



Topology and Orchestration Specification for Cloud Applications Version 1.0

Committee Specification Draft 07

18 March 2013

Specification URIs

This version:

<http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd07/TOSCA-v1.0-csd07.pdf> (Authoritative)
<http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd07/TOSCA-v1.0-csd07.html>
<http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd07/TOSCA-v1.0-csd07.doc>

Previous version:

<http://docs.oasis-open.org/tosca/TOSCA/v1.0/csprd01/TOSCA-v1.0-csprd01.pdf> (Authoritative)
<http://docs.oasis-open.org/tosca/TOSCA/v1.0/csprd01/TOSCA-v1.0-csprd01.html>
<http://docs.oasis-open.org/tosca/TOSCA/v1.0/csprd01/TOSCA-v1.0-csprd01.doc>

Latest version:

<http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf> (Authoritative)
<http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html>
<http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.doc>

Technical Committee:

OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC

Chairs:

Paul Lipton (paul.lipton@ca.com), CA Technologies
Simon Moser (smoser@de.ibm.com), IBM

Editors:

Derek Palma (dpalma@vnomnic.com), Vnomic
Thomas Spatzier (thomas.spatzier@de.ibm.com), IBM

Additional artifacts:

This prose specification is one component of a Work Product which also includes:

- XML schema: <http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd07/schemas/>

Declared XML namespace:

- <http://docs.oasis-open.org/tosca/ns/2011/12>

Abstract:

The concept of a “service template” is used to specify the “topology” (or structure) and “orchestration” (or invocation of management behavior) of IT services. Typically, services are provisioned in an IT infrastructure and their management behavior must be orchestrated in accordance with constraints or policies from there on, for example in order to achieve service level objectives.

This specification introduces the formal description of Service Templates, including their structure, properties, and behavior.

Status:

This document was last revised or approved by the OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC on the above date. The level of approval is also listed above. Check the "Latest version" location noted above for possible later revisions of this document.

Technical Committee members should send comments on this specification to the Technical Committee's email list. Others should send comments to the Technical Committee by using the "Send A Comment" button on the Technical Committee's web page at <http://www.oasis-open.org/committees/tosca/>.

For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the Technical Committee web page (<http://www.oasis-open.org/committees/tosca/ipr.php>).

Citation format:

When referencing this specification the following citation format should be used:

[TOSCA-v1.0]

Topology and Orchestration Specification for Cloud Applications Version 1.0. 18 March 2013. OASIS Committee Specification Draft 07. <http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd07/TOSCA-v1.0-csd07.html>.

Notices

Copyright © OASIS Open 2013. All Rights Reserved.

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full [Policy](#) may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Committee Specification or OASIS Standard, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification.

OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this specification by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification. OASIS may include such claims on its website, but disclaims any obligation to do so.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Committee Specification or OASIS Standard, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

The name "OASIS" is a trademark of [OASIS](#), the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see <http://www.oasis-open.org/policies-guidelines/trademark> for above guidance.

Table of Contents

1	Introduction	7
2	Language Design	8
2.1	Dependencies on Other Specifications	8
2.2	Notational Conventions	8
2.3	Normative References	8
2.4	Non-Normative References	8
2.5	Typographical Conventions	9
2.6	Namespaces	9
2.7	Language Extensibility	10
3	Core Concepts and Usage Pattern	11
3.1	Core Concepts	11
3.2	Use Cases	12
3.2.1	Services as Marketable Entities	12
3.2.2	Portability of Service Templates	13
3.2.3	Service Composition	13
3.2.4	Relation to Virtual Images	13
3.3	Service Templates and Artifacts	13
3.4	Requirements and Capabilities	14
3.5	Composition of Service Templates	15
3.6	Policies in TOSCA	15
3.7	Archive Format for Cloud Applications	16
4	The TOSCA Definitions Document	18
4.1	XML Syntax	18
4.2	Properties	19
4.3	Example	22
5	Service Templates	23
5.1	XML Syntax	23
5.2	Properties	26
5.3	Example	37
6	Node Types	39
6.1	XML Syntax	39
6.2	Properties	40
6.3	Derivation Rules	43
6.4	Example	43
7	Node Type Implementations	45
7.1	XML Syntax	45
7.2	Properties	46
7.3	Derivation Rules	48
7.4	Example	49
8	Relationship Types	50
8.1	XML Syntax	50
8.2	Properties	51
8.3	Derivation Rules	52

8.4 Example	53
9 Relationship Type Implementations	54
9.1 XML Syntax.....	54
9.2 Properties.....	54
9.3 Derivation Rules	56
9.4 Example	57
10 Requirement Types	58
10.1 XML Syntax	58
10.2 Properties.....	58
10.3 Derivation Rules	59
10.4 Example	60
11 Capability Types	61
11.1 XML Syntax	61
11.2 Properties.....	61
11.3 Derivation Rules	62
11.4 Example	62
12 Artifact Types.....	64
12.1 XML Syntax	64
12.2 Properties.....	64
12.3 Derivation Rules	65
12.4 Example	65
13 Artifact Templates.....	67
13.1 XML Syntax	67
13.2 Properties.....	67
13.3 Example	69
14 Policy Types	70
14.1 XML Syntax	70
14.2 Properties.....	70
14.3 Derivation Rules	71
14.4 Example	72
15 Policy Templates	73
15.1 XML Syntax	73
15.2 Properties.....	73
15.3 Example	74
16 Cloud Service Archive (CSAR).....	75
16.1 Overall Structure of a CSAR.....	75
16.2 TOSCA Meta File.....	75
16.3 Example	76
17 Security Considerations	80
18 Conformance	81
Appendix A. Portability and Interoperability Considerations	82
Appendix B. Acknowledgements	83
Appendix C. Complete TOSCA Grammar	85
Appendix D. TOSCA Schema.....	93
Appendix E. Sample	109

E.1 Sample Service Topology Definition	109
Appendix F. Revision History	112

1 Introduction

2 Cloud computing can become more valuable if the semi-automatic creation and management of
3 application layer services can be ported across alternative cloud implementation environments so that the
4 services remain interoperable. This core TOSCA specification provides a language to describe service
5 components and their relationships using a *service topology*, and it provides for describing the
6 management procedures that create or modify services using *orchestration processes*. The combination
7 of topology and orchestration in a *Service Template* describes what is needed to be preserved across
8 deployments in different environments to enable interoperable deployment of cloud services and their
9 management throughout the complete lifecycle (e.g. scaling, patching, monitoring, etc.) when the
10 applications are ported over alternative cloud environments.

11 2 Language Design

12 The TOSCA language introduces a grammar for describing service templates by means of Topology
13 Templates and plans. The focus is on design time aspects, i.e. the description of services to ensure their
14 exchange. Runtime aspects are addressed by providing a container for specifying models of plans which
15 support the management of instances of services.

16 The language provides an extension mechanism that can be used to extend the definitions with additional
17 vendor-specific or domain-specific information.

18 2.1 Dependencies on Other Specifications

19 TOSCA utilizes the following specifications:

- 20 • XML Schema 1.0

21 2.2 Notational Conventions

22 The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD
23 NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described
24 in [RFC2119].

25 This specification follows XML naming and design rules as described in **Error! Reference source not
26 found.**, i.e. uses upper camel-case notation for XML element names and lower camel-case notation for
27 XML attribute names.

28 2.3 Normative References

- 29 **[RFC2119]** S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*,
30 <http://www.ietf.org/rfc/rfc2119.txt>, IETF RFC 2119, March 1997.
- 31 **[RFC 2396]** Uniform Resource Identifiers (URI): Generic Syntax, RFC 2396, available via
32 <http://www.faqs.org/rfcs/rfc2396.html>
- 33 **[XML Base]** XML Base (Second Edition), W3C Recommendation,
34 <http://www.w3.org/TR/xmlbase/>
- 35 **[XML Infoset]** XML Information Set, W3C Recommendation, <http://www.w3.org/TR/2001/REC-xml-infoset-20011024/>
- 37 **[XML Namespaces]** Namespaces in XML 1.0 (Second Edition), W3C Recommendation,
38 <http://www.w3.org/TR/REC-xml-names/>
- 39 **[XML Schema Part 1]** XML Schema Part 1: Structures, W3C Recommendation, October 2004,
40 <http://www.w3.org/TR/xmlschema-1/>
- 41 **[XML Schema Part 2]** XML Schema Part 2: Datatypes, W3C Recommendation, October 2004,
42 <http://www.w3.org/TR/xmlschema-2/>
- 43 **[XMLSpec]** XML Specification, W3C Recommendation, February 1998,
44 <http://www.w3.org/TR/1998/REC-xml-19980210>
- 45

46 2.4 Non-Normative References

- 47 **[BPEL 2.0]** *Web Services Business Process Execution Language Version 2.0*. OASIS
48 Standard. 11 April 2007. <http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html>.
- 49 **[BPMN 2.0]** OMG Business Process Model and Notation (BPMN) Version 2.0,
50 <http://www.omg.org/spec/BPMN/2.0/>
- 51 **[OVF]** Open Virtualization Format Specification Version 1.1.0,
52 http://www.dmtf.org/standards/published_documents/DSP0243_1.1.0.pdf

53 **[XPath 1.0]** XML Path Language (XPath) Version 1.0, W3C Recommendation, November
 54 1999, <http://www.w3.org/TR/1999/REC-xpath-19991116>
 55 **[UNCEFACT XMLNDR]** UN/CEFACT XML Naming and Design Rules Technical Specification,
 56 Version 3.0,
 57 [http://www.unece.org/fileadmin/DAM/cefact/xml/UNCEFACT+XML+NDR+V3p0.p](http://www.unece.org/fileadmin/DAM/cefact/xml/UNCEFACT+XML+NDR+V3p0.pdf)
 58 [df](http://www.unece.org/fileadmin/DAM/cefact/xml/UNCEFACT+XML+NDR+V3p0.pdf)
 59

60 2.5 Typographical Conventions

61 This specification uses the following conventions inside tables describing the resource data model:

- 62 • Resource names, and any other name that is usable as a type (i.e., names of embedded
 63 structures as well as atomic types such as "integer", "string"), are in italic.
- 64 • Attribute names are in regular font.

65 In addition, this specification uses the following syntax to define the serialization of resources:

- 66 • Values in italics indicate data types instead of literal values.
- 67 • Characters are appended to items to indicate cardinality:
 - 68 ○ "?" (0 or 1)
 - 69 ○ "*" (0 or more)
 - 70 ○ "+" (1 or more)
- 71 • Vertical bars, "|", denote choice. For example, "a|b" means a choice between "a" and "b".
- 72 • Parentheses, "(" and ")", are used to indicate the scope of the operators "?", "*", "+" and "|".
- 73 • Ellipses (i.e., "...") indicate points of extensibility. Note that the lack of an ellipses does not mean
 74 no extensibility point exists, rather it is just not explicitly called out - usually for the sake of brevity.

75 2.6 Namespaces

76 This specification uses a number of namespace prefixes throughout; they are listed in Table 1. Note that
 77 the choice of any namespace prefix is arbitrary and not semantically significant (see [XML Namespaces]).
 78 Furthermore, the namespace <http://docs.oasis-open.org/tosca/ns/2011/12> is assumed to be the default
 79 namespace, i.e. the corresponding namespace name `ste` is omitted in this specification to improve
 80 readability.

81

Prefix	Namespace
tosca	http://docs.oasis-open.org/tosca/ns/2011/12
xs	http://www.w3.org/2001/XMLSchema

82 Table 1: Prefixes and namespaces used in this specification

83

84 All information items defined by TOSCA are identified by one of the XML namespace URIs above [XML
 85 Namespaces]. A normative XML Schema ([XML Schema Part 1][XML Schema Part 2]) document for
 86 TOSCA can be obtained by dereferencing one of the XML namespace URIs.

87 **2.7 Language Extensibility**

88 The TOSCA extensibility mechanism allows:

- 89 • Attributes from other namespaces to appear on any TOSCA element
- 90 • Elements from other namespaces to appear within TOSCA elements
- 91 • Extension attributes and extension elements **MUST NOT** contradict the semantics of any attribute
- 92 or element from the TOSCA namespace

93 The specification differentiates between mandatory and optional extensions (the section below explains
94 the syntax used to declare extensions). If a mandatory extension is used, a compliant implementation
95 **MUST** understand the extension. If an optional extension is used, a compliant implementation **MAY**
96 ignore the extension.

