOSCA elements have a pseudo schema (X1), a list of property descriptions (X2) and they are part of the complete TOSCA schema (X3).

The TOSCA YAML specification structures TOSCA element information in a different way (Figure 4.2). Each TOSCA element has at least one grammar (Y1), in some cases there are several grammars. For example the element Artifact Definition has a single-line grammar “`<artifact_name>: <artifact_file_URI>`”, which can be used if the file type, which is required, can be inferred from the file URI. The multi-line grammar contain all properties, which are specified in the keyname table (Y2). For the comparison with TOSCA XML elements, the complete multi-line grammar is used. Some TOSCA YAML elements have different grammars based on the context the elements are used in, in such cases the grammars are combined and than converted.

Because the grammar (Y1) does not contains any information about the type or whether specific properties are required, the TOSCA YAML specification presents a keyname table (Y2) additionally providing descriptions of the properties.

The TOSCA YAML and TOSCA XML specification describe information about TOSCA elements in a different way. To make elements from both specification comparable, a simplification process, which is presented in Figure 4.3, constructs a new simplified grammar.

For TOSCA XML elements the simplification starts with converting several used datatypes to simple types (S1). TOSCA YAML does not have datatypes for IDs, URIs, and references since the standard YAML types boolean, int, string, and float are used. The next step (S2) is to ignore properties or attributes that are not relevant for the comparison. In case of the attribute “`Documentation`” the comparison is simple, since each TOSCA XML element has this property and most of the TOSCA YAML elements have a property named “`description`”. During the third simplification step (S3) the style of TOSCA XML elements is converted to the style of TOSCA YAML elements.

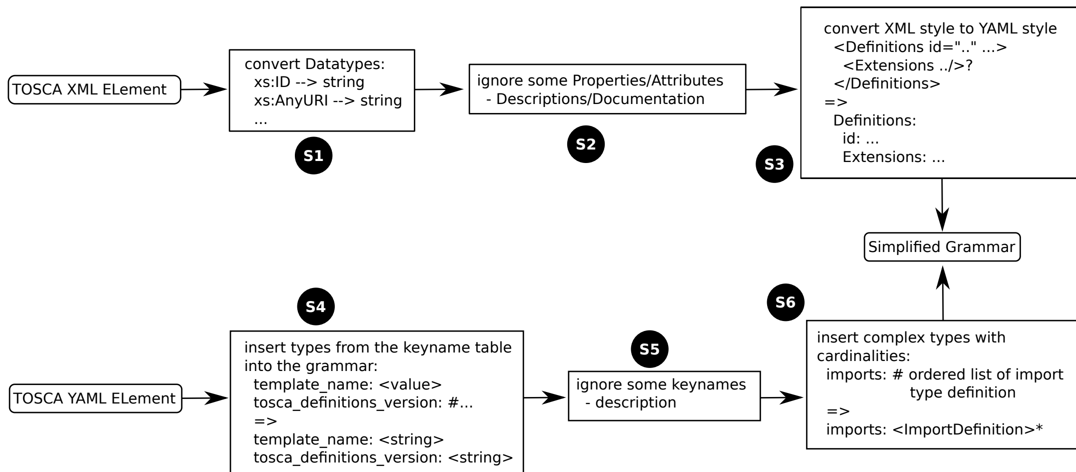

Figure 4.3: Simplification of TOSCA YAML and TOSCA XML elements, modeled with FMC [Tab06]

The simplification process for TOSCA YAML elements starts with inserting the simple types (string, boolean, integer, float) from the keyname table into the grammar (S4). Complex types specifying other TOSCA elements replace the rest of the textual descriptions in the grammar (S6). The same properties that were ignored (S2) for the TOSCA XML elements are ignored for TOSCA YAML elements as well (S5).

In the last step of the TOSCA YAML or TOSCA XML simplification, typographical conventions similar to the ones provided for in the TOSCA XML specification (cf. [PS13, Section 2.5]) are used. Thus, the characters "?", "*", "+" indicate the cardinality (0 or 1, 0 or more, 1 or more) of the elements, "|" denotes a choice and parentheses indicate the scope of the cardinality characters.

Examples of the results are presented in Figure 4.4. The new simplified grammar for TOSCA XML and TOSCA YAML uses indentation to specify sub-elements, and it uses cardinality characters (default cardinality is 1). The syntax "`entry:*`" (A1) specifies that multiple entries are possible and an entry is a complex element with sub-elements.

Showing an additional example, the pseudo schema of an artifact template (Listing 4.1) and the grammar of an artifact definition (Listing 4.2) are used to be converted to simplified grammars (Listings 4.7 and 4.8).

After creating both simplified grammars, the differences are compared with respect to three categories:

- Elements that differ in name
- Elements and functionality that YAML has, but are missing in XML

TOSCA YAML Simplified Grammar (A1)

```
01: ServiceTemplate:
02:   tosca_definitions_version: <string>
03:   metadata:
04:     entry:*
05:       key: <string>
06:       value: <string>
07:   imports: <ImportDefinition>*
08:   ...
14: ImportDefinition:
15:   file: <string>
16:   repository: <string>?
17:   ...
```

TOSCA XML Simplified Grammar (A2)

```
01: Definitions:
02:   id: <string>
03:   name: <string>?
04:   Extensions:
05:     Extension:*
06:       namespace: <string>
07:       mustUnderstand: <boolean>?
08:   Import: <ImportDefinition>*
09:   ...
14: ImportDefinition:
15:   namespace: <string>?
16:   location: <string>?
17:   ...
```

Figure 4.4: Simplified Grammar examples

Listing 4.1 Pseudo schema defining TOSCA XML Artifact Templates [PS13, Section 13.1]

```
1  <ArtifactTemplate id="xs:ID" name="xs:string"? type="xs:QName">
2     <Properties>
3           XML fragment
4     </Properties> ?
5        <PropertyConstraints>
6           <PropertyConstraint property="xs:string" constraintType="xs:anyURI"> +
7              constraint ?
8           </PropertyConstraint>
9        </PropertyConstraints> ?
10       <ArifactReferences>
11          <ArtifactReference reference="xs:anyURI">
12          (
13             <Include pattern="xs:string"/>
14          |
15             <Exclude pattern="xs:string"/>
16          )*
17          </ArtifactReference> +
18    </ArtifactReferences> ?
19 </ArtifactTemplate>
```

Listing 4.2 Original grammar for TOSCA YAML Artifact Definition [TY17, Section 3.5.6.2.2]

```
1  <artifact_name>:
2     description: <artifact_description>
3     type: <artifact_type_name>
4     file: <artifact_file_URI>
5     repository: <artifact_repository_name>
6     deploy_path: <file_deployment_path>
```

- Elements and functionality that XML has, but are missing in YAML

4.2 Abstract Elements

This section describes abstract elements other elements are derived from and which can be compared.

4.2.1 Entity Type

Entity Type is the base type for Node Types (Section 4.4.1), Relationship Types (Section 4.9.1), Artifact Types (Section 4.3.1), Requirement Types (Section 4.9.1), Capability Types (Section 4.6.1), Interface Types (Section 4.7), Policy Types (Section 4.11.1) and Data Types (Section 4.16).

Listing 4.3 Simplified grammar for TOSCA YAML Entity Type [TY17, Section 3.6.1]

```
1  EntityType:
2     name: <string>
3     derived_from: <string>?
4     version: <string>?
5     metadata:
6        entry:*
7           key: <string>
8           value: <string>
9
10    properties: <PropertyDefinition>*
```

Listing 4.4 Simplified grammar for TOSCA XML Entity Type [TY17, Appendix D. TOSCA Schema line 197]

```
1  EntityType:
2     name: <string>
3     DerivedFrom: <string>
4     Tags:
5        Tag:*
6           name: <string>
7           value: <string>
8     abstract: <boolean>?
9     final: <boolean>?
10    targetNamespace: <string>?
11    PropertiesDefinition: <PropertyDefinition>*
```

The TOSCA YAML and XML Entity Types have an identifier “`name`” (Line 2 in Listings 4.3 and 4.4) to uniquely identify the type and an attribute “`derived from`” (Line 3 in Listings 4.3 and 4.4) to specify the super type.

The attributes “`Tags`” (Line 4 in Listing 4.4) and “`metadata`” (Line 5 in Listing 4.3) differ by their names, they are, however, optional elements used to describe the Entity Type.

The sub-element “`PropertiesDefinition`” (Line 11 Listing 4.4) is only defined for TOSCA XML Entity Types. TOSCA YAML has this sub-element declared for all derived sub-types, but Interface Type. To simplify the grammars the element “`properties`” (Line 10 in Listing 4.3) has been added to TOSCA YAML Entity Type.

The attributes “`abstract`” and “`final`” (Line 8–9 in Listing 4.4) are not supported in the TOSCA YAML language. In TOSCA XML, however, the first specifies that no instance can be created from the Artifact Template if “`abstract`” is set to true, and the second specifies that it is not possible to derive from the Entity Type if “`final`” is set to true.

TOSCA XML provides the additional attribute “`targetNamespace`” (Line 10 in Listing 4.4), which is directly included as a namespace prefix in the TOSCA YAML Type name. For exampfe, the name “`tosca.artifacts.Root`” has the namespace prefix “`tosca`” indicating that it is part of the TOSCA YAML namespace [TY17, Section 3.1.1]. The “`version`” (Line 4 in Listing 4.3) property supports the uniqueness of elements: TOSCA YAML requires that imported Type Definitions must be equivalent if they have the same namespace URI, local name and version.

4.3 Artifacts

The TOSCA language describes an artifact as the content needed to realize a deployment. In this context an artifact may be any type of script or program, database dump files, virtual images and other relevant data.

TOSCA XML divides artifacts into two types:

- Deployment Artifacts: Represent the executable for materializing instances of a node.
- Implementation Artifacts: Represent the executable of an operation of a node type.

TOSCA YAML logically divides the Artifact Types into categories and states that additional types will be developed for future drafts of the specification [TY17, Section 5.4]:

Deployment Artifacts: Relevant for the deployment phase and used in create and install operations.

Implementation Artifacts: Implement TOSCA Interface operations.

Runtime Artifacts: Used during runtime in the final application, e.g., data for the database or program libraries.