97 3 Core Concepts and Usage Pattern

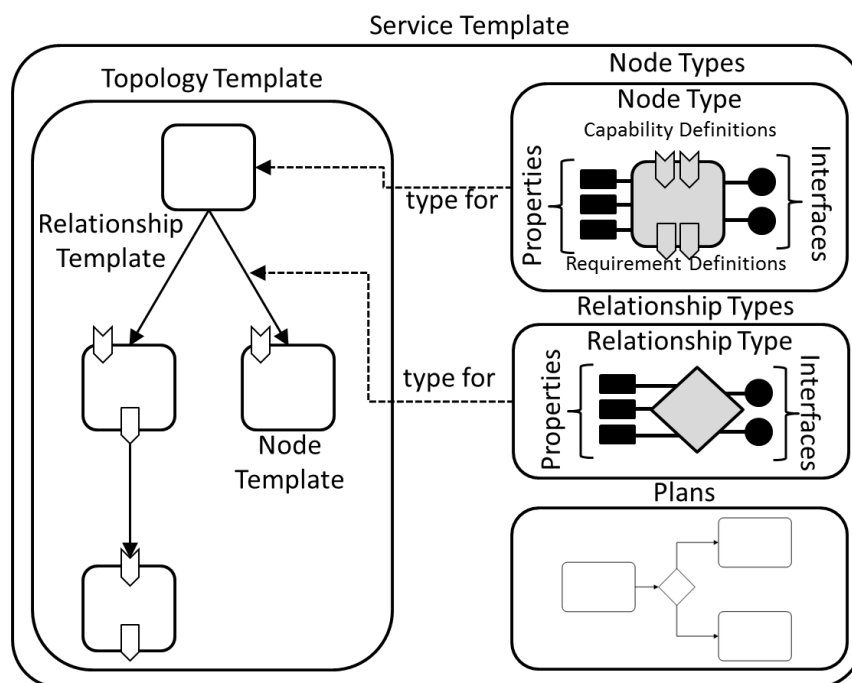
98 The main concepts behind TOSCA are described and some usage patterns of Service Templates are
99 sketched.

100 3.1 Core Concepts

101 This specification defines a *metamodel* for defining IT services. This metamodel defines both the
102 structure of a service as well as how to manage it. A *Topology Template* (also referred to as the *topology*
103 *model* of a service) defines the *structure* of a service. *Plans* define the process models that are used to
104 create and terminate a service as well as to manage a service during its whole lifetime. The major
105 elements defining a service are depicted in Figure 1.

106
107 A Topology Template consists of a set of Node Templates and Relationship Templates that together
108 define the topology model of a service as a (not necessarily connected) directed graph. A node in this
109 graph is represented by a *Node Template*. A Node Template specifies the occurrence of a Node Type as
110 a component of a service. A *Node Type* defines the properties of such a component (via *Node Type*
111 *Properties*) and the operations (via *Interfaces*) available to manipulate the component. Node Types are
112 defined separately for reuse purposes and a Node Template references a Node Type and adds usage
113 constraints, such as how many times the component can occur.

114



115

116 Figure 1: Structural Elements of a Service Template and their Relations

117 For example, consider a service that consists of an application server, a process engine, and a process
118 model. A Topology Template defining that service would include one Node Template of Node Type
119 "application server", another Node Template of Node Type "process engine", and a third Node Template
120 of Node Type "process model". The application server Node Type defines properties like the IP address
121 of an instance of this type, an operation for installing the application server with the corresponding IP
122 address, and an operation for shutting down an instance of this application server. A constraint in the
123 Node Template can specify a range of IP addresses available when making a concrete application server
124 available.

125 A *Relationship Template* specifies the occurrence of a relationship between nodes in a Topology
126 Template. Each Relationship Template refers to a Relationship Type that defines the semantics and any
127 properties of the relationship. Relationship Types are defined separately for reuse purposes. The
128 Relationship Template indicates the elements it connects and the direction of the relationship by defining
129 one source and one target element (in nested `SourceElement` and `TargetElement` elements). The
130 Relationship Template also defines any constraints with the OPTIONAL
131 `RelationshipConstraints` element.

132 For example, a relationship can be established between the process engine Node Template and
133 application server Node Template with the meaning “hosted by”, and between the process model Node
134 Template and process engine Node Template with meaning “deployed on”.

135 A deployed service is an instance of a Service Template. More precisely, the instance is derived by
136 instantiating the Topology Template of its Service Template, most often by running a special plan defined
137 for the Service Template, often referred to as build plan. The build plan will provide actual values for the
138 various properties of the various Node Templates and Relationship Templates of the Topology Template.
139 These values can come from input passed in by users as triggered by human interactions defined within
140 the build plan, by automated operations defined within the build plan (such as a directory lookup), or the
141 templates can specify default values for some properties. The build plan will typically make use of
142 operations of the Node Types of the Node Templates.

143 For example, the application server Node Template will be instantiated by installing an actual application
144 server at a concrete IP address considering the specified range of IP addresses. Next, the process
145 engine Node Template will be instantiated by installing a concrete process engine on that application
146 server (as indicated by the “hosted by” relationship template). Finally, the process model Node Template
147 will be instantiated by deploying the process model on that process engine (as indicated by the “deployed
148 on” relationship template).

149 *Plans* defined in a Service Template describe the management aspects of service instances, especially
150 their creation and termination. These plans are defined as process models, i.e. a workflow of one or more
151 steps. Instead of providing another language for defining process models, the specification relies on
152 existing languages like BPMN or BPEL. Relying on existing standards in this space facilitates portability
153 and interoperability, but any language for defining process models can be used. The TOSCA metamodel
154 provides containers to either refer to a process model (via *Plan Model Reference*) or to include the actual
155 model in the plan (via *Plan Model*). A process model can contain tasks (using BPMN terminology) that
156 refer to operations of Interfaces of Node Templates (or operations defined by the Node Types specified in
157 the `type` attribute of the Node Templates, respectively), operations of Interfaces of Relationship
158 Templates (or operations defined by the Relationship Types specified in the `type` attribute of the
159 Relationship Templates, respectively), or any other interface (e.g. the invocation of an external service for
160 licensing); in doing so, a plan can directly manipulate nodes of the topology of a service or interact with
161 external systems.

162 3.2 Use Cases

163 The specification supports at least the following major use cases.

164 3.2.1 Services as Marketable Entities

165 Standardizing Service Templates will support the creation of a market for hosted IT services. Especially, a
166 standard for specifying Topology Templates (i.e. the set of components a service consists of as well as
167 their mutual dependencies) enables interoperable definitions of the structure of services. Such a service
168 topology model could be created by a service developer who understands the internals of a particular
169 service. The Service Template could then be published in catalogs of one or more service providers for
170 selection and use by potential customers. Each service provider would map the specified service topology
171 to its available concrete infrastructure in order to support concrete instances of the service and adapt the
172 management plans accordingly.

173 Making a concrete instance of a Topology Template can be done by running a corresponding Plan (so-
174 called instantiating management plan, a.k.a. build plan). This build plan could be provided by the service
175 developer who also creates the Service Template. The build plan can be adapted to the concrete

176 environment of a particular service provider. Other management plans useful in various states of the
177 whole lifecycle of a service could be specified as part of a Service Template. Similar to build plans such
178 management plans can be adapted to the concrete environment of a particular service provider.
179 Thus, not only the structure of a service can be defined in an interoperable manner, but also its
180 management plans. These Plans describe how instances of the specified service are created and
181 managed. Defining a set of management plans for a service will significantly reduce the cost of hosting a
182 service by providing reusable knowledge about best practices for managing each service. While the
183 modeler of a service can include deep domain knowledge into a plan, the user of such a service can use
184 a plan by simply “invoking” it. This hides the complexity of the underlying service behavior. This is very
185 similar to the situation resulting in the specification of ITIL.

186 3.2.2 Portability of Service Templates

187 Standardizing Service Templates supports the portability of definitions of IT Services. Here, portability
188 denotes the ability of one cloud provider to understand the structure and behavior of a Service Template
189 created by another party, e.g. another cloud provider, enterprise IT department, or service developer.
190 Note that portability of a service does not imply portability of its encompassed components. Portability of
191 a service means that its definition can be understood in an interoperable manner, i.e. the topology model
192 and corresponding plans are understood by standard compliant vendors. Portability of the individual
193 components themselves making up a particular service has to be ensured by other means – if it is
194 important for the service.

195 3.2.3 Service Composition

196 Standardizing Service Templates facilitates composing a service from components even if those
197 components are hosted by different providers, including the local IT department, or in different automation
198 environments, often built with technology from different suppliers. For example, large organizations could
199 use automation products from different suppliers for different data centers, e.g., because of geographic
200 distribution of data centers or organizational independence of each location. A Service Template provides
201 an abstraction that does not make assumptions about the hosting environments.

202 3.2.4 Relation to Virtual Images

203 A cloud provider can host a service based on virtualized middleware stacks. These middleware stacks
204 might be represented by an image definition such as an OVF [OVF] package. If OVF is used, a node in a
205 Service Template can correspond to a virtual system or a component (OVF's "product") running in a
206 virtual system, as defined in an OVF package. If the OVF package defines a virtual system collection
207 containing multiple virtual systems, a sub-tree of a Service Template could correspond to the OVF virtual
208 system collection.

209 A Service Template provides a way to declare the association of Service Template elements to OVF
210 package elements. Such an association expresses that the corresponding Service Template element can
211 be instantiated by deploying the corresponding OVF package element. These associations are not limited
212 to OVF packages. The associations could be to other package types or to external service interfaces.
213 This flexibility allows a Service Template to be composed from various virtualization technologies, service
214 interfaces, and proprietary technology.

215 3.3 Service Templates and Artifacts

216 An artifact represents the content needed to realize a deployment such as an executable (e.g. a script, an
217 executable program, an image), a configuration file or data file, or something that might be needed so that
218 another executable can run (e.g. a library). Artifacts can be of different types, for example EJBs or python
219 scripts. The content of an artifact depends on its type. Typically, descriptive metadata will also be
220 provided along with the artifact. This metadata might be needed to properly process the artifact, for
221 example by describing the appropriate execution environment.

222 TOSCA distinguishes two kinds of artifacts: *implementation artifacts* and *deployment artifacts*. An
223 implementation artifact represents the executable of an operation of a node type, and a deployment

224 artifact represents the executable for materializing instances of a node. For example, a REST operation
225 to store an image can have an implementation artifact that is a WAR file. The node type this REST
226 operation is associated with can have the image itself as a deployment artifact.

227 The fundamental difference between implementation artifacts and deployment artifacts is twofold, namely

- 228 1. the point in time when the artifact is deployed, and
- 229 2. by what entity and to where the artifact is deployed.

230 The operations of a node type perform management actions on (instances of) the node type. The
231 implementations of such operations can be provided as implementation artifacts. Thus, the
232 implementation artifacts of the corresponding operations have to be deployed in the management
233 environment before any management operation can be started. In other words, “a TOSCA supporting
234 environment” (i.e. a so-called TOSCA container) MUST be able to process the set of implementation
235 artifacts types needed to execute those management operations. One such management operation could
236 be the instantiation of a node type.

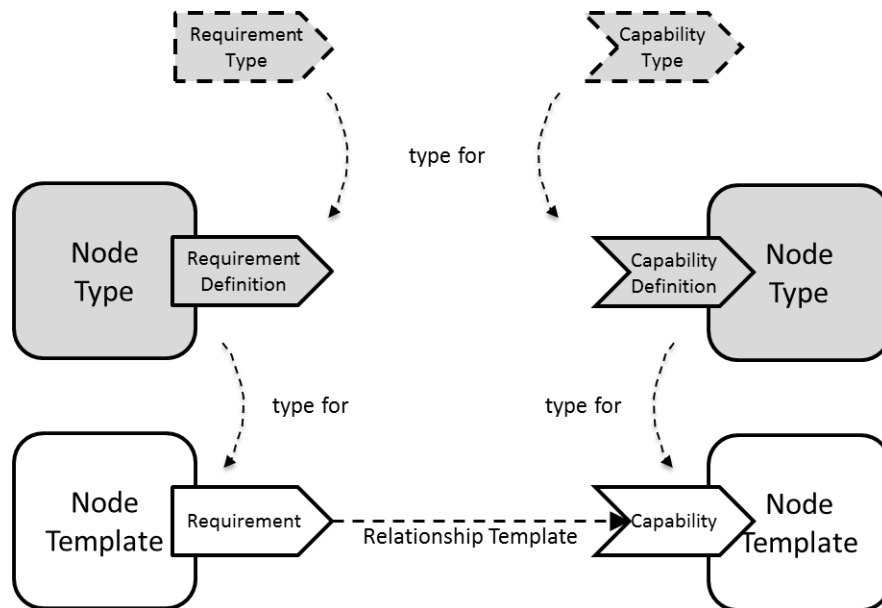
237 The instantiation of a node type can require providing deployment artifacts in the target managed
238 environment. For this purpose, a TOSCA container supports a set of types of deployment artifacts that it
239 can process. A service template that contains (implementation or deployment) artifacts of non-supported
240 types cannot be processed by the container (resulting in an error during import).

241 3.4 Requirements and Capabilities

242 TOSCA allows for expressing *requirements* and *capabilities* of components of a service. This can be
243 done, for example, to express that one component depends on (requires) a feature provided by another
244 component, or to express that a component has certain requirements against the hosting environment
245 such as for the allocation of certain resources or the enablement of a specific mode of operation.

246 Requirements and capabilities are modeled by annotating Node Types with *Requirement Definitions* and
247 *Capability Definitions* of certain types. *Requirement Types* and *Capability Types* are defined as reusable
248 entities so that those definitions can be used in the context of several Node Types. For example, a
249 Requirement Type “DatabaseConnectionRequirement” might be defined to describe the requirement of a
250 client for a database connection. This Requirement Type can then be reused for all kinds of Node Types
251 that represent, for example, application with the need for a database connection.

252



253

254

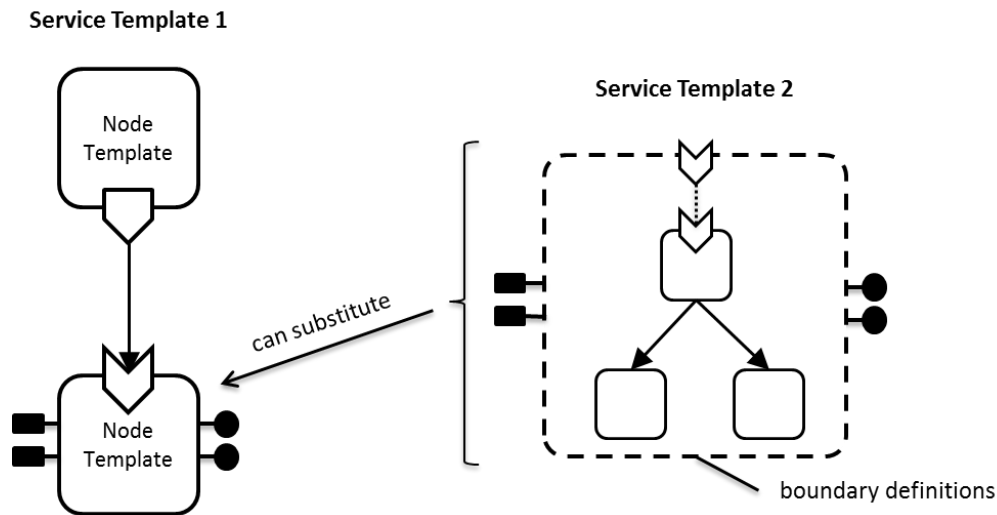
255

Figure 2: Requirements and Capabilities

256 Node Templates which have corresponding Node Types with Requirement Definitions or Capability
 257 Definitions will include representations of the respective *Requirements* and *Capabilities* with content
 258 specific to the respective Node Template. For example, while Requirement Types just represent
 259 Requirement metadata, the Requirement represented in a Node Template can provide concrete values
 260 for properties defined in the Requirement Type. In addition, Requirements and Capabilities of Node
 261 Templates in a Topology Template can optionally be connected via Relationship Templates to indicate
 262 that a specific requirement of one node is fulfilled by a specific capability provided by another node.
 263 Requirements can be matched in two ways as briefly indicated above: (1) requirements of a Node
 264 Template can be matched by capabilities of another Node Template in the same Service Template by
 265 connecting the respective requirement-capability-pairs via Relationship Templates; (2) requirements of a
 266 Node Template can be matched by the general hosting environment (or the TOSCA container), for
 267 example by allocating needed resources for a Node Template during instantiation.

268 3.5 Composition of Service Templates

269 Service Templates can be based on and built on-top of other Service Templates based on the concept of
 270 Requirements and Capabilities introduced in the previous section. For example, a Service Template for a
 271 business application that is hosted on an application server tier might focus on defining the structure and
 272 manageability behavior of the application itself. The structure of the application server tier hosting the
 273 application can be provided in a separate Service Template built by another vendor specialized in
 274 deploying and managing application servers. This approach enables separation of concerns and re-use of
 275 common infrastructure templates.



276
 277 Figure 3: Service Template Composition

278 From the point of view of a Service Template (e.g. the business application Service Template from the
 279 example above) that uses another Service Template, the other Service Template (e.g. the application
 280 server tier) “looks” like just a Node Template. During deployment, however, this Node Template can be
 281 substituted by the second Service Template if it exposes the same boundaries (i.e. properties,
 282 capabilities, etc.) as the Node Template. Thus, a substitution with any Service Template that has the
 283 same *boundary definitions* as a certain Node Template in one Service Template becomes possible,
 284 allowing for a flexible composition of different Service Templates. This concept also allows for providing
 285 substitutable alternatives in the form of Service Templates. For example, a Service Template for a single
 286 node application server tier and a Service Template for a clustered application server tier might exist,
 287 and the appropriate option can be selected per deployment.

288 3.6 Policies in TOSCA

289 Non-functional behavior or quality-of-services are defined in TOSCA by means of policies. A Policy can
 290 express such diverse things like monitoring behavior, payment conditions, scalability, or continuous
 291 availability, for example.

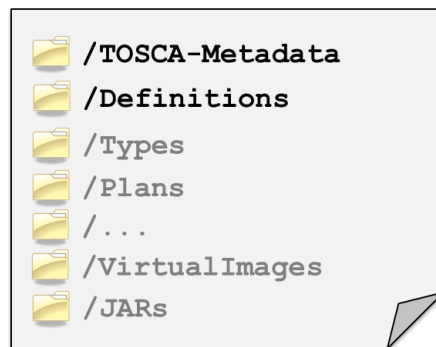
292 A Node Template can be associated with a set of Policies collectively expressing the non-functional
293 behavior or quality-of-services that each instance of the Node Template will expose. Each Policy specifies
294 the actual properties of the non-functional behavior, like the concrete payment information (payment
295 period, currency, amount etc) about the individual instances of the Node Template.

296 These properties are defined by a Policy Type. Policy Types might be defined in hierarchies to properly
297 reflect the structure of non-functional behavior or quality-of-services in particular domains. Furthermore, a
298 Policy Type might be associated with a set of Node Types the non-functional behavior or quality-of-
299 service it describes.

300 Policy Templates provide actual values of properties of the types defined by Policy Types. For example, a
301 Policy Template for monthly payments for US customers will set the “payment period” property to
302 “monthly” and the “currency” property to “US\$”, leaving the “amount” property open. The “amount”
303 property will be set when the corresponding Policy Template is used for a Policy within a Node Template.
304 Thus, a Policy Template defines the invariant properties of a Policy, while the Policy sets the variant
305 properties resulting from the actual usage of a Policy Template in a Node Template.

306 3.7 Archive Format for Cloud Applications

307 In order to support in a certain environment the execution and management of the lifecycle of a cloud
308 application, all corresponding artifacts have to be available in that environment. This means that beside
309 the service template of the cloud application, the deployment artifacts and implementation artifacts have
310 to be available in that environment. To ease the task of ensuring the availability of all of these, this
311 specification defines a corresponding archive format called CSAR (Cloud Service ARchive).



312

313

Figure 4: Structure of the CSAR

314 A CSAR is a container file, i.e. it contains multiple files of possibly different file types. These files are
315 typically organized in several subdirectories, each of which contains related files (and possibly other
316 subdirectories etc). The organization into subdirectories and their content is specific for a particular cloud
317 application. CSARs are zip files, typically compressed.

318 Each CSAR MUST contain a subdirectory called *TOSCA-Metadata*. This subdirectory MUST contain a
319 so-called *TOSCA meta file*. This file is named *TOSCA* and has the file extension *.meta*. It represents
320 metadata of the other files in the CSAR. This metadata is given in the format of name/value pairs. These
321 name/value pairs are organized in blocks. Each block provides metadata of a certain artifact of the CSAR.
322 An empty line separates the blocks in the *TOSCA meta file*.

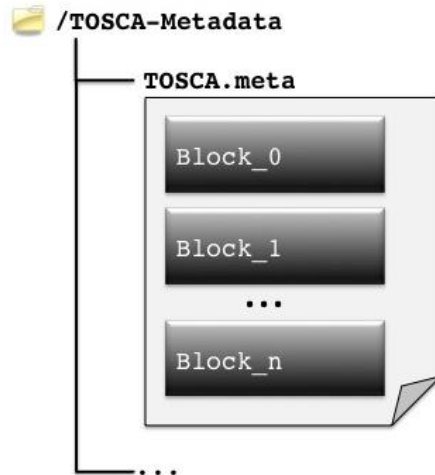
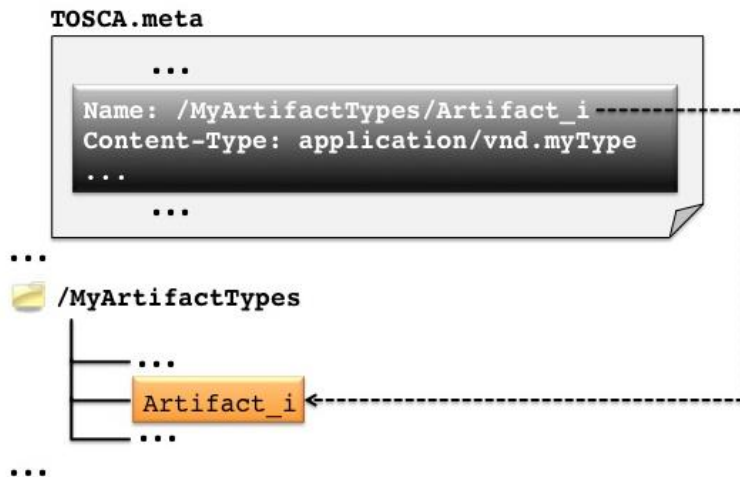


Figure 5: Structure of the TOSCA Meta File

323
324

325 The first block of the TOSCA meta file (Block_0 in Figure 5) provides metadata of the CSAR itself (e.g. its
326 version, creator etc). Each other block begins with a name/value pair that points to an artifact within the
327 CSAR by means of a pathname. The remaining name/value pairs in a block are the proper metadata of
328 the pointed to artifact. For example, a corresponding name/value pair specifies the MIME-type of the
329 artifact.



330
331
332

Figure 6: Providing Metadata for Artifacts

333 4 The TOSCA Definitions Document

334 All elements needed to define a TOSCA Service Template – such as Node Type definitions, Relationship
335 Type definitions, etc. – as well as Service Templates themselves are provided in TOSCA *Definitions*
336 documents. This section explains the overall structure of a TOSCA Definitions document, the extension
337 mechanism, and import features. Later sections describe in detail Service Templates, Node Types, Node
338 Type Implementations, Relationship Types, Relationship Type Implementations, Requirement Types,
339 Capability Types, Artifact Types, Artifact Templates, Policy Types and Policy Templates.