In order to use artifacts, the user should write an Artifact Definition/Template with an Artifact Type. TOSCA YAML provides several default Artifact Types:

tosca.artifact.Root: Parent of all Artifact Types.

tosca.artifact.File: For Artifacts defined as files where any special handling is not needed.

tosca.artifacts.Deployment: Parent of all Deployment Artifact Types.

tosca.artifacts.Deployment.Image: Represents Deployment Artifact that are images.

tosca.artifacts.Deployment.Image.VM: Virtual Machine Image.

tosca.artifacts.Implementation: Parent of Implementation Artifact Types.

tosca.artifacts.Implementation.Bash: Bash scripts.

tosca.artifacts.Implementation.Python: Python scripts.

4.3.1 Artifact Type

In addition to the default Artifact Types the user can define personalized Artifact Types.

Listing 4.5 Simplified grammar for TOSCA XML Artifact Type [PS13, Section 12.1]

```
1 ArtifactType:
2     [... elements inherited from Entity Type ...]
```

Listing 4.6 Simplified grammar for TOSCA YAML Artifact Type [TY17, Section 3.6.4.2]

```
1 ArtifactType:
2     [... elements inherited from Entity Type ...]
3     mime_type: <string>?
4     file_ext: <string>*
```

The two grammars Listings 4.5 and 4.6 are similar. The TOSCA XML Artifact Type does not define new attributes or elements than its super type: Entity Type.

TOSCA YAML extends the TOSCA XML Artifact Type definition with the attributes "`mime_type`" (Line 3 in Listing 4.6) and "`file_ext`" (Line 4 in Listing 4.6). The first describes optional Multipurpose Internet Mail Extensions (MIME) type [FB96] and the second describes optional file extensions for the artifacts files.

4.3.2 Artifact Template/Definition[1]

Artifact Template/Definition[1] allows the user to define artifacts of the specific types. Listings 4.7 and 4.8 show the simplified grammars for TOSCA XML and TOSCA YAML.

Listing 4.7 Simplified grammar for TOSCA XML Artifact Templates [PS13, Section 13.1]

```
1  ArtifactTemplate:
2     id: <string>
3     name: <string>?
4     type: <string>
5     Properties: <Properties>*
6     PropertyConstraints: <PropertyConstraint>*
7     ArtifactReferences:
8        ArtifactReference:*
9           reference: <string>
10          Include:*
11             pattern: <string>
12          Exclude:*
13             pattern: <string>
```

Listing 4.8 Simplified grammar for TOSCA YAML Artifact Definition [TY17, Section 3.5.6.2.2]

```
1  ArtifactDefinition:
2     id: <string>
3     type: <string>
4     file: <string>
5     repository: <string>?
6     deploy_path: <string>?
```

Both simplified grammars have the attributes "`id`" (Line 2 in Listings 4.7 and 4.8) and "`type`" (Line 4 in Listing 4.7 and Line 3 in Listing 4.8), which identify the unique name and the Artifact Type.

The attribute "`file`" (Line 4 in Listing 4.8) and the element "`ArtifactReferences`" (Line 7 in Listing 4.7) have the same purpose: to specify the artifact file URI with a string. TOSCA XML allows referencing multiple artifacts in one Artifact Template. For example, the URI points to a directory instead of a file and several "`Include pattern`" (Line 10–11 in Listing 4.7) or "`Exclude pattern`" (Line 12–13 in Listing 4.7) specify a set of artifact files.

[1]When writing Artifact Template/Definition, this means that TOSCA XML calls the concept "Artifact Template" and TOSCA YAML calls the concept "Artifact Definition".

With the property "`repository`" (Line 5 in Listing 4.8) TOSCA YAML offers to the user to be able to reference artifacts that are part of repositories and allows the user to specify the "`deploy_path`" (Line 6 in Listing 4.8) for the artifacts.

TOSCA XML has additional attributes "`name`" (Line 3 in Listing 4.7), "`Properties`" (Line 5 in Listing 4.7) and "`PropertyConstraints`" (Line 6 in Listing 4.7). The first specifies the name of the Artifact Template. "`Properties`" and "`Property Constraints`" specify invariant properties and constraints on the Artifact Type properties.

4.4 Application Topology Nodes

Application Topology Nodes describe the manageable software components the cloud application is made of. *E.g.*, a Service Template for a MySQL Database can have three Nodes: A database node, a database management system node, and an operating system node.

4.4.1 Node Type

Each Application Topology Node has a Node Type which defines the structure of properties and the supported interfaces. The TOSCA YAML and XML Node Types are both derived from their abstract Entity Types (Section 4.2.1).

In addition to the differences and similarities inherited from the super types TOSCA YAML/XML Entity Type, both grammars have the properties "`requirements`" (Line 3 in Listing 4.9 and Line 4 in Listing 4.10) a list of Requirement Definitions (Section 4.5.1), "`capabilities`" (Line 4 in Listing 4.9 and Line 5 in Listing 4.10) a list of Capability Definitions (Section 4.6.2) and "`interfaces`" (Line 8 in Listing 4.9 and Line 6 in Listing 4.10) a list of Interface Definitions (Section 4.7).

TOSCA XML has a "`NodeTypeImplementation`" beside "`NodeType`". Node Type Implementations describe executable code that implements a specific Node Type. Compared to TOSCA YAML, which defines "`artifacts`" (Line 7 in Listing 4.10) a list of Artifact Definitions Section 4.3.2, TOSCA XML has the elements "`RequiredContainerFeatures`" (Line 16 in Listing 4.9), "`ImplementationArtifacts`" (Line 19 in Listing 4.9) and "`DeploymentArtifacts`" (Line 25 in Listing 4.9).

"`RequiredContainerFeatures`" are features the "`NodeTypeImplementation`" depends on, for example libraries needed to deploy virtual images. The TOSCA container can select the appropriate "`NodeTypeImplementation`" with respect to the features.

Listing 4.9 Simplified grammar for TOSCA XML Node Types [PS13, Section 6.1] and TOSCA XML Node Type Implementation [PS13, Section 7.1].

```
1  NodeType:
2     [... elements inherited from Entity Type ...]
3     RequirementDefinitions: <RequirementDefinition>*
4     CapabilityDefinitions: <CapabilityDefinition>+
5     InstanceStates:
6        InstanceState:*
7           state: <string>
8     Interfaces: <Interface>*
9
10 NodeTypeImplementation:
11    Tags:
12       Tag:*
13          name: <string>
14          value: <string>
15    DerivedFrom: <string>?
16    RequiredContainerFeatures:
17       RequiredContainerFeature:*
18          feature: <string>
19    ImplementationArtifacts:
20       <ImplementationArtifact>:*
21          interfaceName: <string>
22          operationName: <string>?
23          artifactType: <string>
24          artifactRef: <string>?
25    DeploymentArtifacts:
26       <DeploymentArtifact>:*
27          name: <string>
28          artifactType: <string>
29          artifactRef: <string>?
```

Listing 4.10 Simplified grammar for TOSCA YAML Node Types [TY17, Section 3.6.9.2]

```
1  NodeType:
2     [... elements inherited from Entity Type ...]
3     attributes: <AttributeDefinition>*
4     requirements: <RequirementDefinition>*
5     capabilities: <CapabilityDefinition>*
6     interfaces: <InterfaceDefinitions>*
7     artifacts: <ArtifactDefinition>*
```

Implementation and Deployment Artifacts reference an Artifact Template or contain the Artifact as sub-element. Additionally, TOSCA XML Implementation Artifacts optionally provide the “`interfaceName`” (Line 21 in Listing 4.9) and “`operationName`” (Line 22 in Listing 4.9).

TOSCA YAML distinguishes between properties (inherited from Entity Type) used by template authors to provide input values to TOSCA entities, and "`attributes`" (Line 3 in Listing 4.10) used to expose the actual state of TOSCA entity properties.

The sub-element "`InstanceStates`" (Line 5 in Listing 4.9) of TOSCA XML Node Types offers the template author to specify a set of states the Node Type can occupy. The attribute "`state`" (Line 7 in Listing 4.9) represents an URI identifying a potential state.

4.4.2 Node Template

A Node Template is an instance of a Node Type.

Listing 4.11 Simplified grammar for TOSCA XML Node Templates [PS13, Chapter 5]

```
1  NodeTemplate:
2     id: <string>
3     name: <string>?
4     type: <string>
5     Properties: <Property>*
6     PropertyConstraints: <PropertyConstraints>*
7     minInstances: <integer>?
8     maxInstances: <string>?
9     Requirements: <Requirement>*
10    Capabilities: <Capability>*
11    Policies: <Policy>*
12    DeploymentArtifacts: <DeploymentArtifacts>*
```

Listing 4.12 Simplified grammar for TOSCA YAML Node Templates [TY17, Section 3.7.3.2]

```
1  NodeTemplate:
2     id: <string>
3     type: <string>
4     metadata:
5        entry:*
6           key: <string>
7           value: <string>
8     properties: <PropertyAssignment>*
9     attributes: <AttributeAssignment>*
10    directives: <string>*
11    requirements: <RequirementAssignment>*
12    capabilities: <CapabilityAssignment>*
13    interfaces: <InterfaceDefinition>*
14    artifacts: <ArtifactDefinition>*
15    node_filter: <NodeFilter>
16    copy: <string>
```

TOSCA YAML and TOSCA XML have template specific elements presented in light gray (Line 2–9 in Listing 4.12 and Line 2–6 in Listing 4.11). Both templates have an “Id”, a “Type”, and “Properties” assignments. TOSCA YAML has additional property “metada” for specifying extra information and “AttributeAssignments” for assigning values to the TOSCA YAML Attributes. TOSCA XML has the additional element “PropertyConstraints”, which allows the template author to specify constraints on Node Type properties, and the additional element “name”.

The remaining elements in the grammars are not template specific: Both simplified grammars have “requirements” (Line 11 in Listing 4.12 and Line 9 in Listing 4.11), a list of Requirement Assignments (Section 4.5.2) and “capabilities” (Line 12 in Listing 4.12 and Line 10 in Listing 4.11), a list of Capability Assignments (Section 4.6.3).

TOSCA XML has the element “DeploymentArtifacts” (Line 12 in Listing 4.11), which specifies Deployment Artifacts which instantiate an Artifact Type (Section 4.3.1) and reference an Artifact Template (Section 4.3.2). This element is similar to “artifacts” (Line 14 in Listing 4.12), which specifies a list of Artifact Definitions.