340 4.1 XML Syntax

341 The following pseudo schema defines the XML syntax of a Definitions document:

```
342 01 <Definitions id="xs:ID"  
343 02     name="xs:string"?  
344 03     targetNamespace="xs:anyURI">  
345 04  
346 05     <Extensions>  
347 06         <Extension namespace="xs:anyURI"  
348 07             mustUnderstand="yes|no"?/> +  
349 08     </Extensions> ?  
350 09  
351 10     <Import namespace="xs:anyURI"?  
352 11         location="xs:anyURI"?  
353 12         importType="xs:anyURI"/> *  
354 13  
355 14     <Types>  
356 15         <xs:schema .../> *  
357 16     </Types> ?  
358 17  
359 18     (  
360 19         <ServiceTemplate> ... </ServiceTemplate>  
361 20     |  
362 21         <NodeType> ... </NodeType>  
363 22     |  
364 23         <NodeTypeImplementation> ... </NodeTypeImplementation>  
365 24     |  
366 25         <RelationshipType> ... </RelationshipType>  
367 26     |  
368 27         <RelationshipTypeImplementation> ... </RelationshipTypeImplementation>  
369 28     |  
370 29         <RequirementType> ... </RequirementType>  
371 30     |  
372 31         <CapabilityType> ... </CapabilityType>  
373 32     |  
374 33         <ArtifactType> ... </ArtifactType>  
375 34     |  
376 35         <ArtifactTemplate> ... </ArtifactTemplate>  
377 36     |  
378 37         <PolicyType> ... </PolicyType>  
379 38     |  
380 39         <PolicyTemplate> ... </PolicyTemplate>  
381 40     ) +  
382 41  
383 42 </Definitions>
```

384 4.2 Properties

385 The `Definitions` element has the following properties:

- 386 • `id`: This attribute specifies the identifier of the Definitions document which MUST be unique
387 within the target namespace.
- 388 • `name`: This OPTIONAL attribute specifies a descriptive name of the Definitions document.
- 389 • `targetNamespace`: The value of this attribute specifies the target namespace for the
390 Definitions document. All elements defined within the Definitions document will be added to this
391 namespace unless they override this attribute by means of their own `targetNamespace`
392 attributes.
- 393 • `Extensions`: This OPTIONAL element specifies namespaces of TOSCA extension attributes
394 and extension elements. If present, the `Extensions` element MUST include at least one
395 `Extension` element.

396 The `Extension` element has the following properties:

- 397 ○ `namespace`: This attribute specifies the namespace of TOSCA extension attributes and
398 extension elements.
- 399 ○ `mustUnderstand`: This OPTIONAL attribute specifies whether the extension MUST
400 be understood by a compliant implementation. If the `mustUnderstand` attribute has
401 value “yes” (which is the default value for this attribute) the extension is mandatory.
402 Otherwise, the extension is optional.
403 If a TOSCA implementation does not support one or more of the mandatory extensions,
404 then the Definitions document MUST be rejected. Optional extensions MAY be ignored. It
405 is not necessary to declare optional extensions.
406 The same extension URI MAY be declared multiple times in the `Extensions` element.
407 If an extension URI is identified as mandatory in one `Extension` element and optional
408 in another, then the mandatory semantics have precedence and MUST be enforced. The
409 extension declarations in an `Extensions` element MUST be treated as an unordered
410 set.
- 411 • `Import`: This element declares a dependency on external TOSCA Definitions, XML Schema
412 definitions, or WSDL definitions. Any number of `Import` elements MAY appear as children of
413 the `Definitions` element.

414 The `Import` element has the following properties:

- 415 ○ `namespace`: This OPTIONAL attribute specifies an absolute URI that identifies the
416 imported definitions. An `Import` element without a `namespace` attribute indicates that
417 external definitions are in use, which are not namespace-qualified. If a `namespace`
418 attribute is specified then the imported definitions MUST be in that namespace. If no
419 namespace is specified then the imported definitions MUST NOT contain a
420 `targetNamespace` specification. The namespace
421 `http://www.w3.org/2001/XMLSchema` is imported implicitly. Note, however, that there is
422 no implicit XML Namespace prefix defined for `http://www.w3.org/2001/XMLSchema`.
- 423 ○ `location`: This OPTIONAL attribute contains a URI indicating the location of a
424 document that contains relevant definitions. The location URI MAY be a relative URI,
425 following the usual rules for resolution of the URI base [XML Base, RFC 2396]. An
426 `Import` element without a `location` attribute indicates that external definitions are
427 used but makes no statement about where those definitions might be found. The
428 `location` attribute is a hint and a TOSCA compliant implementation is not obliged to
429 retrieve the document being imported from the specified location.

430 o `importType`: This REQUIRED attribute identifies the type of document being imported
431 by providing an absolute URI that identifies the encoding language used in the document.
432 The value of the `importType` attribute MUST be set to `http://docs.oasis-`
433 `open.org/tosca/ns/2011/12` when importing Service Template documents, to
434 `http://schemas.xmlsoap.org/wsdl/` when importing WSDL 1.1 documents, and to
435 `http://www.w3.org/2001/XMLSchema` when importing an XSD document.

436 According to these rules, it is permissible to have an `Import` element without `namespace` and
437 `location` attributes, and only containing an `importType` attribute. Such an `Import`
438 element indicates that external definitions of the indicated type are in use that are not
439 namespace-qualified, and makes no statement about where those definitions might be found.

440 A Definitions document MUST define or import all Node Types, Node Type Implementations,
441 Relationship Types, Relationship Type Implementations, Requirement Type, Capability Types,
442 Artifact Types, Policy Types, WSDL definitions, and XML Schema documents it uses. In order to
443 support the use of definitions from namespaces spanning multiple documents, a Definitions
444 document MAY include more than one import declaration for the same `namespace` and
445 `importType`. Where a Definitions document has more than one import declaration for a given
446 `namespace` and `importType`, each declaration MUST include a different `location` value.
447 `Import` elements are conceptually unordered. A Definitions document MUST be rejected if the
448 imported documents contain conflicting definitions of a component used by the importing
449 Definitions document.

450 Documents (or namespaces) imported by an imported document (or namespace) are not
451 transitively imported by a TOSCA compliant implementation. In particular, this means that if an
452 external item is used by an element enclosed in the Definitions document, then a document (or
453 namespace) that defines that item MUST be directly imported by the Definitions document. This
454 requirement does not limit the ability of the imported document itself to import other documents or
455 namespaces.

456 • `Types`: This element specifies XML definitions introduced within the Definitions document. Such
457 definitions are provided within one or more separate Schema definitions (usually `xs:schema`
458 elements). The `Types` element defines XML definitions within a Definitions document without
459 having to define these XML definitions in separate files and importing them. Note, that an
460 `xs:schema` element nested in the `Types` element MUST be a valid XML schema definition. In
461 case the `targetNamespace` attribute of a nested `xs:schema` element is not specified, all
462 definitions within this element become part of the target namespace of the encompassing
463 Definitions element.

464 Note: The specification supports the use of any type system nested in the `Types` element.
465 Nevertheless, only the support of `xs:schema` is REQUIRED from any compliant
466 implementation.

467 • `ServiceTemplate`: This element specifies a complete Service Template for a cloud
468 application. A Service Template contains a definition of the Topology Template of the cloud
469 application, as well as any number of Plans. Within the Service Template, any type definitions
470 (e.g. Node Types, Relationship Types, etc.) defined in the same Definitions document or in
471 imported Definitions document can be used.

472 • `NodeType`: This element specifies a type of Node that can be referenced as a type for Node
473 Templates of a Service Template.

474 • `NodeTypeImplementation`: This element specifies the implementation of the manageability
475 behavior of a type of Node that can be referenced as a type for Node Templates of a Service
476 Template.

477 • `RelationshipType`: This element specifies a type of Relationship that can be referenced as
478 a type for Relationship Templates of a Service Template.

- 479 • `RelationshipTypeImplementation`: This element specifies the implementation of the
480 manageability behavior of a type of Relationship that can be referenced as a type for Relationship
481 Templates of a Service Template.
- 482 • `RequirementType`: This element specifies a type of Requirement that can be exposed by
483 Node Types used in a Service Template.
- 484 • `CapabilityType`: This element specifies a type of Capability that can be exposed by Node
485 Types used in a Service Template.
- 486 • `ArtifactType`: This element specifies a type of artifact used within a Service Template.
487 Artifact Types might be, for example, application modules such as .war files or .ear files,
488 operating system packages like RPMs, or virtual machine images like .ova files.
- 489 • `ArtifactTemplate`: This element specifies a template describing an artifact referenced by
490 parts of a Service Template. For example, the installable artifact for an application server node
491 might be defined as an artifact template.
- 492 • `PolicyType`: This element specifies a type of Policy that can be associated to Node Templates
493 defined within a Service Template. For example, a scaling policy for nodes in a web server tier
494 might be defined as a Policy Type, which specifies the attributes the scaling policy can have.
- 495 • `PolicyTemplate`: This element specifies a template of a Policy that can be associated to
496 Node Templates defined within a Service Template. Other than a Policy Type, a Policy Template
497 can define concrete values for a policy according to the set of attributes specified by the Policy
498 Type the Policy Template refers to.

499 A TOSCA Definitions document MUST define at least one of the elements `ServiceTemplate`,
500 `NodeType`, `NodeTypeImplementation`, `RelationshipType`,
501 `RelationshipTypeImplementation`, `RequirementType`, `CapabilityType`,
502 `ArtifactType`, `ArtifactTemplate`, `PolicyType`, or `PolicyTemplate`, but it can define any
503 number of those elements in an arbitrary order.

504 This technique supports a modular definition of Service Templates. For example, one Definitions
505 document can contain only Node Type and Relationship Type definitions that can then be imported into
506 another Definitions document that only defines a Service Template using those Node Types and
507 Relationship Types. Similarly, Node Type Properties can be defined in separate XML Schema Definitions
508 that are imported and referenced when defining a Node Type.

509 All TOSCA elements MAY use the `documentation` element to provide annotation for users. The
510 content could be a plain text, HTML, and so on. The `documentation` element is OPTIONAL and has
511 the following syntax:

```
512 01 <documentation source="xs:anyURI"? xml:lang="xs:language"?>
513 02   ...
514 03 </documentation>
```

515 Example of use of a `documentation` element:

```
516 01 <Definitions id="MyDefinitions" name="My Definitions" ...>
517 02
518 03   <documentation xml:lang="EN">
519 04     This is a simple example of the usage of the documentation
520 05     element nested under a Definitions element. It could be used,
521 06     for example, to describe the purpose of the Definitions document
522 07     or to give an overview of elements contained within the Definitions
523 08     document.
524 09   </documentation>
525 10
526 11 </Definitions>
```

527 4.3 Example

528 The following Definitions document defines two Node Types, “Application” and “ApplicationServer”, as
529 well as one Relationship Type “ApplicationHostedOnApplicationServer”. The properties definitions for the
530 two Node Types are specified in a separate XML schema definition file which is imported into the
531 Definitions document by means of the `Import` element.

```
532 01 <Definitions id="MyDefinitions" name="My Definitions"  
533 02   targetNamespace="http://www.example.com/MyDefinitions"  
534 03   xmlns:my="http://www.example.com/MyDefinitions">  
535 04  
536 05   <Import importType="http://www.w3.org/2001/XMLSchema"  
537 06     namespace="http://www.example.com/MyDefinitions">  
538 07  
539 08   <NodeType name="Application">  
540 09     <PropertiesDefinition element="my:ApplicationProperties"/>  
541 10   </NodeType>  
542 11  
543 12   <NodeType name="ApplicationServer">  
544 13     <PropertiesDefinition element="my:ApplicationServerProperties"/>  
545 14   </NodeType>  
546 15  
547 16   <RelationshipType name="ApplicationHostedOnApplicationServer">  
548 17     <ValidSource typeRef="my:Application"/>  
549 18     <ValidTarget typeRef="my:ApplicationServer"/>  
550 19   </RelationshipTemplate>  
551 20  
552 21 </Definitions>
```

553

5 Service Templates

554 This chapter specifies how *Service Templates* are defined. A Service Template describes the structure of
555 a cloud application by means of a Topology Template, and it defines the manageability behavior of the
556 cloud application in the form of Plans.

557 Elements within a Service Template, such as Node Templates defined in the Topology Template, refer to
558 other TOSCA element, such as Node Types that can be defined in the same Definitions document
559 containing the Service Template, or that can be defined in separate, imported Definitions documents.

560 Service Templates can be defined for being directly used for the deployment and management of a cloud
561 application, or they can be used for composition into larger Service Template (see section 3.5 for details).