TOSCA YAML defines the additional elements “directives” (Line 10 in Listing 4.12), “copy” (Line 16 in Listing 4.12) and “node_filter” (Line 15 in Listing 4.12). Directives provide processing instructions to the orchestrator and tooling, but there are no directive values defined for the current version of TOSCA YAML. The “copy” attribute specifies the optional name of the Node Template, from which this Node Template has been copied, and “NodeFilter” (Section 4.19)helps the TOSCA orchestrator to select a target node.

TOSCA YAML allows the template author to specify “interfaces” (Line 13 in Listing 4.12) for a Node Template besides the interfaces provided by Node Types. This is not available in TOSCA XML.

TOSCA XML has the element “policies” (Line 11 in Listing 4.11). This element specifies Policy Assignments, which specifies a Policy Type (Section 4.11.1) and references a Policy Definition (Section 4.11.2). The TOSCA YAML specification writes in the description of TOSCA Policy Definition, that policies can be associated with a TOSCA topology or other top-level entities, including Node Templates [TY17, Section 3.7.6]. Because Policy Definitions are neither part of the known keywords for Node Templates, nor part of the original Node Template grammar and only part of the Topology Template (Section 4.8), it is possible that future drafts of the specification include Policy Definitions for Node Template.

Other missing elements for TOSCA YAML are “minInstances” (Line 7 in Listing 4.11) and “maxInstances” (Line 8 in Listing 4.11), which specify the maximum number and minimum number of Application Topology Nodes that can be instantiated from a Node Template.

4.5 Requirements

Requirements describe dependencies of Application Topology Nodes that need to be fulfilled by Capabilities from other components. For example TOSCA XML proposes several ways to match Requirements of a Node Template: Either by another Node Template in the same Service Template or by the general hosting environment or by the TOSCA container.

4.5.1 Requirement Definition

TOSCA YAML and TOSCA XML Node Types both allow the template author to specify Requirement Definitions.

Listing 4.13 Simplified grammar for TOSCA XML Requirement Definition [PS13, Section 6.1]

```
1 RequirementDefinition:
2    name: <string>
3    requirementType: <string>
4    lowerBound: <integer>
5    upperBound: <string>
6    Constraints:
7       Constraint:*
8          constraintType: <string>
9          content: <any>
```

Listing 4.14 Simplified grammar for TOSCA YAML Requirement Definition [TY17, Section 3.6.3.1]

```
1 RequirementDefinition:
2    name: <string>
3    capability: <string>
4    node: <string>?
5    relationship: <string>?
6       type: <string>?
7       interfaces: <InterfaceDefinition>*
8    occurences: <string>*
```

TOSCA YAML does not use Requirement Types. This is part of the simplification process: Instead of declaring Requirement Types from nodes, TOSCA YAML declares their features sets using TOSCA Capability Types. Each TOSCA XML Requirement Definition has a Requirement Type (Listing 4.15). The Type specifies the kind of requirement a Node Type can expose. The structure of observable properties, e.g., name, data types, allowed values is given as a Property Definition.

Listing 4.15 Simplified grammar for TOSCA XML Requirement Type [PS13, Chapter 10]

```
1 RequirementType:
2    [... elements inherited from Entity Type ...]
3    name: <string>
4    requiredCapabilityType: <string>
```

Listing 4.16 Simplified grammar for TOSCA XML Requirements [PS13, Section 5.1]

```
1 Requirement:
2    id: <string>
3    name: <string>
4    type: <string>
5    Properties: <Property>*
6    PropertiesContraints: <PropertyConstraint>*
```

TOSCA YAML specifies "`occurrences`" (Line 8 in Listing 4.14), which is similar to "`lowerBound`" (Line 4 in Listing 4.13) and "`upperBound`" (Line 5 in Listing 4.13) and defines the minimal and maximal occurrences for the Requirement.

The element "`requiredCapabilityType`" (Line 4 in Listing 4.15), which is part of the TOSCA XML Requirement Type and references the types of Capabilities that can match the Requirement, is similar to the element "`capability`" (Line 3 in Listing 4.14).

TOSCA XML defines the additional element "`Constraints`" (Line 6 in Listing 4.13), which specifies constraints for properties from the Requirement Type.

TOSCA YAML has the additional elements "`node`" (Line 4 in Listing 4.14) and "`relationship`" (Line 5 in Listing 4.14). The first element is optional and specifies the Node Type containing the required "`capability`" (Line 3 in Listing 4.14), and the second optional element specifies the resulting Relationship Type, which gets constructed when fulfilling the requirement.

4.5.2 Requirement Assignment

This section presents the grammars for TOSCA YAML Requirement Assignments and TOSCA XML Requirement elements. Both grammars depend on their Requirement Definition and will not be compared in detail.

Requirement Assignments (Listing 4.17) are used in TOSCA YAML Node Templates to fulfill a Requirement by either specifying concrete Node Templates or providing abstract selection criteria to find matching Node Templates.

TOSCA XML defines the "`Requirement`" (Line 1 in Listing 4.16) element. The template author specifies a Requirement Type, Properties and Property Constraints.

Listing 4.17 Simplified grammar for TOSCA YAML Requirements Assignment [TY17, Section 3.7.2.2.2]

```
1  RequirementAssignment:
2     name: <string>
3     node: <string>?
4     relationship: <string>?
5        type: <string>
6        properties: <PropertyAssingment>*
7        interfaces: <InterfaceAssignment>*
8     capability: <string>?
9     node_filter: <NodeFilterDefinition>?
10    occurrences: <string>*
```

Comparing TOSCA YAML Requirement Assignment with TOSCA YAML Requirement Definition, the Requirement Assignment has one addition: "`node_filter`" (Line 9 in Listing 4.17). This element can be used at runtime by TOSCA orchestrators or providers to choose a type-compatible target node.

4.6 Capabilities

TOSCA Capabilities are used to describe a transparent capability or feature that can be associated with Node Types or Node Templates.

4.6.1 Capability Type

TOSCA XML and YAML Capabilities have Capability Types that are used for Requirements or instantiated as Capability Definitions.

Listing 4.18 Simplified grammar for TOSCA XML Capability Type [PS13, Section 11.1]

```
1  CapabilityType:
2     [... elements inherited from Entity Type ...]
```

Listing 4.19 Simplified grammar for TOSCA YAML Capability Type [TY17, Section 3.6.7.2]

```
1  CapabilityType:
2     [... elements inherited from Entity Type ...]
3     attributes: <AttributeDefinition>
4     valid_source_types: <string>
```

Aside from the differences of the Entity Types (Section 4.2.1) and the previously discussed "`attributes`" (Line 3 in Listing 4.19), TOSCA YAML has the additional element "`valid_source_types`" (Line 4 in Listing 4.19), which provides an optional list of suitable Node Types to serve as sources of relationships to a node with the declared Capability Type.

The TOSCA XML Capability Type define no extra elements and contains the element inherited from the Entity Type (Line 2 in Listing 4.18).

4.6.2 Capability Definition

Capabilities for Node Types are defined as Capability Definitions.

Listing 4.20 Simplified grammar for TOSCA XML Capability Definition [PS13, Section 6.1]

```
1 CapabilityDefinition:
2    name: <string>
3    capabilityType: <string>
4    lowerBound: <integer>?
5    upperBound: <string>?
6    Constraints:
7       Constraint:*
8          constraintType: <string>
```

Listing 4.21 Simplified grammar for TOSCA YAML Capability Definition [TY17, Section 3.6.2.2.2]

```
1 CapabilityDefinition:
2    name: <string>
3    type: <string>
4    properties: <PropertyDefinition>*
5    attributes: <AttributeDefinition>
6    valid_source_types: <string>*
7    occurrences: <string>*
```

Both simplified grammars have the elements "`type`" (Line 3 in Listings 4.20 and 4.21), "`name`" (Line 2 in Listings 4.20 and 4.21) and "`occurences`" (Line 7 in Listing 4.21 and Line 4–5 in Listing 4.20).

TOSCA YAML has the additional elements "`valid_source_types`" (Line 6 in Listing 4.21), similar to the valid source types specified for Capability Types, "`properties`" (Line 4 Listing 4.21) and "`attributes`" (Line 5 in Listing 4.21). TOSCA XML can define Properties only for Capability Types.

As described in the previous section TOSCA XML has no property for defining valid source types for Capability Types. For Capability Definition TOSCA XML has the option to provide "`Constraints`" (Line 6 in Listing 4.20), which can be used to restrict the valid source types and to define other constraints.

4.6.3 Capability Assignment

Capabilities Assignments are used in Node Templates (Section 4.4.2) to assign values to "`properties`" (Line 3 in Listings 4.22 and 4.23), in case of TOSCA YAML additionally to "`attributes`" (Line 4 in Listing 4.23), and in case of TOSCA XML to "`PropertyConstraints`" (Line 4 in Listing 4.22) of named Capabilities Definitions.

Listing 4.22 Simplified grammar for TOSCA XML Capability Assignment[PS13, Section 6.1]

```
1 Capability:
2    name: <string>
3    Properties: <Property>*
4    PropertyConstraints: <PropertyConstraints>*
```

Listing 4.23 Simplified grammar for TOSCA YAML Capability Assignment [TY17, Section 3.7.1]

```
1 CapabilityAssignment:
2    name: <string>
3    properties: <PropertyAssignments>*
4    attributes: <AttributeAssignments>*
```

4.7 Interfaces

Interfaces contain operation definitions that can be performed on instances of Node Types, and these can be used to manage relationships.

Both grammars have the element "`name`" (Line 2 in Listings 4.24 and 4.25) and "`Operation`" (Line 7 in Listing 4.25 and Line 3 in Listing 4.24).

The Operation Definition elements have again a sub-element "`name`" (Line 10 in Listing 4.25 and Line 4 in Listing 4.24), but TOSCA XML has "`InputParameters`" and "`OutputParameters`" (Line 5–6 in Listing 4.24) while TOSCA YAML defines "`inputs`" (Line 14 in Listing 4.25). TOSCA YAML does not distinguish between Input and Output Parameter, but by using Property Definitions or Assignments, which will be available for all

Listing 4.24 Simplified grammar for TOSCA XML Interface Definition [PS13, Section 6.1]

```
1  InterfaceDefinition:
2     name: <string>
3     Operation:∗
4        name: <string>
5        InputParameters: <Parameter>∗
6        OutputParameters: <Parameter>∗
```

Listing 4.25 Simplified grammar for TOSCA YAML Interface Definition [TY17, Section 3.5.14.2.1]

```
1  InterfaceDefinition:
2     name: <string>
3     type: <InterfaceType>?
4     inputs:
5        <PropertyAssignment>∗
6        <PropertyDefinition>∗
7     <OperationDefinition>∗
8
9  OperationDefinition:
10    name: <string>
11    implementation: <string>
12       primary: <string>
13       dependencies: <string>∗
14    inputs:
15       <PropertyDefinition>∗
16       <PropertyAssignment>∗
```

operation implementation artifacts or for each operation individually, such parameters might be constructed.

TOSCA YAML Operation Definition defines the element “`implementation`” (Line 11 in Listing 4.25), which is used to specify an implementation artifact and dependencies for the artifact.

Another difference is that TOSCA YAML uses Interface Types, which inherit all elements from the Entity Type, and define “`inputs`” (Line 3 in Listing 4.26) and “`operations`” (Line 4 in Listing 4.26).

Listing 4.26 Simplified grammar for TOSCA YAML Interface Type [TY17, Section 3.6.5.2]

```
1 InterfaceType:
2    [... elements inherited from Entity Type ...]
3    inputs: <PropertyDefinition>*
4    <OperationDefinition>*
```

4.8 Topology Template

A Topology Template specifies the structure of a cloud application. The main components are Node Templates and Relationship Templates, which define the components and the relationship between the components of the application.

Listing 4.27 Simplified grammar for TOSCA XML Topology Template [PS13, Section 5.1]

```
1 TopologyTemplate:
2    NodeTemplates: <NodeTemplate>*
3    RelationshipTemplates: <RelationshipTemplate>*
```

Listing 4.28 Simplified grammar for TOSCA YAML Topology Template [TY17, Section 3.8]

```
1 TopologyTemplate:
2    inputs: <ParameterDefinition>*
3    node_templates: <NodeTemplate>*
4    relationship_templates: <RelationshipTemplate>*
5    groups: <GroupDefinition>*
6    policies: <PolicyDefinition>*
7    outputs: <ParameterDefinition>*
8    substitution_mappings: <SubstitutionMapping>*
9    workflows: <Workflows>*
```

Both grammars define “`node_templates`” (Line 3 in Listing 4.28 and Line 2 in Listing 4.27), which is a list of Node Templates (Section 4.4.2), and “`relationship_templates`” (Line 4 in Listing 4.28 and Line 3 in Listing 4.27), a list of Relationship Templates (Section 4.9.2).

TOSCA YAML has several additional elements: Input and Output Parameter Definitions (Line 2 and Line 7 in Listing 4.28), which allows defining parameters, with constraints and default values. Input Parameter can be mapped to properties of node or relationship templates, this is similar to the TOSCA XML “`BoundaryDefinitions`” (Line 7 in Listing 4.43) in the Service Template. Output Parameters allow to expose node or relationship template attributes to users of a service. Combining these elements with “`substitution_mapping`” (Line 8 in Listing 4.28) allows the template author to implement a Topology Template as a Node Type.

Furthermore, TOSCA XML has the elements “`groups`”, “`policies`” and “`worksflows`” (Line 5,6,9 in Listing 4.28) These allow the template author to specify groups of Node Template (Section 4.4.2), assign Policies (Section 4.11.2) to Entities and specify Workflows (Section 4.10).

4.9 Application Topology Relationships

An Application Topology Relationship defines the connection between Application Topology Nodes. Topology Templates have Relationship Templates, which refer to a Relationship Type.

4.9.1 Relationship Type

TOSCA YAML and TOSCA XML define Relationship Type elements, additionally, TOSCA XML defines Relationship Type Implementations, which represents runnable code that implements a specific Relationship Type.

TOSCA YAML has simplified the TOSCA XML elements: Instead of the elements “`SourceInterfaces`” and “`TargetInterfaces`” (Line 6–7 in Listing 4.29) only “`interfaces`” (Line 4 in Listing 4.30) can be defined.

The TOSCA XML elements “`ValidTarget`” and “`ValidSource`” (Line 8 and Line 10 in Listing 4.29) can reference Node Types Section 4.4.1 or Capability Types Section 4.6.1. TOSCA YAML only specifies Capability Types as “`valid_target_types`” (Line 5 in Listing 4.30) for Application Topology Relationships and does not distinguish between source and target.

TOSCA XML defines “`InstanceStates`” (Line 3 in Listing 4.29), which specifies the states an instance of Relationship Type can occupy at runtime.

The elements specified for Relationship Type Implementations (Line 13–27 in Listing 4.29) are similar to the elements specified for Node Type Implementations (Line 10–29 in Listing 4.9), except that Relationship Type Implementations does not have Deployment Artifacts. TOSCA YAML allows the template author specify artifacts as part of the Operations of Interface Definitions Section 4.7.

Listing 4.29 Simplified grammar for TOSCA XML Relationship Type [PS13, Section 8.1] and TOSCA XML Relationship Type Implementation [PS13, Section 9.1]

```
1  RelationshipType:
2     [... elements inherited from Entity Type ...]
3     InstanceStates:
4        InstanceState:*
5           state: <string>
6     SourceInterfaces: <Interface>*
7     TargetInterface: <Interface>*
8     ValidSource:
9        typeRef: <string>?
10    ValidTarget:
11       typeRef: <string>?
12
13 RelationshipTypeImplementation:
14    Tags:
15       Tag:*
16          name: <string>
17          value: <string>
18    DerivedFrom: <string>?
19    RequiredContainerFeatures:
20       RequiredContainerFeature:*
21          feature: <string>
22    ImplementationArtifacts:
23       <ImplementationArtifact>:*
24          interfaceName: <string>
25          operationName: <string>?
26          artifactType: <string>
27          artifactRef: <string>?
```

Listing 4.30 Simplified grammar for TOSCA YAML Relationship Type [TY17, Section 3.6.10.2]

```
1  RelationshipType:
2     [... elements inherited from Entity Type ...]
3     attributes: <AttributeDefinition>*
4     interfaces: <InterfaceDefinition>*
5     valid_target_types: <string>*
```

4.9.2 Relationship Template

Relationship Templates are instances of Relationship Types and describe the occurrence of manageable relationship between Node Templates.

The light grey part of both grammars (Line 2–9 in Listing 4.32 and Line 2–6 in Listing 4.29) is template related and similar to the template related elements described in Node Templates Section 4.4.2.

Listing 4.31 Simplified grammar for TOSCA XML Relationship Template [PS13, Section 5.1]

```
1  RelationshipTemplate:
2     id: <string>
3     name: <string>?
4     type: <string>
5     Properties: <Property>*
6     PropertyConstraints: <PropertyConstraints>*
7     SourceElement:
8        ref: <string>
9     TargetElement:
10       ref: <string>
11    RelationshipConstraints: <RelationshipConstraint>*
```

Listing 4.32 Simplified grammar for TOSCA YAML Relationship Template [TY17, Section 3.7.4]

```
1  RelationshipTemplate:
2     id: <string>
3     type: <string>
4     metadata:
5        entry:*
6           key: <string>
7           value: <string>
8     properties: <PropertyAssignment>*
9     attributes: <AttributeAssignment>*
10    interfaces: <InterfaceDefinition>*
11    copy: <string>?
```

TOSCA YAML allows the instance of Relationship Type to have the elements "`interfaces`" (Line 10 in Listing 4.32), a list of Interface Definitions Section 4.7, and "`copy`" (Line 11 in Listing 4.32), which optionally specifies the original Relationship Template, from which this Relationship Template has been copied.

TOSCA XML specifies "`SourceElement`" and "`TargetElement`" (Line 7 and Line 9 in Listing 4.31), which references for sources a Node Template or Requirement and for targets a Node Template or Capability. Both referenced entities must be type compatible to the Relationship Type "`ValidSource`" and "`ValidTarget`" (Line 8 and Line 10 in Listing 4.29) elements.

Additionally, TOSCA XML template authors can define "`RelationshipConstraints`" (Line 11 in Listing 4.31), which can constraint the use of relationships.

4.10 Workflows / Plans

TOSCA XML defines Plans to describe management aspects of service instances. They are particularly responsible for the creation and termination of Topologies and can be defined in existing languages such as BPMN [OMG11] or BPEL [BEA03]. The process models contain tasks, which for example refer to operations of Node Template interfaces. It is not required for TOSCA XML to model Workflows. They can be derived from the topology model [BBK+14].

For Workflows TOSCA YAML has defined its own language to describe the deployment, undeployment or management of TOSCA topologies. Two types of Workflows exists: Declarative Workflows, that are automatically created by the TOSCA orchestrator and imperative Workflows, that are manually created by the template author.

A complete comparison between BPMN and BPEL and the TOSCA YAML Workflows in this thesis is out of scope. The following sections describe the main components of TOSCA YAML Workflows.

4.10.1 Imperative Workflow Definition

An Imperative Workflow Definition (Listing 4.33) contains the elements "`metadata`" (Line 2), "`inputs`" (Line 6), a list of Property Definitions (Section 4.18), "`preconditions`" (Line 7), a list of Precondition Definitions (Section 4.10.2), and "`steps`" (Line 8), a list of Step Definitions (Section 4.10.3).

Listing 4.33 Simplified grammar for TOSCA YAML Imperative Workflow Definition [TY17, Section 3.7.7.2]

```
1 ImperativeWorkflow:
2    metadata:
3       entry:*
4          key: <string>
5          value: <string>
6    inputs: <PropertyDefinition>*
7    preconditions: <PreconditionDefinition>*
8    steps: <StepDefinition>*
```

4.10.2 Precondition Definition

Precondition Definitions (Listing 4.34) check whether a workflow can be processed or not. The defined elements are "`target`" (Line 2), which specifies a Node Template or

Group name, "target_relationship" (Line 3), which specifies the name of a Requirement if the precondition is processed on a relationship, and "condition" (Line 4), a list of Condition Clause definition.

Listing 4.34 Simplified grammar for TOSCA YAML Precondition Definition [TY17, Section 3.5.20.2]

```
1 PreconditionDefinition:
2    target: <string>
3    target_relationship: <string>?
4    condition: <ConditionClause>*
```

4.10.3 Step Definition

Step Definitions (Listing 4.35) specify sequenced activities and their connection to other steps.