562 5.1 XML Syntax

563 The following pseudo schema defines the XML syntax of a Service Template:

```
564 01 <ServiceTemplate id="xs:ID"  
565 02     name="xs:string"?  
566 03     targetNamespace="xs:anyURI"  
567 04     substitutableNodeType="xs:QName"?>  
568 05  
569 06 <Tags>  
570 07   <Tag name="xs:string" value="xs:string"/> +  
571 08 </Tags> ?  
572 09  
573 10 <BoundaryDefinitions>  
574 11   <Properties>  
575 12     XML fragment  
576 13     <PropertyMappings>  
577 14       <PropertyMapping serviceTemplatePropertyRef="xs:string"  
578 15         targetObjectRef="xs:IDREF"  
579 16         targetPropertyRef="xs:string"/> +  
580 17     </PropertyMappings/> ?  
581 18   </Properties> ?  
582 19  
583 20   <PropertyConstraints>  
584 21     <PropertyConstraint property="xs:string"  
585 22       constraintType="xs:anyURI"> +  
586 23     constraint ?  
587 24   </PropertyConstraint>  
588 25 </PropertyConstraints> ?  
589 26  
590 27 <Requirements>  
591 28   <Requirement name="xs:string"? ref="xs:IDREF"/> +  
592 29 </Requirements> ?  
593 30  
594 31 <Capabilities>  
595 32   <Capability name="xs:string"? ref="xs:IDREF"/> +  
596 33 </Capabilities> ?  
597 34  
598 35 <Policies>  
599 36   <Policy name="xs:string"? policyType="xs:QName"  
600 37     policyRef="xs:QName"?>  
601 38     policy specific content ?  
602 39   </Policy> +  
603 40 </Policies> ?
```

```

604 41
605 42     <Interfaces>
606 43         <Interface name="xs:NCName">
607 44             <Operation name="xs:NCName">
608 45                 (
609 46                     <NodeOperation nodeRef="xs:IDREF"
610 47                         interfaceName="xs:anyURI"
611 48                         operationName="xs:NCName"/>
612 49                 |
613 50                     <RelationshipOperation relationshipRef="xs:IDREF"
614 51                         interfaceName="xs:anyURI"
615 52                         operationName="xs:NCName"/>
616 53                 |
617 54                     <Plan planRef="xs:IDREF"/>
618 55                 )
619 56             </Operation> +
620 57         </Interface> +
621 58     </Interfaces> ?
622 59
623 60 </BoundaryDefinitions> ?
624 61
625 62 <TopologyTemplate>
626 63     (
627 64         <NodeTemplate id="xs:ID" name="xs:string"? type="xs:QName"
628 65             minInstances="xs:integer"?
629 66             maxInstances="xs:integer | xs:string"?>
630 67             <Properties>
631 68                 XML fragment
632 69             </Properties> ?
633 70
634 71             <PropertyConstraints>
635 72                 <PropertyConstraint property="xs:string"
636 73                     constraintType="xs:anyURI">
637 74                     constraint ?
638 75                 </PropertyConstraint> +
639 76             </PropertyConstraints> ?
640 77
641 78             <Requirements>
642 79                 <Requirement id="xs:ID" name="xs:string" type="xs:QName"> +
643 80                     <Properties>
644 81                         XML fragment
645 82                     <Properties> ?
646 83                     <PropertyConstraints>
647 84                         <PropertyConstraint property="xs:string"
648 85                             constraintType="xs:anyURI"> +
649 86                             constraint ?
650 87                         </PropertyConstraint>
651 88                     </PropertyConstraints> ?
652 89                 </Requirement>
653 90             </Requirements> ?
654 91
655 92             <Capabilities>
656 93                 <Capability id="xs:ID" name="xs:string" type="xs:QName"> +
657 94                     <Properties>
658 95                         XML fragment
659 96                     <Properties> ?
660 97                     <PropertyConstraints>
661 98                         <PropertyConstraint property="xs:string"

```



```

662 99          constraintType="xs:anyURI">
663 100          constraint ?
664 101          </PropertyConstraint> +
665 102          </PropertyConstraints> ?
666 103          </Capability>
667 104          </Capabilities> ?
668 105
669 106          <Policies>
670 107          <Policy name="xs:string"? policyType="xs:QName"
671 108          policyRef="xs:QName"?>
672 109          policy specific content ?
673 110          </Policy> +
674 111          </Policies> ?
675 112
676 113          <DeploymentArtifacts>
677 114          <DeploymentArtifact name="xs:string" artifactType="xs:QName"
678 115          artifactRef="xs:QName"?>
679 116          artifact specific content ?
680 117          </DeploymentArtifact> +
681 118          </DeploymentArtifacts> ?
682 119          </NodeTemplate>
683 120          |
684 121          <RelationshipTemplate id="xs:ID" name="xs:string"?
685 122          type="xs:QName">
686 123          <Properties>
687 124          XML fragment
688 125          </Properties> ?
689 126
690 127          <PropertyConstraints>
691 128          <PropertyConstraint property="xs:string"
692 129          constraintType="xs:anyURI">
693 130          constraint ?
694 131          </PropertyConstraint> +
695 132          </PropertyConstraints> ?
696 133
697 134          <SourceElement ref="xs:IDREF"/>
698 135          <TargetElement ref="xs:IDREF"/>
699 136
700 137          <RelationshipConstraints>
701 138          <RelationshipConstraint constraintType="xs:anyURI">
702 139          constraint ?
703 140          </RelationshipConstraint> +
704 141          </RelationshipConstraints> ?
705 142
706 143          </RelationshipTemplate>
707 144          ) +
708 145          </TopologyTemplate>
709 146
710 147          <Plans>
711 148          <Plan id="xs:ID"
712 149          name="xs:string"?
713 150          planType="xs:anyURI"
714 151          planLanguage="xs:anyURI">
715 152
716 153          <Precondition expressionLanguage="xs:anyURI">
717 154          condition
718 155          </Precondition> ?
719 156

```

```

720 157     <InputParameters>
721 158         <InputParameter name="xs:string" type="xs:string"
722 159             required="yes|no"?/> +
723 160     </InputParameters> ?
724 161
725 162     <OutputParameters>
726 163         <OutputParameter name="xs:string" type="xs:string"
727 164             required="yes|no"?/> +
728 165     </OutputParameters> ?
729 166
730 167     (
731 168         <PlanModel>
732 169             actual plan
733 170         </PlanModel>
734 171         |
735 172         <PlanModelReference reference="xs:anyURI"/>
736 173     )
737 174
738 175     </Plan> +
739 176 </Plans> ?
740 177
741 178 </ServiceTemplate>

```

742 5.2 Properties

743 The `ServiceTemplate` element has the following properties:

- 744 • `id`: This attribute specifies the identifier of the Service Template which **MUST** be unique within
745 the target namespace.
- 746 • `name`: This **OPTIONAL** attribute specifies a descriptive name of the Service Template.
- 747 • `targetNamespace`: The value of this **OPTIONAL** attribute specifies the target namespace for
748 the Service Template. If not specified, the Service Template will be added to the namespace
749 declared by the `targetNamespace` attribute of the enclosing `Definitions` element.
- 750 • `substitutableNodeType`: This **OPTIONAL** attribute specifies a Node Type that can be
751 substituted by this Service Template. If another Service Template contains a Node Template of
752 the specified Node Type (or any Node Type this Node Type is derived from), this Node Template
753 can be substituted by an instance of this Service Template that then provides the functionality of
754 the substituted node. See section 3.5 for more details.
- 755 • `Tags`: This **OPTIONAL** element allows the definition of any number of tags which can be used by
756 the author to describe the Service Template. Each tag is defined by a separate, nested `Tag`
757 element.
758 The `Tag` element has the following properties:
 - 759 ○ `name`: This attribute specifies the name of the tag.
 - 760 ○ `value`: This attribute specifies the value of the tag.
- 761
762 **Note:** The name/value pairs defined in tags have no normative interpretation.
- 763 • `BoundaryDefinitions`: This **OPTIONAL** element specifies the properties the Service
764 Template exposes beyond its boundaries, i.e. properties that can be observed from outside the
765 Service Template. The `BoundaryDefinitions` element has the following properties.
 - 766 ○ `Properties`: This **OPTIONAL** element specifies global properties of the Service
767 Template in the form of an XML fragment contained in the body of the `Properties`
768 element. Those properties **MAY** be mapped to properties of components within the

769 Service Template to make them visible to the outside.
770 The `Properties` element has the following properties:

- 771 ▪ `PropertyMappings`: This OPTIONAL element specifies mappings of one or
772 more of the Service Template's properties to properties of components within the
773 Service Template (e.g. Node Templates, Relationship Templates, etc.). Each
774 property mapping is defined by a separate, nested `PropertyMapping`
775 element. The `PropertyMapping` element has the following properties:
 - 776 • `serviceTemplatePropertyRef`: This attribute identifies a property
777 of the Service Template by means of an XPath expression to be
778 evaluated on the XML fragment defining the Service Template's
779 properties.
 - 780 • `targetObjectRef`: This attribute specifies the object that provides
781 the property to which the respective Service Template property is
782 mapped. The referenced target object MUST be one of Node Template,
783 Requirement of a Node Template, Capability of a Node Template, or
784 Relationship Template.
 - 785 • `targetPropertyRef`: This attribute identifies a property of the target
786 object by means of an XPath expression to be evaluated on the XML
787 fragment defining the target object's properties.

788 Note: If a Service Template property is mapped to a property of a
789 component within the Service Template, the XML schema type of the
790 Service Template property and the mapped property MUST be
791 compatible.
792
793 Note: If a Service Template property is mapped to a property of a
794 component within the Service Template, reading the Service Template
795 property corresponds to reading the mapped property, and writing the
796 Service Template property corresponds to writing the mapped property.
797
- 798 ○ `PropertyConstraints`: This OPTIONAL element specifies constraints on one or
799 more of the Service Template's properties. Each constraint is specified by means of a
800 separate, nested `PropertyConstraint` element.
801 The `PropertyConstraint` element has the following properties:
 - 802 ▪ `property`: This attribute identifies a property by means of an XPath expression
803 to be evaluated on the XML fragment defining the Service Template's properties.
804
805 Note: If the property affected by the property constraint is mapped to a property
806 of a component within the Service Template, the property constraint SHOULD be
807 compatible with any property constraint defined for the mapped property.
 - 808 ▪ `constraintType`: This attribute specifies the type of constraint by means of a
809 URI, which defines both the semantic meaning of the constraint as well as the
810 format of the content.
 - 811 ▪ The body of the `PropertyConstraint` element provides the actual
812 constraint.
813 Note: The body MAY be empty in case the `constraintType` URI already
814 specifies the constraint appropriately. For example, a "read-only" constraint could
815 be expressed solely by the `constraintType` URI.
- 816 ○ `Requirements`: This OPTIONAL element specifies Requirements exposed by the
817 Service Template. Those Requirements correspond to Requirements of Node Templates
818 within the Service Template that are propagated beyond the boundaries of the Service
819 Template. Each Requirement is defined by a separate, nested `Requirement` element.
820 The `Requirement` element has the following properties:

- 821 ▪ `name`: This OPTIONAL attribute allows for specifying a name of the Requirement
822 other than that specified by the referenced Requirement of a Node Template.
- 823 ▪ `ref`: This attribute references a Requirement element of a Node Template
824 within the Service Template.
- 825 ○ `Capabilities`: This OPTIONAL element specifies Capabilities exposed by the
826 Service Template. Those Capabilities correspond to Capabilities of Node Templates
827 within the Service Template that are propagated beyond the boundaries of the Service
828 Template. Each Capability is defined by a separate, nested `Capability` element. The
829 `Capability` element has the following properties:
- 830 ▪ `name`: This OPTIONAL attribute allows for specifying a name of the Capability
831 other than that specified by the referenced Capability of a Node Template.
- 832 ▪ `ref`: This attribute references a `Capability` element of a Node Template
833 within the Service Template.
- 834 ○ `Policies`: This OPTIONAL element specifies global policies of the Service Template
835 related to a particular management aspect. All Policies defined within the `Policies`
836 element MUST be enforced by a TOSCA implementation, i.e. Policies are AND-
837 combined. Each policy is defined by a separate, nested `Policy` element.
838 The `Policy` element has the following properties:
- 839 ▪ `name`: This OPTIONAL attribute allows for the definition of a name for the Policy.
840 If specified, this name MUST be unique within the containing `Policies`
841 element.
- 842 ▪ `policyType`: This attribute specifies the type of this Policy. The QName value
843 of this attribute SHOULD correspond to the QName of a `PolicyType` defined
844 in the same Definitions document or in an imported document.
- 845 The `policyType` attribute specifies the artifact type specific content of the
846 `Policy` element body and indicates the type of Policy Template referenced by
847 the Policy via the `policyRef` attribute.
- 848 ▪ `policyRef`: The QName value of this OPTIONAL attribute references a Policy
849 Template that is associated to the Service Template. This Policy Template can
850 be defined in the same TOSCA Definitions document, or it can be defined in a
851 separate document that is imported into the current Definitions document. The
852 type of Policy Template referenced by the `policyRef` attribute MUST be the
853 same type or a sub-type of the type specified in the `policyType` attribute.
854 The type of Policy Template referenced by the `policyRef` attribute MUST be the
855 same type or a sub-type of the type specified in the `policyType` attribute.
- 856 Note: if no Policy Template is referenced, the policy specific content of the
857 `Policy` element alone is assumed to represent sufficient policy specific
858 information in the context of the Service Template.
- 859 Note: while Policy Templates provide invariant information about a non-functional
860 behavior (i.e. information that is context independent, such as the availability
861 class of an availability policy), the `Policy` element defined in a Service
862 Template can provide variant information (i.e. information that is context specific,
863 such as a specific heartbeat frequency for checking availability of a service) in
864 the policy specific body of the `Policy` element.
- 865 The `Policy` element has the following properties:
- 866 ○ `Interfaces`: This OPTIONAL element specifies the interfaces with operations that can
867 be invoked on complete service instances created from the Service Template.
868 The `Interfaces` element has the following properties:
- 869 ▪ `Interface`: This element specifies one interfaces exposed by the Service
870 Template.
871 The `Interface` element has the following properties:

872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923

- `name`: This attribute specifies the name of the interfaces as either a URI or an NCName that **MUST** be unique in the scope of the Service Template's boundary definitions.
- `Operation`: This element specifies one exposed operation of an interface exposed by the Service Template.