Listing 4.35 Simplified grammar for TOSCA YAML Step Definition [TY17, Section 3.5.21.2]

```
1 StepDefinition:
2    name: <string>
3    target: <string>
4    target_relationship: <string>?
5    operation_host: <string>?
6    filter: <ConstraintClause>*
7    activities: <ActivityDefinition>*
8    on_success: <string>*
9    on_failure: <string>*
```

The elements "target" (Line 3) and "target_relationship" (Line 4) are similar to the two elements defined in Precondition Definition Section 4.10.2.

The element "operation_host" (Line 5) specifies the name of the node on which operations should be executed, "filter" (Line 6) provide a filtering logic, "activities" (Line 7) is a list of Activity Definition Section 4.10.4, and the last two elements "on_success, on_failure" (Line 8–9) specify Step names that will be performed in case of success or failure.

4.10.4 Activity Definition

An Activity Definition (Listing 4.36) defines operations, which are performed in a Workflow.

Listing 4.36 Simplified grammar for TOSCA YAML Activity Definition [TY17, Section 3.5.17.2]

```
1  ActivityDefinition:
2     delegate: <string>?
3     set_state: <string>?
4     call_operation: <string>?
5     inline: <string>?
```

There are four different possibilities to define such operation. Thus, only one element of Activity Definition can be used per definition.

The element "`delegate`" (Line 2) specifies the name of the delegate workflow, "`set_state`" (Line 3) specifies the new value of a node state, "`call_operation`" (Line 4) defines the name of an interface that will be called, and "`inline`" (Line 5) specifies another workflow to be inlined.

4.11 Policies

Policies are non-functional behavior or quality-of-service (QoS) an entity can declare.

4.11.1 Policy Types

TOSCA YAML and TOSCA XML define Policy Type which inherit from their Entity Type Section 4.2.1.

Listing 4.37 Simplified grammar for TOSCA XML Policy Type [PS13, Section 14.1]

```
1  PolicyType:
2     [... elements inherited from Entity Type ...]
3     AppliesTo:
4        NodeTypeReference:*
5           typeRef: <string>
6     content: <any>?
```

Listing 4.38 Simplified grammar for TOSCA YAML Policy Types [TY17, Section 3.6.12.2]

```
1  PolicyType:
2     [... elements inherited from Entity Type ...]
3     targets: <string>*
4     triggers: <TriggerDefinition>*
```

The elements “`targets`” (Line 3 in Listing 4.38) and “`AppliesTo`” (Line 3 in Listing 4.37) define the Node Types, in case of TOSCA YAML additionally Node Types and Group Types, the Policy applies to.

TOSCA YAML template authors can specify the element “`triggers`” (Line 4 in Listing 4.38), a list of Trigger Definitions, which define an event, a condition or an action that triggers the Policy.

The TOSCA XML element “`content`” allows defining Policy Type specific content.

4.11.2 Policy Definition

Policy Definitions are instances of Policy Types.

Listing 4.39 Simplified grammar for TOSCA XML Policy [PS13, Section 5.1] and TOSCA XML Policy Templates [PS13, Section 15.1]

```
1  Policy:
2     name: <string>?
3     policyType: <string>
4     policyRef: <string>?
5     content: <any>?
6
7  PolicyTemplate:
8     id: <string>
9     name: <string>?
10    type: <string>
11    Properties: <Property>*
12    PropertyConstraints: <PropertyConstraints>*
13    name: <string>?
14    content: <any>?
```

Listing 4.40 Simplified grammar for TOSCA YAML Policy Definition [TY17, Section 3.7.6]

```
1  PolicyDefinition:
2     type: <string>
3     metadata:
4        entry:*
5           key: <string>
6           value: <string>
7     properties: <PropertyAssignment>
8     targets: <string>*
9     triggers: <TriggerDefinition>*
```

TOSCA XML defines "`Policy`" (Line 1 in Listing 4.39) element in the Service Template, which have a Policy Type name (Line 2 in Listing 4.39) and an optional name of a Policy Template (Line 4 in Listing 4.39). If no Policy Template is specified, the sufficient information about the Policy is provided in the "`content`" (Line 5 in Listing 4.39) element. The TOSCA XML Policy Template refers to a specific Policy Type, which defines the structure of the properties, for which the template provides values.

Besides assigning properties and metadata TOSCA YAML Policy Definition has similar elements compared to TOSCA Policy Types.

4.12 Definitions

The TOSCA Definitions Document is the top-level entity for TOSCA XML and is compared to relevant element from TOSCA Service Template, which is the top-level element for TOSCA YAML.

Listing 4.41 Simplified grammar for TOSCA XML Definitions Document [PS13, Section 4.1]

```
1  Definitions:
2     id: <string>
3     name: <string>?
4     targetNamespace: <string>
5     Extensions:
6        Extension:*
7           namespace: <string>
8           mustUnderstand: <boolean>?
9     Import: <ImportDefinition>*
10    Types: <Schema>*
11    ServiceTemplates: <ServiceTemplate>*
12    NodeTypes: <NodeType>*
13    NodeTypeImplemenations: <NodeTypeImplementation>*
14    RelationshipTypes: <RelationshipType>*
15    RelationshipTypeImplementations: <RelationshipTypeImplementations>*
16    RequirementTypes: <RequirementTypes>*
17    CapabilityTypes: <CapabilityType>*
18    ArtifactTypes: <ArtifactType>*
19    ArtifactTemplates: <ArtifactTemplate>*
20    PolicyTypes: <PolicyType>*
21    PolicyTemplates: <PolicyTemplate>*
```

Both grammars have the element "`imports`" (Line 6 in Listing 4.42 and Line 9 in Listing 4.41), which allows the importing of other Definition Documents or in case of TOSCA YAML other Service Templates.

Listing 4.42 Part 1 of simplified grammar for TOSCA YAML Service Template definition elements [TY17, Section 3.9.1]

```
1  ServiceTemplate:
2     [... elements comperable to XML Service Tempalte ...]
3     tosca_definitions_version: <string>
4     dsl_definitions: <any>
5     repositories: <RepositoryDefinition>*
6     imports: <ImportDefinition>*
7     artifact_types: <ArtifactType>*
8     data_types: <DataType>*
9     capability_types: <CapabilityType>*
10    interface_types: <InterfaceType>*
11    relationship_types: <RelationshipTypes>*
12    node_types: <NodeType>*
13    group_types: <GroupType>*
14    policy_types: <PolicyType>*
```

TOSCA YAML and TOSCA XML have several elements which represent a list of Type declarations (Line 7–14 in Listing 4.42 and Line 12–20 in Listing 4.41):

- Artifact Types: Section 4.3.1
- Capability Types: Section 4.6.1
- Node Type and Node Type Implementation: Section 4.4.1
- Relationship Types and Relationship Type Implementation: Section 4.9.1
- Requirement Types: Section 4.5.1
- Capability Types: Section 4.6.1
- Artifact Types: Section 4.3.1
- Policy Types: Section 4.11.1

TOSCA YAML defines the elements “`data_types`” (Line 8 in Listing 4.42), a list of Data Types (Section 4.16), and “`group_types`”, a list of Group Types (Section 4.17.1), while TOSCA XML defines “`Types`” (Line 10 in Listing 4.41) with XML schema.

Since TOSCA YAML only defines Service Templates as top-level entities, it does not have a sub-element “`ServiceTemplate`”. Additionally, TOSCA XML defines “`ArtifactTemplates`” (Line 19 in Listing 4.41), “`PolicyTemplates`” (Line 21 in Listing 4.41) and “`Extensions`” (Line 5 in Listing 4.41).

TOSCA YAML defines Artifact Definitions similar to Artifact Templates (Section 4.3.2) as sub-elements for TOSCA YAML Node Types (Section 4.4.1) or TOSCA YAML Node Templates (Section 4.4.2).

Extensions allow TOSCA XML template authors to define namespaces of TOSCA extension attributes and elements to be used in the Definitions Document.

TOSCA YAML defines three additional elements "`interface_types`", a list of Interface Types (Section 4.7), "`dsl_definitions`", dynamic source language definitions, and "`repositories`", a list of Repository Definition (Section 4.15).

Dynamic source language definitions allow defining reusable YAML macros for use throughout the Service Template.

4.13 Service Template

TOSCA XML defines Service Templates as elements which describe the structure and manageability behavior of cloud applications. TOSCA YAML describes a Service Template as a top-level entity that contains element definitions of building blocks or complete models of cloud applications.

This section compares the remaining elements from TOSCA YAML Service Template, which were not part of the Definitions comparison Section 4.12, with the TOSCA XML Service Template.

Listing 4.43 Simplified grammar for TOSCA XML Service Template [PS13, Section 5.1]

```
1  ServiceTemplate:
2    id: <string>
3    name: <string>?
4    targetNamespace: <string>
5    substitutableNodeType: <string>?
6    Tags: <Tag>*
7    BoundaryDefinitions:
8      Properties:
9        content: <any>
10       PropertyMappings: <PropertyMapping>
11     PropertyConstraints: <PropertyConstraint>*
12     Requirements: <Requirement>*
13     Capabilities: <Capability>*
14     Policies: <Policy>*
15     Interface: <Interface>*
16   TopologyTemplate: <TopologyTemplate>
17   Plans: <Plan>*
```

The element "`metada`" (Line 3 in Listing 4.44) is similar to "`Tags`" (Line 6 in Listing 4.43), both contain additional information.

Listing 4.44 Part 2 of simplified grammar for TOSCA YAML Service Template without the Definition Document part [TY17, Section 3.9.1]

```
1 ServiceTemplate:
2    [... elements comparable to TOSCA XML Definitions ...]
3    metadata:
4       entry:*
5          key: <string>
6          value: <string>
7    topology_template: <TopologyTemplate>?
```

The second element in common to both grammars is “`topology_template`” (Line 7 in Listing 4.44 and Line 16 in Listing 4.43), which defines the element Topology Template (Section 4.8).

The TOSCA XML Service Template simplified grammar has additional elements (Listing 4.43):

- “`substitutableNodeType`” (Line 5): specifies a Node Type that can be substituted by the Service Template.
- “`BoundaryDefinitions`” (Line 7): specifies properties the Service Template exposes beyond its boundaries. Similar to TOSCA YAML Topology Template sub-element Substitution Mapping (Line 8 in Listing 4.28).
- “`Plans`” (Line 17): Similar to TOSCA YAML Workflows (Section 4.10) specified in Topology Template.

4.14 Import Definition

Import Definitions are used to include other TOSCA YAML Service Templates or TOSCA XML Definitions into the current TOSCA YAML Service Template or TOSCA XML Definitions.

Listing 4.45 Simplified grammar for TOSCA XML Import Definition [PS13, Section 4.1]

```
1 ImportDefinition:
2    namespace: <string>?
3    location: <string>?
4    importType: <string>
```

The element “`file`” (Line 2 in Listing 4.46) is similar to “`location`” (Line 3 in Listing 4.45), both specify the file URI.

Listing 4.46 Simplified grammar for TOSCA YAML Import Definition [TY17, Section 3.5.7.2]

```
1 ImportDefiniton:
2   file: <string>
3   repository: <string>?
4   namespace_uri: <string>?
5   namespace_prefix: <string>?
```

The element “`namespace_uri`” (Line 4 in Listing 4.46 and Line 2 in Listing 4.45) specifies a namespace that identifies the Imported Definition.

TOSCA XML Import Definition uses the element “`importType`” (Line 4 in Listing 4.45), which is an URI defining the document type.

TOSCA YAML has the element “`namespace_prefix`” (Line 5 in Listing 4.46), which defines a prefix. Thus, all definitions from the imported TOSCA YAML Service Template can be referenced by adding “`<prefix>.`” to the name of the entity.

The element “`repository`” (Line 3 Listing 4.46) indicates for TOSCA YAML Import Definition, that the file is located in a repository.

4.15 Repository Definition

TOSCA YAML uses Repository Definitions to specify external repositories containing artifacts. The repositories can be referenced in TOSCA YAML Artifact Definition Section 4.10.4.

Because TOSCA XML does not have Repository Definitions this sections only presents the simplified grammar for TOSCA YAML Repository Definitions (Listing 4.47).

Listing 4.47 Simplified grammar for TOSCA YAML Repository Definition [TY17, Section 3.5.4.3]

```
1 RepositoryDefinition:
2   name: <string>
3   url: <string>
4   credential: <Credential>?
```

The element “`url`” (Line 3) specifies a required URL or network address and the element “`credential`” (Line 4) provides the Credentials required to access the repository.

4.16 Data Types

The TOSCA YAML Data Type is a sub-type of the Entity Type and allows the definition of new named data types in TOSCA YAML. TOSCA XML Types are defined as XML schema, thus each TOSCA YAML Data Type can be converted to a TOSCA XML Type.

The simplified grammar for TOSCA YAML Data Types is presented in Listing 4.48.

Listing 4.48 Simplified grammar for TOSCA YAML Data Type [TY17, Section 3.6.6.2]

```
1 DataType:
2     [... elements inherited from Entity Type ...]
3     constraints: <ConstraintClause>*
```

In addition to the elements inherited from the Entity Type, the Data Type has the element “`constraints`”, a list of Constraint Clause to restrict the Data Type.

4.16.1 Constraint Clause

Constraint Clauses restrict TOSCA YAML elements and are combination of operator and values. TOSCA XML do not need to define Constraint Clauses, since XML has this already included. Property Definitions for example are defined with XML schema, which can define valid values, max min values, and other restrictions.

The simplified grammar for TOSCA YAML Constraint Clause is presented in Listing 4.49.

Listing 4.49 Simplified grammar for TOSCA YAML Constraint Clause [TY17, Section 3.5.2.3]

```
1 ConstraintClause:
2     <operator>: <string>*
```

Each Constraint Clause has an “`operator`” (equal, greater than, greater or equal, less than, less or equal, in range, valid values, length, min length, max length, pattern) and a value, value list or expression.

4.17 Groups

Groups define the logical grouping of Node Templates, *e.g.*, for management purpose. Only TOSCA YAML supports Groups.

4.17.1 Group Types

The simplified grammar for TOSCA YAML Group Types is presented in Listing 4.50.

Listing 4.50 Simplified grammar for TOSCA YAML Group Type [TY17, Section 3.5.2.3]

```
1 GroupType:
2    [... elements inherited from Entity Type ...]
3    attributes: <AttributeDefinition>*
4    members: <string>*
5    requirements: <RequirementDefinition>*
6    capabilities: <CapabilityDefinifion>*
7    interfaces: <InterfaceDefinition>*
```

The TOSCA YAML Group Type defines “`members`” (Line 4), a list of valid Node Types that are allowed for the group, “`requirements`” (Line 5), a list of Requirement Definitions (Section 4.5.1), “`capabilities`” (Line 6), a list of Capability Definitions (Section 4.6.2), and “`interfaces`” (Line 7), a list of Interface Definition (Section 4.7).

4.17.2 Group Definition

The simplified grammar for TOSCA YAML Group Definitions is presented in Listing 4.51.

Listing 4.51 Simplified grammar for TOSCA YAML Group Definition [TY17, Section 3.7.5.2]

```
1 GroupDefinition:
2    type: <string>
3    metadata:
4       entry:*
5          key: <string>
6          value: <string>
7    properties: <PropertyAssignment>*
8    members: <string>*
9    interfaces: <InterfaceDefinition>*
```

The TOSCA YAML Group Definition specifies “`members`” (Line 8), names of Node Templates Section 4.4.2, and “`interfaces`” (Line 9), a list of Interface Definitions Section 4.7.

4.18 Property Definitions

Property Definitions describe named, typed values and other data that can be associated with Entity Types and other TOSCA elements. The simplified grammar for TOSCA YAML Property Definitions is presented in Listing 4.52. TOSCA XML Property Definitions reference an XML element or XML type, which is defining the structure of the property.

Listing 4.52 Simplified grammar for TOSCA YAML Property Definition [TY17, Section 3.5.8.4]

```
1 PropertyDefinition:
2     type: <string>
3     required: <boolean>?
4     default: <string>?
5     status: <StatusValue>?
6     constraints: <ConstraintClause>*
7     entry_schema: <EntrySchema>?
```

The elements supported for TOSCA YAML Property Definitions are:

- "`type`" (Line 2): Defines the required Data Type (Section 4.16).
- "`required`" (Line 3): Specifies if the property is required.
- "`default`" (Line 4): Specifies the default value for the property.
- "`status`" (Line 5): Specifies the status of the property. Supported values are "supported", "unsupported" and "experimental".
- "`status`" (Line 6): Specifies Constraints Clauses for the property (Section 4.16.1).
- "`entry_schema`" (Line 7): Specifies the name of the Data Type Definition for entries of set types.

4.18.1 Property Assignments

Property Assignments are used to assign values to properties. TOSCA XML does this by assigning XML fragments to properties and the simplified grammar for TOSCA YAML Property Assignemnt is presented in Listing 4.53.

The element "`name`" (Line 2) specifies the name of the property and "`value`" (Line 3) specifies the type compatible value, which might be the result of an evaluation, to assign to the property.

Listing 4.53 Simplified grammar for TOSCA YAML Property Assignment [TY17, Section 3.5.9.2.1]

```
1 PropertyAssignemnt:
2    name: <string>
3    value: <string>
```

4.18.2 Attributes

TOSCA YAML additionally specifies the elements Attribute Definition and Attribute Assignment.

Listing 4.54 Simplified grammar for TOSCA YAML Attribute Definition [TY17, Section 3.5.10.3]

```
1 AttributeDefinition:
2    type: <string>
3    default: <string>?
4    status: <StatusValue>?
5    entry_schema: <EntrySchema>?
```

Compared to TOSCA YAML Properties, TOSCA YAML Attributes expose the actual state of some properties after the TOSCA entity has been deployed. The TOSCA YAML orchestrator creates Attribute Definition for any Property Definitions, which makes their values accessible [TY17, Section 3.5.10.1].

The simplified grammar of TOSCA YAML Property Definitions (Listing 4.52) and TOSCA YAML Attribute Definitions (Listing 4.54) are similar, except the fact that TOSCA YAML Attribute Definitions do not support the keynames "`required`" (Line 3 in Listing 4.52) and "`constraints`" (Line 6 in Listing 4.52).

Listing 4.55 Simplified grammar for TOSCA YAML Attribute Assignment [TY17, Section 3.5.11.2.1]

```
1 AttributeAssignemnt:
2    name: <string>
3    value: <string>
```

The TOSCA YAML Attribute Assignment (Listing 4.55) is identical to TOSCA YAML Property Assignment (Listing 4.53).

4.18.3 Parameter Definition

TOSCA YAML Parameter Definitions (Listing 4.56) are TOSCA YAML Property Definitions (Listing 4.52) that have an additional element "`value`" (Line 3 in Listing 4.56). Thus,

a combination of a TOSCA YAML Property Definition and a TOSCA YAML Property Assignment.

Listing 4.56 Simplified grammar for TOSCA YAML Parameter Definition [TY17, Section 3.5.12.2]

```
1 ParameterDefinition:
2   [... elements inherited from Property Definition ...]
3   value: <string>
```

4.19 Node Filter Definition and Property Filter Definition

A Node Filter Definitions (Listing 4.57) specifies a list of Property Filter Definitions and a list of Capability Definition (Section 4.6.2) names together with Capability Type (Section 4.6.1) names. Based upon this information a TOSCA YAML Node Template is selected [TY17, Section 3.5.4].

Listing 4.57 Simplified grammar for TOSCA YAML Node Filter Definition [TY17, Section 3.5.4.3]

```
1 NodeFilterDefinition:
2   properties: <PropertyFilterDefinition>*
3   capabilities: <string>*
```

The Property Filter Definition (Listing 4.58) specifies a list of Constraint Clauses Section 4.16.1 for filtering TOSCA YAML entities based on their properties [TY17, Section 3.5.3].

Listing 4.58 Simplified grammar for TOSCA YAML Property Filter Definition [TY17, Section 3.5.3.1.2]

```
1 PropertyFilterDefinition:
2   name: <string>
3   value: <ConstraintClause>*
```

4.20 Additional Specification Information

Besides information about the Main TOSCA Components, the TOSCA YAML and the TOSCA XML specification define additional information. This information, which is no discussed further, ranges from Language Design to Core Concepts, Use Cases, additional examples, Cloud Service Archive, TOSCA Networking, TOSCA Policies, Security Considerations and Conformance.

5 Mapping of Specification

This section presents a mapping between the TOSCA XML and TOSCA YAML specification and the sections of Chapter 4. Each section in the specification is presented with a page range and a short description.