An operation exposed by a Service Template maps to an internal component of the Service Template which actually provides the operation: it can be mapped to an operation provided by a Node Template (i.e. an operation defined by the Node Type specified in the `type` attribute of the Node Template), it can be mapped to an operation provided by a Relationship Template (i.e. an operation defined by the Relationship Type specified in the `type` attribute of the Relationship Template), or it can be mapped to a Plan of the Service Template.

When an exposed operation is invoked on a service instance created from the Service Template, the operation or Plan mapped to the exposed operation will actually be invoked.

The `Operation` element has the following properties:

- `name`: This attribute specifies the name of the operation, which **MUST** be unique within the containing interface.
- `NodeOperation`: This element specifies a reference to an operation of a Node Template. The `nodeRef` attribute of this element specifies a reference to the respective Node Template. The specific interface and operation to be mapped to the operation exposed by the Service Template are specified by means of the `interfaceName` and `operationName` attributes, respectively.

Note: An interface and operation with the specified names **MUST** be defined by the Node Type (or one of its super types) defined in the `type` attribute of the referenced Node Template.

- `RelationshipOperation`: This element specifies a reference to an operation of a Relationship Template. The `relationshipRef` attribute of this element specifies a reference to the respective Relationship Template. The specific interface and operation to be mapped to the operation exposed by the Service Template are specified by means of the `interfaceName` and `operationName` attributes, respectively.

Note: An interface and operation with the specified names **MUST** be defined by the Relationship Type (or one of its super types) defined in the `type` attribute of the referenced Relationship Template.

- `Plan`: This element specifies by means of its `planRef` attribute a reference to a Plan that provides the implementation of the operation exposed by the Service Template.

One of `NodeOperation`, `RelationshipOperation` or `Plan` **MUST be specified within the `Operation` element.**

924 • `TopologyTemplate`: This element specifies the overall structure of the cloud application
925 defined by the Service Template, i.e. the components it consists of, and the relations between
926 those components. The components of a service are referred to as *Node Templates*, the relations
927 between the components are referred to as *Relationship Templates*.

928 The `TopologyTemplate` element has the following properties:

929 ○ `NodeTemplate`: This element specifies a kind of a component making up the cloud
930 application.

931 The `NodeTemplate` element has the following properties:

932 ▪ `id`: This attribute specifies the identifier of the Node Template. The identifier of
933 the Node Template MUST be unique within the target namespace.

934 ▪ `name`: This OPTIONAL attribute specifies the name of the Node Template.

935 ▪ `type`: The QName value of this attribute refers to the Node Type providing the
936 type of the Node Template.

937
938 Note: If the Node Type referenced by the `type` attribute of a Node Template is
939 declared as abstract, no instances of the specific Node Template can be created.
940 Instead, a substitution of the Node Template with one having a specialized,
941 derived Node Type has to be done at the latest during the instantiation time of
942 the Node Template.

943 ▪ `minInstances`: This integer attribute specifies the minimum number of
944 instances to be created when instantiating the Node Template. The default value
945 of this attribute is 1. The value of `minInstances` MUST NOT be less than 0.

946 ▪ `maxInstances`: This attribute specifies the maximum number of instances that
947 can be created when instantiating the Node Template. The default value of this
948 attribute is 1. If the string is set to "unbounded", an unbounded number of
949 instances can be created. The value of `maxInstances` MUST be 1 or greater
950 and MUST NOT be less than the value specified for `minInstances`.

951 ▪ `Properties`: Specifies initial values for one or more of the Node Type
952 Properties of the Node Type providing the property definitions in the concrete
953 context of the Node Template.

954 The initial values are specified by providing an instance document of the XML
955 schema of the corresponding Node Type Properties. This instance document
956 considers the inheritance structure deduced by the `DerivedFrom` property of
957 the Node Type referenced by the `type` attribute of the Node Template.

958 The instance document of the XML schema might not validate against the
959 existence constraints of the corresponding schema: not all Node Type properties
960 might have an initial value assigned, i.e. mandatory elements or attributes might
961 be missing in the instance provided by the `Properties` element. Once the
962 defined Node Template has been instantiated, any XML representation of the
963 Node Type properties MUST validate according to the associated XML schema
964 definition.

965 ▪ `PropertyConstraints`: Specifies constraints on the use of one or more of
966 the Node Type Properties of the Node Type providing the property definitions for
967 the Node Template. Each constraint is specified by means of a separate nested
968 `PropertyConstraint` element.

969 The `PropertyConstraint` element has the following properties:

- 970
- 971
- 972
- 973
- `property`: The string value of this property is an XPath expression pointing to the property within the Node Type Properties document that is constrained within the context of the Node Template. More than one constraint MUST NOT be defined for each property.
 - `constraintType`: The constraint type is specified by means of a URI, which defines both the semantic meaning of the constraint as well as the format of the content.

974

975

976

977

978

979

980

981

982

983

For example, a constraint type of `http://www.example.com/PropertyConstraints/unique` could denote that the reference property of the node template under definition has to be unique within a certain scope. The constraint type specific content of the respective `PropertyConstraint` element could then define the actual scope in which uniqueness has to be ensured in more detail.

- 984
- 985
- 986
- 987
- 988
- `Requirements`: This element contains a list of requirements for the Node Template, according to the list of requirement definitions of the Node Type specified in the `type` attribute of the Node Template. Each requirement is specified in a separate nested `Requirement` element.

The `Requirement` Element has the following properties:

- 989
- 990
- 991
- `id`: This attribute specifies the identifier of the Requirement. The identifier of the Requirement MUST be unique within the target namespace.
 - `name`: This attribute specifies the name of the Requirement. The `name` and `type` of the Requirement MUST match the `name` and `type` of a Requirement Definition in the Node Type specified in the `type` attribute of the Node Template.
 - `type`: The QName value of this attribute refers to the Requirement Type definition of the Requirement. This Requirement Type denotes the semantics and well as potential properties of the Requirement.
 - `Properties`: This element specifies initial values for one or more of the Requirement Properties according to the Requirement Type providing the property definitions. Properties are provided in the form of an XML fragment. The same rules as outlined for the `Properties` element of the Node Template apply.
 - `PropertyConstraints`: This element specifies constraints on the use of one or more of the Properties of the Requirement Type providing the property definitions for the Requirement. Each constraint is specified by means of a separate nested `PropertyConstraint` element. The same rules as outlined for the `PropertyConstraints` element of the Node Template apply.

- 1000
- 1001
- 1002
- 1003
- 1004
- 1005
- 1006
- 1007
- 1008
- 1009
- `Capabilities`: This element contains a list of capabilities for the Node Template, according to the list of capability definitions of the Node Type specified in the `type` attribute of the Node Template. Each capability is specified in a separate nested `Capability` element.

The `Capability` Element has the following properties:

1010

1011

1012

1013

1014

- 1015
- 1016
- 1017
- 1018
- 1019
- 1020
- 1021
- 1022
- 1023
- 1024
- 1025
- 1026
- 1027
- 1028
- 1029
- 1030
- 1031
- 1032
- 1033
- 1034
- 1035
- 1036
- 1037
- 1038
- 1039
- 1040
- 1041
- 1042
- 1043
- 1044
- 1045
- 1046
- 1047
- 1048
- 1049
- 1050
- 1051
- 1052
- 1053
- 1054
- 1055
- 1056
- 1057
- 1058
- 1059
- 1060
- 1061
- `id`: This attribute specifies the identifier of the Capability. The identifier of the Capability MUST be unique within the target namespace.
 - `name`: This attribute specifies the name of the Capability. The name and type of the Capability MUST match the name and type of a Capability Definition in the Node Type specified in the type attribute of the Node Template.
 - `type`: The QName value of this attribute refers to the Capability Type definition of the Capability. This Capability Type denotes the semantics and well as potential properties of the Capability.
 - `Properties`: This element specifies initial values for one or more of the Capability Properties according to the Capability Type providing the property definitions. Properties are provided in the form of an XML fragment. The same rules as outlined for the Properties element of the Node Template apply.
 - `PropertyConstraints`: This element specifies constraints on the use of one or more of the Properties of the Capability Type providing the property definitions for the Capability. Each constraint is specified by means of a separate nested PropertyConstraint element. The same rules as outlined for the PropertyConstraints element of the Node Template apply.
 - `Policies`: This OPTIONAL element specifies policies associated with the Node Template. All Policies defined within the Policies element MUST be enforced by a TOSCA implementation, i.e. Policies are AND-combined. Each policy is specified by means of a separate nested Policy element. The Policy element has the following properties:
 - `name`: This OPTIONAL attribute allows for the definition of a name for the Policy. If specified, this name MUST be unique within the containing Policies element.
 - `policyType`: This attribute specifies the type of this Policy. The QName value of this attribute SHOULD correspond to the QName of a PolicyType defined in the same Definitions document or in an imported document.

The policyType attribute specifies the artifact type specific content of the Policy element body and indicates the type of Policy Template referenced by the Policy via the policyRef attribute.
 - `policyRef`: The QName value of this OPTIONAL attribute references a Policy Template that is associated to the Node Template. This Policy Template can be defined in the same TOSCA Definitions document, or it can be defined in a separate document that is imported into the current Definitions document. The type of Policy Template referenced by the policyRef attribute MUST be the same type or a sub-type of the type specified in the policyType attribute.

Note: if no Policy Template is referenced, the policy specific content of the Policy element alone is assumed to represent sufficient policy specific information in the context of the Node Template.

1062
1063
1064
1065
1066
1067
1068
1069

1070
1071
1072
1073
1074

1075
1076
1077

1078
1079
1080
1081
1082
1083
1084
1085
1086

1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107

1108
1109

Note: while Policy Templates provide invariant information about a non-functional behavior (i.e. information that is context independent, such as the availability class of an availability policy), the `Policy` element defined in a Node Template can provide variant information (i.e. information that is context specific, such as a specific heartbeat frequency for checking availability of a component) in the policy specific body of the `Policy` element.

- `DeploymentArtifacts`: This element specifies the deployment artifacts relevant for the Node Template under definition. Its nested `DeploymentArtifact` elements specify details about individual deployment artifacts.

The `DeploymentArtifact` element has the following properties:

- `name`: This attribute specifies the name of the artifact. Uniqueness of the name within the scope of the encompassing Node Template SHOULD be guaranteed by the definition.
- `artifactType`: This attribute specifies the type of this artifact. The `QName` value of this attribute SHOULD correspond to the `QName` of an `ArtifactType` defined in the same Definitions document or in an imported document.

The `artifactType` attribute specifies the artifact type specific content of the `DeploymentArtifact` element body and indicates the type of Artifact Template referenced by the Deployment Artifact via the `artifactRef` attribute.

- `artifactRef`: This OPTIONAL attribute contains a `QName` that identifies an Artifact Template to be used as deployment artifact. This Artifact Template can be defined in the same Definitions document or in a separate, imported document. The type of Artifact Template referenced by the `artifactRef` attribute MUST be the same type or a sub-type of the type specified in the `artifactType` attribute.