Page	Description	Section
1–11	Introduction of TOSCA YAML	Section 4.20
12–43	TOSCA by example	Section 4.20
18–56	TOSCA Simple Profile definitions in YAML	Section 4.20
56–58	Constraint clause	Section 4.16.1
58–59	Property Filter definition	Section 4.19
59–60	Node Filter definition	Section 4.19
60–61	Repository definition	Section 4.15
61–62	Artifact definition	Section 4.3.2
63–64	Import definition	Section 4.14
64–66	Property definition	Section 4.18
66–66	Property Assignment	Section 4.18.1
66–68	Attribute definition	Section 4.18.2
68–69	Attribute assignment	Section 4.18.2
69–70	Parameter definition	Section 4.18.3
70–73	Operation definition	Section 4.7
73–74	Interface definition	Section 4.7
74–75	Event Filter definition	Section 4.19
75–76	Trigger definition	Section 4.11.1
76–77	Workflow activity definition	Section 4.10.4
77–78	Assertion definition	Section 4.10
78–80	Condition clause definition	Section 4.16.1

Table 5.1: Mapping between TOSCA YAML specification [TY17] and sections in Chapter 4 (Part 1)

Page	Description	Section
80–80	Workflow precondition definition	Section 4.10.2
80–82	Workflow step definition	Section 4.10.3
82-82	Entity Type Schema	Section 4.2.1
83-84	Capability definition	Section 4.6.2
84-87	Requirement definition	Section 4.5.1
87-88	Artifact Type	Section 4.3.1
88-89	Interface Type	Section 4.7
89-90	Data Type	Section 4.16
91-92	Capability Type	Section 4.6.1
92-92	Requirement Type	Section 4.5.1
92-94	Node Type	Section 4.4.1
94-95	Relationship Type	Section 4.9.1
95-97	Group Type	Section 4.17.1
97-98	Policy Type	Section 4.11.1
98-99	Capability assignment	Section 4.6.3
99-103	Requirement assignment	Section 4.5.2
103-105	Node Template	Section 4.4.2
105-107	Relationship Template	Section 4.9.2
107-108	Group Definition	Section 4.17.2
108-109	Policy Definition	Section 4.11.2
109-110	Imperative Workflow definition	Section 4.10.1
110-115	Topology Template definition	Section 4.8
115-126	Service Template definition	Section 4.13
126-138	TOSCA functions	Section 4.20
139-180	TOSCA normative type definitions	Section 4.20
181-182	TOSCA Cloud Service Archive (CSAR) format	Section 4.20
183-201	TOSCA workflows	Section 4.20
202-218	TOSCA networking	Section 4.20
219-223	Non-normative type definitions	Section 4.20
224-230	Component Modeling Use Cases	Section 4.20
231-270	Application Modeling Use Cases	Section 4.20
271-275	TOSCA Policies	Section 4.20
276-277	Conformance	Section 4.20
277-282	Appendix	Section 4.20

Table 5.2: Mapping between TOSCA YAML specification [TY17] and sections in Chapter 4 (Part 2)

Page	Description	Section
1–7	Introduction	Section 4.20
8–10	Language Design	Section 4.20
11–17	Core Concepts	Section 4.20
18–22	TOSCA Definitions Document	Section 4.12
23–38	Service Templates	Section 4.13
39–44	Node Types	Section 4.4.1
45–49	Node Type Implementation	Section 4.4.1
50–53	Relationship Types	Section 4.9.1
54–57	Relationship Type Implementation	Section 4.9.1
58–60	Requirement Types	Section 4.5.1
61–63	Capability Types	Section 4.6.1
64–66	Artifact Types	Section 4.3.1
67–69	Artifact Template	Section 4.3.2
70–72	Policy Types	Section 4.11.1
73–74	Policy Templates	Section 4.11.2
75–79	Cloud Service Archive (CSAR)	Section 4.20
80–80	Security Considerations	Section 4.20
81–81	Conformance	Section 4.20
82–114	Appendix	Section 4.20

Table 5.3: Mapping between TOSCA XML specification [PS13] and sections in Chapter 4

6 Implementation

The implementation part of this master thesis was to create a sub module for Winery [ESF17a], which allows the conversion between TOSCA YAML and TOSCA XML elements.

The three main parts of the implementaiton were to create

1. A data model for TOSCA YAML
2. A translator from TOSCA YAML to TOSCA XML
3. A translator from TOSCA XML to TOSCA YAML

A data model for TOSCA XML has already been established as an XML schema ([PS13, Appendix D. TOSCA Schema]).

Since TOSCA YAML has no XML Schema, the informations for the data model must be extracted from the specifications, *i.e.* from the keynames tables and the grammar, which exists for each TOSCA element (E.g. [TY17, Section 3.9.1,3.9.2]).

The first step to a data model was to converted the specification to an UML diagram [ESF17b]. The diagram models the structure between the TOSCA YAML elements and the available properties.

The following list shows some UML diagram design decisions:

- The keyname “type” is always given as a string, with additional information from the description in the keyname table the corresponding TOSCA element type can be used.
- The keyname “description” is represented as a string.
- Keynames having other TOSCA elements as Types are modeled as directed named association.
- Types inherit from Entity Type.

Keyname	Required	Type	Description
derived_from	no	string	An optional parent Policy Type . . .
description	no	description	The optional . . .
version	no	version	An optional version for . . .
properties	no	list of property definitions	An optiona list of . . .
targets	no	string[]	An optional list of valid Node Types or Group Types . . .

Table 6.1: Part of the Policy Type keyname table [TY16, Section 3.6.11]

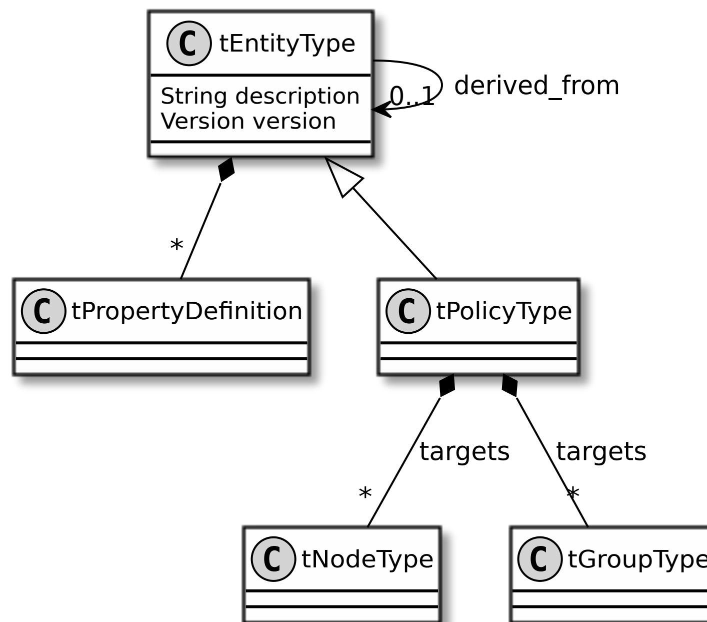

Figure 6.1: Part of UML diagram for Policy Type and Entity Type

Exemplary, the conversion from the Policy Type keyname table (Table 6.1) to UML (Figure 6.1) is shown, for brevity the Property Definition, Node Type and Group Type are not complete.

With the knowledge from the UML diagram and with respect to the YAML parser "SnakeYAML" [Som17], TOSCA YAML Java classes and an XML schema were created.

SnakeYAML is a complete YAML 1.1 processor for Java. Basically, the processor takes a YAML file and a Java class as input and constructs an instance of the Java class.

The TOSCA YAML specification contains several cases where a direct mapping between the specified grammar and a Java class is difficult. Listing 6.1 is such a case: Usually, a list of elements is specified by the property "`inputs`" (Line 3) and the Java class would contain a field "`Map<String,TPropertyDefinition>inputs;`", where the key specifies the name of the Property and the value the Property Definition.

The element "`operation_definitions`" (Line 5) is a set of key value Entries, which is not grouped for example by a keyname "`operations`". Thus, an operation with the name "`inputs`" or "`type`" would be erroneous, because line 2 and line 4 already contain those keys for Interface Definitions.

Listing 6.1 Original grammar for Interface Definitions [TY16, Section 3.5.14.2.1]

```
1 <interface_definition_name>:
2    type: <interface_type_name>
3    inputs:
4       <property_definitions>
5    <operation_definitions>
```

This issue is solved by creating a grouping field "`Map<String,TOperationDefinition>operations;`" for Interface Definitions and telling SnakeYAML to put Operations into the field.

The implementation has defined personalized procedures, which extend SnakeYaml. Thus, all YAML Service Templates files conforming to the TOSCA YAML specification will be accepted by the implementation.

A data model representing the TOSCA YAML specification exactly, would have to use the Java super type "`Object`" for some TOSCA YAML elements, e.g. no direct Java class representation for Interface Definitions is possible.

The final data model for TOSCA YAML is given as an XML Schema and as a set of Java classes.

The following list shows general design decision:

- Arrays will be represented as Lists.
- Elements without any keynames, for example Assignments, get a keyname "`ObjectValue value;`".
- Properties without a type have the "`ObjectValue`" type.

- The keyname status, which has a simple string type, which is limited to several values, gets the new type “`Enum StatusValue`”.
- List of named elements are converted to “`Map<String,Element>`”.
- Range of type T is converted to “`List<T>`”.
- List of List or Map of List are converted to List of Class A or Map of Class A, where a denotes a unique name: for example “`constraints: -<type_constraints>`” is a list of sequenced constraints. SnakeYAML interprets it as a “`List<List<Constraints>>`”, which causes problem when creating a schema. Thus, an Class named “`SequencedConstraints`” is used: “`List<SequencedConstraints>`”.
- If the specification contains several grammars for one Element, for example a short notation grammar “`<requirement_name>: <node_template_name>`” versus an extended notation grammar with Relationship, Capability, Node Filter and occurrences information, than both notations will be combined into one Class.

After creating the data model, TOSCA YAML Service Templates files can be deserialized to the data model. The conversion between the generated data model for TOSCA YAML and the given data model for TOSCA XML is done as a recursive descend, starting from the top level Service Template class instance. The conversion from TOSCA XML to TOSCA YAML starts with an instance of the TOSCA XML Definitions class and constructs an instance of a TOSCA YAML Service Template class.

What is needed for the mapping between TOSCA YAML and TOSCA XML elements has been described in Chapter 4.

The complete structure of the winery module created as part of the master thesis is presented in Figure 6.2.

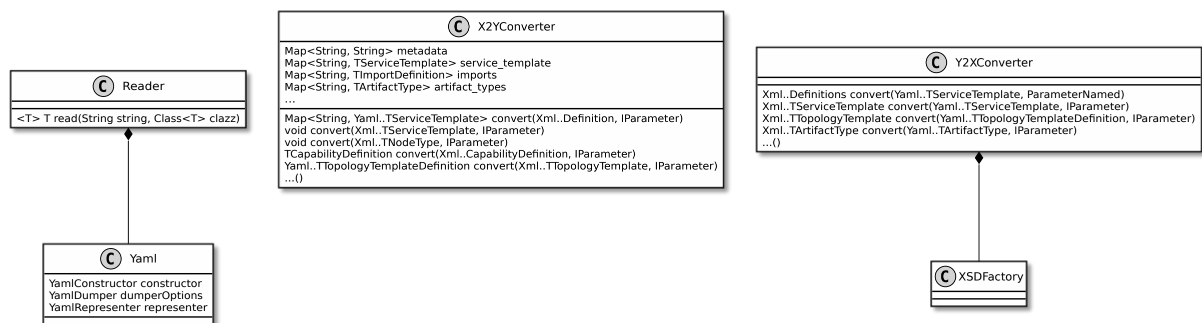

Figure 6.2: UML class diagram for winery module

The classes are “`Y2XConverter`”, a TOSCA YAML to TOSCA XML converter, “`X2YConverter`”, a TOSCA XML to TOSCA YAML converter, “`Reader`”, a TOSCA YAML file reader using the personalized SnakeYAML class “`Yaml`”, and “`XSDFactory`”, which converts TOSCA YAML

Property Definitions to XML Schema. Both converter have for each TOSCA element a method "`S convert(T,IParameter)`", which handles the conversion from TOSCA YAML to TOSCA XML or TOSCA XML to TOSCA YAML.

The converter was tested with the default TOSCA types specified in Parser for TOSCA Simple Profile in YAML [Ope17a] document. This consist of Node Types, Relationship Types, Capability Types, Interfaces, Data Types, Artifact Types, Policy Types and Group Types.

7 Summary and Outlook

This thesis started with the look at tosca converters and found the YANG to TOSCA converter and the Heat translator. Both projects can convert only a small part of TOSCA elements either from YANG to TOSCA YAML or from TOSCA YAML to Heat templates. A converter between TOSCA YAML and TOSCA XML, which takes the complete set of TOSCA YAML elements and converts them into TOSCA XML elements has previously not been specified nor programmed yet.

A converter for the complete set of Tosca YAML elements into corresponding XML was, therefore, to be programmed.

When specifying the converter initially, the specification languages were compared.

- Since XML is well-known, this thesis extensively describes the specification language YAML and compares YAML to XML.
- An YAML file data structure is then represented as lists, mappings and scalars combined with directives.
- Well-formed XML files can be abstractly presented as an XML Infoset which consist of eleven information items.
- Each YAML document can be presented as an XML document and vice versa. Either by converting lists, mappings, scalars and directives to the equivalent XML elements, sequences and namespaces, or by representing the Infoset tree with all properties as YAML lists, mappings, and scalars.

The structure of the TOSCA YAML specification differs from the structure of the TOSCA XML specification. Thus, a grammar representing TOSCA YAML and TOSCA XML elements has been established and used for the comparison. With this new grammar the complete set of TOSCA elements was compared.

TOSCA YAML elements can have different names compared to TOSCA XML elements, they can have additional elements and elements from TOSCA XML can be mising for TOSCA YAML or placed at a different position in the Service Template or Definition document.

Since TOSCA YAML is based on YAML results in additional TOSCA YAML elements, for example a TOSCA YAML Property Definition has a set of sub-elements, while TOSCA XML uses XML features to specify properties as instances of XML schema definitions.

The mapping between the specification and sections of this thesis shows where to find the TOSCA elements in this thesis and that all main TOSCA elements are covered.

The implementation part describes the development of a data structure for TOSCA YAML, which is specified as UML diagram, XML schema and Java classes. Additionally, the Java classes, which represent TOSCA YAML elements, are converted to those Java classes specified in TOSCA XML schema and vice versa. The implementation code is available as a sub module for the Winery [ESF17a].

Outlook

In the actual state the sub-module for the Winery offers functions to convert between TOSCA YAML and TOSCA XML elements. For further extensions the data model specified for TOSCA YAML can be used to specify TOSCA YAML elements in the Winery. This allows the specification of features that are not directly implemented in TOSCA XML.

Another possibility is to extend the TOSCA XML data model to a version which combines the actual TOSCA YAML data models as well as the TOSCA XML data models.

The TOSCA YAML specification has changed from TOSCA Simple Profile in YAML version 1.0 to version 1.1. The major changes were adding Workflows, which were not part of version 1.0, and Entity Types as a super types. Future versions of TOSCA YAML for example could specify other super types or extend these elements.

Bibliography

[BBK+14] U. Breitenbücher, T. Binz, K. Kepes, O. Kopp, F. Leymann, J. Wettinger. "Combining Declarative and Imperative Cloud Application Provisioning Based on TOSCA." In: *IC2E*. IEEE Computer Society, 2014, pp. 87–96. ISBN: 978-1-4799-3766-0 (cit. on p. 45).

[BEA03] BEA and IBM and Microsoft and SAP AG and Siebel. *Business Process Execution Language (BPEL)*. 2003 (cit. on p. 45).

[BEI05] O. Ben-Kiki, C. Evans, B. Ingerson. *Language-Independent Types for YAML™ Version 1.1*. Jan. 18, 2005. URL: http://yaml.org/type/ (cit. on p. 20).

[BEI06] O. Ben-Kiki, C. Evans, B. Ingerson. *YAXML, the (draft) XML Binding for YAML*. Tech. rep. 2006. URL: http://yaml.org/xml/ (cit. on p. 22).

[CT04] J. Cowan, R. Tobin. *XML Information Set (Second Edition)*. Tech. rep. W3C, 2004. URL: http://www.w3.org/TR/2004/REC-xml-infoset-20040204/ (cit. on p. 21).

[EBBS11] R. Enns, M. Bjorklund, A. Bierman, J. Schönwälder. *Network Configuration Protocol (NETCONF)*. RFC 6241. June 2011. DOI: 10.17487/rfc6241. URL: https://rfc-editor.org/rfc/rfc6241.txt (cit. on p. 15).

[ESF17a] Eclipse Software Foundation. *Eclipse Winery*. 2017. URL: http://eclipse.org/winery/ (cit. on pp. 13, 63, 70).

[ESF17b] Eclipse Software Foundation. *Eclipse Winery Uml diagram*. May 3, 2017. URL: https://github.com/eclipse/winery/tree/master/docs/TOSCA%20specification (cit. on p. 63).

[FB96] N. Freed, N. Borenstein. *RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*. Internet Engineering Task Force. Nov. 1996. URL: http://www.ietf.org/rfc/rfc2045.txt (cit. on p. 29).

[Gud02] M. Gudgin. *Understanding Infosets*. 2002. URL: https://msdn.microsoft.com/en-us/library/aa468561.aspx#xmlinfoset_topic7 (cit. on p. 22).

[Ley09] F. Leymann. “Cloud Computing: The Next Revolution in IT.” In: *Proc. 52th Photogrammetric Week*. Wichmann Verlag, Sept. 2009, pp. 3–12. ISBN: 978-3-87907-483-9 (cit. on p. 13).

[OAS17] OASIS Organiszation. *OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC FAQ*. 2017. URL: https://www.oasis-open.org/committees/tosca/faq.php (cit. on p. 13).

[OMG11] OMG. *Business Process Model and Notation (BPMN), Version 2.0*. Object Management Group, Jan. 2011. URL: http://www.omg.org/spec/BPMN/2.0 (cit. on p. 45).

[Ope16a] OpenStack Organisation. *Heat-Translator*. Aug. 9, 2016. URL: https://github.com/openstack/heat-translator (cit. on p. 15).

[Ope16b] OpenStack Organisation. *Heat-Translator (TOSCA NodeTypes)*. Aug. 9, 2016. URL: https://github.com/openstack/heat-translator/tree/master/translator/hot/tosca (cit. on p. 16).

[Ope17a] Openstack Foundation. *Parser for TOSCA Simple Profile in YAML - TOSCA Types*. Mar. 12, 2017. URL: https://github.com/openstack/tosca-parser/blob/master/toscaparser/elements/TOSCA_definition_1_0.yaml (cit. on p. 67).

[Ope17b] OpenStack Organisation. *TOSCA Parser*. Apr. 10, 2017. URL: https://github.com/openstack/tosca-parser (cit. on p. 17).

[PS13] D. Palma, T. Spatzier, eds. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. Nov. 25, 2013. OASIS Standard. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html (cit. on pp. 13, 23, 25, 26, 29, 30, 32, 33, 35–41, 43, 44, 47–49, 51, 52, 61, 63).

[Som17] A. Somov. *SnakeYAML v1.18*. Feb. 22, 2017. URL: https://bitbucket.org/asomov/snakeyaml (cit. on p. 65).

[Tab06] P. Tabeling. *Softwaresysteme und ihre Modellierung*. 2006 (cit. on p. 25).

[TY16] D. Palma, M. Rutkowski, T. Spatzier, eds. *TOSCA Simple Profile in YAML Version 1.0*. Aug. 17, 2016. URL: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/csprd02/TOSCA-Simple-Profile-YAML-v1.0-csprd02.html. OASIS Committee Specification Draft 02 / Public Review Draft 02. (Cit. on pp. 13, 64, 65).

[TY17] M. Rutkowski, L. Boutier, eds. *TOSCA Simple Profile in YAML Version 1.1*. Jan. 12, 2017. OASIS Committee Specification Draft 02 / Public Review Draft 02. (Cit. on pp. 23, 26–30, 32–35, 37–41, 43–48, 50, 52–60, 63).

[YamlSpec09] C. E. Oren Ben-Kiki, I. döt Net, eds. *YAML Ain't Markup Language (YAML™) Version 1.2*. Oct. 1, 2009. URL: http://yaml.org/spec/1.2/spec.pdfhtml (cit. on pp. 19, 20).

[yttc16] K. Ilyashenko, ed. *yttc - YANG to TOSCA converter*. 2016. URL: https://github.com/cloudify-cosmo/yttc (cit. on p. 15).

All links were last followed on April 3, 2017.

Declaration

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

---

place, date, signature