Note: if no Artifact Template is referenced, the artifact type specific content of the `DeploymentArtifact` element alone is assumed to represent the actual artifact. For example, the contents of a simple config file could be defined in place within the `DeploymentArtifact` element.

Note, that a deployment artifact specified with the Node Template under definition overrides any deployment artifact of the same `name` and the same `artifactType` (or any Artifact Type it is derived from) specified with the Node Type Implementation implementing the Node Type given as value of the `type` attribute of the Node Template under definition. Otherwise, the deployment artifacts of Node Type Implementations and the deployment artifacts defined with the Node Template are combined.

- `RelationshipTemplate`: This element specifies a kind of relationship between the components of the cloud application. For each specified Relationship Template the

1110 source element and target element MUST be specified in the Topology Template.
 1111 The `RelationshipTemplate` element has the following properties:

- 1112 ▪ `id`: This attribute specifies the identifier of the Relationship Template. The
 1113 identifier of the Relationship Template MUST be unique within the target
 1114 namespace.
- 1115 ▪ `name`: This OPTIONAL attribute specifies the name of the Relationship
 1116 Template.
- 1117 ▪ `type`: The QName value of this property refers to the Relationship Type
 1118 providing the type of the Relationship Template.

1119
 1120 Note: If the Relationship Type referenced by the `type` attribute of a Relationship
 1121 Template is declared as abstract, no instances of the specific Relationship
 1122 Template can be created. Instead, a substitution of the Relationship Template
 1123 with one having a specialized, derived Relationship Type has to be done at the
 1124 latest during the instantiation time of the Relationship Template.

- 1125 ▪ `Properties`: Specifies initial values for one or more of the Relationship Type
 1126 Properties of the Relationship Type providing the property definitions in the
 1127 concrete context of the Relationship Template.
 1128 The initial values are specified by providing an instance document of the XML
 1129 schema of the corresponding Relationship Type Properties. This instance
 1130 document considers the inheritance structure deduced by the `DerivedFrom`
 1131 property of the Relationship Type referenced by the `type` attribute of the
 1132 Relationship Template.
 1133 The instance document of the XML schema might not validate against the
 1134 existence constraints of the corresponding schema: not all Relationship Type
 1135 properties might have an initial value assigned, i.e. mandatory elements or
 1136 attributes might be missing in the instance provided by the `Properties`
 1137 element. Once the defined Relationship Template has been instantiated, any
 1138 XML representation of the Relationship Type properties MUST validate according
 1139 to the associated XML schema definition.
- 1140 ▪ `PropertyConstraints`: Specifies constraints on the use of one or more of
 1141 the Relationship Type Properties of the Relationship Type providing the property
 1142 definitions for the Relationship Template. Each constraint is specified by means
 1143 of a separate nested `PropertyConstraint` element.
 1144 The `PropertyConstraint` element has the following properties:
 - 1145 • `property`: The string value of this property is an XPath expression
 1146 pointing to the property within the Relationship Type Properties
 1147 document that is constrained within the context of the Relationship
 1148 Template. More than one constraint MUST NOT be defined for each
 1149 property.
 - 1150 • `constraintType`: The constraint type is specified by means of a URI,
 1151 which defines both the semantic meaning of the constraint as well as the
 1152 format of the content.

1153 For example, a constraint type of
 1154 <http://www.example.com/PropertyConstraints/unique> could denote that
 1155 the reference property of the node template under definition has to be
 1156

1157 unique within a certain scope. The constraint type specific content of the
1158 respective `PropertyConstraint` element could then define the
1159 actual scope in which uniqueness has to be ensured in more detail.

1160 ▪ `SourceElement`: This element specifies the origin of the relationship
1161 represented by the current `Relationship Template`.

1162 The `SourceElement` element has the following property:

- 1163 • `ref`: This attribute references by ID a Node Template or a Requirement
1164 of a Node Template within the same Service Template document that is
1165 the source of the `Relationship Template`.

1166
1167 If the `Relationship Type` referenced by the `type` attribute defines a
1168 constraint on the valid source of the relationship by means of its
1169 `ValidSource` element, the `ref` attribute of `SourceElement` MUST
1170 reference an object the type of which complies with the valid source
1171 constraint of the respective `Relationship Type`.

1172
1173 In the case where a Node Type is defined as valid source in the
1174 `Relationship Type` definition, the `ref` attribute MUST reference a Node
1175 Template of the corresponding Node Type (or of a sub-type).

1176
1177 In the case where a Requirement Type is defined a valid source in the
1178 `Relationship Type` definition, the `ref` attribute MUST reference a
1179 Requirement of the corresponding Requirement Type within a Node
1180 Template.

1181 ▪ `TargetElement`: This element specifies the target of the relationship
1182 represented by the current `Relationship Template`.

1183 The `TargetElement` element has the following property:

- 1184 • `ref`: This attribute references by ID a Node Template or a Capability of
1185 a Node Template within the same Service Template document that is the
1186 target of the `Relationship Template`.

1187
1188 If the `Relationship Type` referenced by the `type` attribute defines a
1189 constraint on the valid source of the relationship by means of its
1190 `ValidTarget` element, the `ref` attribute of `TargetElement` MUST
1191 reference an object the type of which complies with the valid source
1192 constraint of the respective `Relationship Type`.

1193
1194 In case a Node Type is defined as valid target in the `Relationship Type`
1195 definition, the `ref` attribute MUST reference a Node Template of the
1196 corresponding Node Type (or of a sub-type).

1197
1198 In case a Capability Type is defined a valid target in the `Relationship`
1199 `Type` definition, the `ref` attribute MUST reference a Capability of the
1200 corresponding Capability Type within a Node Template.

1201 ▪ `RelationshipConstraints`: This element specifies a list of constraints on
1202 the use of the relationship in separate nested `RelationshipConstraint`
1203 elements.

1204 The `RelationshipConstraint` element has the following properties:

- 1205
- 1206
- 1207
- 1208
- `constraintType`: This attribute specifies the type of relationship constraint by means of a URI. Depending on the type, the body of the `RelationshipConstraint` element might contain type specific content that further details the actual constraint.
- 1209
- `Plans`: This element specifies the operational behavior of the service. A `Plan` contained in the `Plans` element can specify how to create, terminate or manage the service.
- 1210
- 1211
- The `Plan` element has the following properties:
- `id`: This attribute specifies the identifier of the Plan. The identifier of the Plan MUST be unique within the target namespace.
 - `name`: This OPTIONAL attribute specifies the name of the Plan.
 - `planType`: The value of the attribute specifies the type of the plan as an indication on what the effect of executing the plan on a service will have. The plan type is specified by means of a URI, allowing for an extensibility mechanism for authors of service templates to define new plan types over time.
- 1212
- 1213
- 1214
- 1215
- 1216
- 1217
- 1218
- 1219
- The following plan types are defined as part of the TOSCA specification.
- <http://docs.oasis-open.org/tosca/ns/2011/12/PlanTypes/BuildPlan> - This URI defines the *build plan* plan type for plans used to initially create a new instance of a service from a Service Template.
 - <http://docs.oasis-open.org/tosca/ns/2011/12/PlanTypes/TerminationPlan> - This URI defines the *termination plan* plan type for plans used to terminate the existence of a service instance.
- 1220
- 1221
- 1222
- 1223
- 1224
- 1225
- 1226
- 1227
- 1228
- Note that all other plan types for managing service instances throughout their life time will be considered and referred to as *modification plans* in general.
- `planLanguage`: This attribute denotes the process modeling language (or metamodel) used to specify the plan. For example, “<http://www.omg.org/spec/BPMN/20100524/MODEL>” would specify that BPMN 2.0 has been used to model the plan.
- 1229
- 1230
- 1231
- 1232
- 1233
- 1234
- 1235
- 1236
- 1237
- TOSCA does not specify a separate metamodel for defining plans. Instead, it is assumed that a process modelling language (a.k.a. metamodel) like BPEL [BPEL 2.0] or BPMN [BPMN 2.0] is used to define plans. The specification favours the use of BPMN for modeling plans.
- `Precondition`: This OPTIONAL element specifies a condition that needs to be satisfied in order for the plan to be executed. The `expressionLanguage` attribute of this element specifies the expression language the nested condition is provided in.
- 1238
- 1239
- 1240
- 1241
- 1242
- 1243
- 1244
- 1245
- Typically, the precondition will be an expression in the instance state attribute of some of the node templates or relationship templates of the topology template. It will be evaluated based on the actual values of the corresponding attributes at the time the plan is requested to be executed. Note, that any other kind of pre-condition is allowed.
- `InputParameters`: This OPTIONAL property contains a list of one or more input parameter definitions for the Plan, each defined in a nested, separate `InputParameter` element.
- 1246
- 1247
- 1248
- 1249
- The `InputParameter` element has the following properties:

- 1250 ▪ name: This attribute specifies the name of the input parameter, which MUST be
- 1251 unique within the set of input parameters defined for the operation.
- 1252 ▪ type: This attribute specifies the type of the input parameter.
- 1253 ▪ required: This OPTIONAL attribute specifies whether or not the input
- 1254 parameter is REQUIRED (required attribute with a value of “yes” – default) or
- 1255 OPTIONAL (required attribute with a value of “no”).
- 1256 ○ OutputParameters: This OPTIONAL property contains a list of one or more output
- 1257 parameter definitions for the Plan, each defined in a nested, separate
- 1258 OutputParameter element.
- 1259 The OutputParameter element has the following properties:
 - 1260 ▪ name: This attribute specifies the name of the output parameter, which MUST be
 - 1261 unique within the set of output parameters defined for the operation.
 - 1262 ▪ type: This attribute specifies the type of the output parameter.
 - 1263 ▪ required: This OPTIONAL attribute specifies whether or not the output
 - 1264 parameter is REQUIRED (required attribute with a value of “yes” – default) or
 - 1265 OPTIONAL (required attribute with a value of “no”).
- 1266 ○ PlanModel: This property contains the actual model content.
- 1267 ○ PlanModelReference: This property points to the model content. Its reference
- 1268 attribute contains a URI of the model of the plan.
- 1269
- 1270 An instance of the Plan element MUST either contain the actual plan as instance of the
- 1271 PlanModel element, or point to the model via the PlanModelReference element.

1272 5.3 Example

1273 The following Service Template defines a Topology Template containing two Node Templates called
 1274 “MyApplication” and “MyAppServer”. These Node Templates have the node types “Application” and
 1275 “ApplicationServer”. The Node Template “MyApplication” is instantiated exactly once. Two of its Node
 1276 Type Properties are initialized by a corresponding Properties element. The Node Template
 1277 “MyAppServer” can be instantiated as many times as needed. The “MyApplication” Node Template is
 1278 connected with the “MyAppServer” Node Template via the Relationship Template named
 1279 “MyHostedRelationship”; the behavior and semantics of the Relationship Template is defined in the
 1280 Relationship Type “HostedOn”, saying that “MyApplication” is hosted on “MyAppServer”. The Service
 1281 Template further defines a Plan “UpdateApplication” for performing an update of the “MyApplication”
 1282 application hosted on the application server. This Plan refers to a BPMN 2.0 process definition contained
 1283 in a separate file.

```

1284 01 <ServiceTemplate id="MyService"
1285 02     name="My Service">
1286 03
1287 04   <TopologyTemplate>
1288 05
1289 06     <NodeTemplate id="MyApplication"
1290 07       name="My Application"
1291 08       type="my:Application">
1292 09       <Properties>
1293 10         <ApplicationProperties>
1294 11           <Owner>Frank</Owner>
1295 12           <InstanceName>Thomas' favorite application</InstanceName>
1296 13         </ApplicationProperties>
1297 14       </Properties>
  
```

```
1298 15     </NodeTemplate>
1299 16
1300 17     <NodeTemplate id="MyAppServer"
1301 18         name="My Application Server"
1302 19         type="my:ApplicationServer"
1303 20         minInstances="0"
1304 21         maxInstances="unbounded"/>
1305 22
1306 23     <RelationshipTemplate id="MyDeploymentRelationship"
1307 24         type="my:deployedOn">
1308 25         <SourceElement ref="MyApplication"/>
1309 26         <TargetElement ref="MyAppServer"/>
1310 27     </RelationshipTemplate>
1311 28
1312 29 </TopologyTemplate>
1313 30
1314 31 <Plans>
1315 32     <Plan id="UpdateApplication"
1316 33         planType="http://www.example.com/UpdatePlan"
1317 34         planLanguage="http://www.omg.org/spec/BPMN/20100524/MODEL">
1318 35         <PlanModelReference reference="plans:UpdateApp"/>
1319 36     </Plan>
1320 37 </Plans>
1321 38
1322 39 </ServiceTemplate>
```

1323

6 Node Types

1324 This chapter specifies how *Node Types* are defined. A Node Type is a reusable entity that defines the
1325 type of one or more Node Templates. As such, a Node Type defines the structure of observable
1326 properties via a *Properties Definition*, i.e. the names, data types and allowed values the properties
1327 defined in Node Templates using a Node Type or instances of such Node Templates can have.

1328 A Node Type can inherit properties from another Node Type by means of the `DerivedFrom` element.
1329 Node Types might be declared as abstract, meaning that they cannot be instantiated. The purpose of
1330 such abstract Node Types is to provide common properties and behavior for re-use in specialized,
1331 derived Node Types. Node Types might also be declared as final, meaning that they cannot be derived by
1332 other Node Types.

1333 A Node Type can declare to expose certain requirements and capabilities (see section 3.4) by means of
1334 `RequirementDefinition` elements or `CapabilityDefinition` elements, respectively.

1335 The functions that can be performed on (an instance of) a corresponding Node Template are defined by
1336 the *Interfaces* of the Node Type. Finally, management Policies are defined for a Node Type.

6.1 XML Syntax

1338 The following pseudo schema defines the XML syntax of Node Types:

```
1339 01 <NodeType name="xs:NCName" targetNamespace="xs:anyURI"?  
1340 02     abstract="yes|no"? final="yes|no"?>  
1341 03  
1342 04 <Tags>  
1343 05     <Tag name="xs:string" value="xs:string"/> +  
1344 06 </Tags> ?  
1345 07  
1346 08 <DerivedFrom typeRef="xs:QName"/> ?  
1347 09  
1348 10 <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?  
1349 11  
1350 12 <RequirementDefinitions>  
1351 13     <RequirementDefinition name="xs:string"  
1352 14         requirementType="xs:QName"  
1353 15         lowerBound="xs:integer"?  
1354 16         upperBound="xs:integer | xs:string"?>  
1355 17         <Constraints>  
1356 18             <Constraint constraintType="xs:anyURI">  
1357 19                 constraint type specific content  
1358 20             </Constraint> +  
1359 21         </Constraints> ?  
1360 22     </RequirementDefinition> +  
1361 23 </RequirementDefinitions> ?  
1362 24  
1363 25 <CapabilityDefinitions>  
1364 26     <CapabilityDefinition name="xs:string"  
1365 27         capabilityType="xs:QName"  
1366 28         lowerBound="xs:integer"?  
1367 29         upperBound="xs:integer | xs:string"?>  
1368 30         <Constraints>  
1369 31             <Constraint constraintType="xs:anyURI">  
1370 32                 constraint type specific content  
1371 33             </Constraint> +  
1372 34         </Constraints> ?
```

```

1373 35     </CapabilityDefinition> +
1374 36 </CapabilityDefinitions>
1375 37
1376 38 <InstanceStates>
1377 39     <InstanceState state="xs:anyURI"> +
1378 40 </InstanceStates> ?
1379 41
1380 42 <Interfaces>
1381 43     <Interface name="xs:NCName | xs:anyURI">
1382 44         <Operation name="xs:NCName">
1383 45             <InputParameters>
1384 46                 <InputParameter name="xs:string" type="xs:string"
1385 47                     required="yes|no"?/> +
1386 48             </InputParameters> ?
1387 49             <OutputParameters>
1388 50                 <OutputParameter name="xs:string" type="xs:string"
1389 51                     required="yes|no"?/> +
1390 52             </OutputParameters> ?
1391 53         </Operation> +
1392 54     </Interface> +
1393 55 </Interfaces> ?
1394 56
1395 57 </NodeType>

```

6.2 Properties

1396

1397 The `NodeType` element has the following properties:

- 1398 • `name`: This attribute specifies the name or identifier of the Node Type, which MUST be unique
1399 within the target namespace.
- 1400 • `targetNamespace`: This OPTIONAL attribute specifies the target namespace to which the
1401 definition of the Node Type will be added. If not specified, the Node Type definition will be added
1402 to the target namespace of the enclosing Definitions document.
- 1403 • `abstract`: This OPTIONAL attribute specifies that no instances can be created from Node
1404 Templates that use this Node Type as their type. If a Node Type includes a Requirement
1405 Definition or Capability Definition of an abstract Requirement Type or Capability Type,
1406 respectively, the Node Type MUST be declared as abstract as well.

1407
1408 As a consequence, the corresponding abstract Node Type referenced by any Node Template has
1409 to be substituted by a Node Type derived from the abstract Node Type at the latest during the
1410 instantiation time of a Node Template.

1411
1412 Note: an abstract Node Type MUST NOT be declared as final.

- 1413 • `final`: This OPTIONAL attribute specifies that other Node Types MUST NOT be derived from
1414 this Node Type.

1415
1416 Note: a final Node Type MUST NOT be declared as abstract.

- 1417 • `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by
1418 the author to describe the Node Type. Each tag is defined by a separate, nested `Tag` element.
1419 The `Tag` element has the following properties:

- 1420 ○ `name`: This attribute specifies the name of the tag.

- 1421 ○ `value`: This attribute specifies the value of the tag.

1422
1423 Note: The name/value pairs defined in tags have no normative interpretation.

- 1424
- 1425
- 1426
- 1427
- **DerivedFrom:** This is an OPTIONAL reference to another Node Type from which this Node Type derives. Conflicting definitions are resolved by the rule that local new definitions always override derived definitions. See section 6.3 Derivation Rules for details.
The `DerivedFrom` element has the following properties:
 - `typeRef`: The QName specifies the Node Type from which this Node Type derives its definitions.
 - **PropertiesDefinition:** This element specifies the structure of the observable properties of the Node Type, such as its configuration and state, by means of XML schema.
The `PropertiesDefinition` element has one but not both of the following properties:
 - `element`: This attribute provides the QName of an XML element defining the structure of the Node Type Properties.
 - `type`: This attribute provides the QName of an XML (complex) type defining the structure of the Node Type Properties.
 - **RequirementDefinitions:** This OPTIONAL element specifies the requirements that the Node Type exposes (see section 3.4 for details). Each requirement is defined in a nested `RequirementDefinition` element.
The `RequirementDefinition` element has the following properties:
 - `name`: This attribute specifies the name of the defined requirement and MUST be unique within the `RequirementDefinitions` of the current Node Type.

Note that one Node Type might define multiple requirements of the same Requirement Type, in which case each occurrence of a requirement definition is uniquely identified by its name. For example, a Node Type for an application might define two requirements for a database (i.e. of the same Requirement Type) where one could be named “customerDatabase” and the other one could be named “productsDatabase”.
 - `requirementType`: This attribute identifies by QName the Requirement Type that is being defined by the current `RequirementDefinition`.
 - `lowerBound`: This OPTIONAL attribute specifies the lower boundary by which a requirement MUST be matched for Node Templates according to the current Node Type, or for instances created for those Node Templates. The default value for this attribute is one. A value of zero would indicate that matching of the requirement is optional.
 - `upperBound`: This OPTIONAL attribute specifies the upper boundary by which a requirement MUST be matched for Node Templates according to the current Node Type, or for instances created for those Node Templates. The default value for this attribute is one. A value of “unbounded” indicates that there is no upper boundary.
 - `Constraints`: This OPTIONAL element contains a list of `Constraint` elements that specify additional constraints on the requirement definition. For example, if a database is needed a constraint on supported SQL features might be expressed.
The nested `Constraint` element has the following properties:
 - `constraintType`: This attribute specifies the type of constraint. According to this type, the body of the `Constraint` element will contain type specific content.
 - **CapabilityDefinitions:** This OPTIONAL element specifies the capabilities that the Node Type exposes (see section 3.4 for details). Each capability is defined in a nested `CapabilityDefinition` element.
The `CapabilityDefinition` element has the following properties:
 - `name`: This attribute specifies the name of the defined capability and MUST be unique within the `CapabilityDefinitions` of the current Node Type.

1473 Note that one Node Type might define multiple capabilities of the same Capability Type,
1474 in which case each occurrence of a capability definition is uniquely identified by its name.

- 1475 ○ `capabilityType`: This attribute identifies by QName the Capability Type of capability
1476 that is being defined by the current `CapabilityDefinition`.
- 1477 ○ `lowerBound`: This OPTIONAL attribute specifies the lower boundary of requiring nodes
1478 that the defined capability can serve. The default value for this attribute is one. A value of
1479 zero is invalid, since this would mean that the capability cannot actually satisfy any
1480 requiring nodes.
- 1481 ○ `upperBound`: This OPTIONAL attribute specifies the upper boundary of client
1482 requirements the defined capability can serve. The default value for this attribute is one.
1483 A value of "unbounded" indicates that there is no upper boundary.
- 1484 ○ `Constraints`: This OPTIONAL element contains a list of `Constraint` elements that
1485 specify additional constraints on the capability definition.
1486 The nested `Constraint` element has the following properties:
 - 1487 ▪ `constraintType`: This attribute specifies the type of constraint. According to
1488 this type, the body of the `Constraint` element will contain type specific
1489 content.
- 1490 • `InstanceStates`: This OPTIONAL element lists the set of states an instance of this Node
1491 Type can occupy. Those states are defined in nested `InstanceState` elements.
1492 The `InstanceState` element has the following nested properties:
 - 1493 ○ `state`: This attribute specifies a URI that identifies a potential state.
- 1494 • `Interfaces`: This element contains the definitions of the operations that can be performed on
1495 (instances of) this Node Type. Such operation definitions are given in the form of nested
1496 `Interface` elements.
1497 The `Interface` element has the following properties:
 - 1498 ○ `name`: The name of the interface. This name is either a URI or it is an NCName that
1499 MUST be unique in the scope of the Node Type being defined.
 - 1500 ○ `Operation`: This element defines an operation available to manage particular aspects
1501 of the Node Type.
1502
1503 The `Operation` element has the following properties:
 - 1504 ▪ `name`: This attribute defines the name of the operation and MUST be unique
1505 within the containing `Interface` of the Node Type.
 - 1506 ▪ `InputParameters`: This OPTIONAL property contains a list of one or more
1507 input parameter definitions, each defined in a nested, separate
1508 `InputParameter` element.
1509 The `InputParameter` element has the following properties:
 - 1510 • `name`: This attribute specifies the name of the input parameter, which
1511 MUST be unique within the set of input parameters defined for the
1512 operation.
 - 1513 • `type`: This attribute specifies the type of the input parameter.
 - 1514 • `required`: This OPTIONAL attribute specifies whether or not the input
1515 parameter is REQUIRED (`required` attribute with a value of "yes" –
1516 default) or OPTIONAL (`required` attribute with a value of "no").
 - 1517 ▪ `OutputParameters`: This OPTIONAL property contains a list of one or more
1518 output parameter definitions, each defined in a nested, separate
1519 `OutputParameter` element.
1520 The `OutputParameter` element has the following properties:

- 1521 • `name`: This attribute specifies the name of the output parameter, which
1522 MUST be unique within the set of output parameters defined for the
1523 operation.
- 1524 • `type`: This attribute specifies the type of the output parameter.
- 1525 • `required`: This OPTIONAL attribute specifies whether or not the
1526 output parameter is REQUIRED (`required` attribute with a value of
1527 “yes” – default) or OPTIONAL (`required` attribute with a value of “no”).

1528 6.3 Derivation Rules

1529 The following rules on combining definitions based on `DerivedFrom` apply:

- 1530 • **Node Type Properties:** It is assumed that the XML element (or type) representing the Node Type
1531 Properties extends the XML element (or type) of the Node Type Properties of the Node Type
1532 referenced in the `DerivedFrom` element.
- 1533 • **Requirements and capabilities:** The set of requirements or capabilities of the Node Type under
1534 definition consists of the set union of requirements or capabilities defined by the Node Type
1535 derived from and the requirements or capabilities defined by the Node Type under definition.
1536

1537 In cases where the Node Type under definition defines a requirement or capability with a certain
1538 name where the Node Type derived from already contains a respective definition with the same
1539 name, the definition in the Node Type under definition overrides the definition of the Node Type
1540 derived from. In such a case, the requirement definition or capability definition, respectively,
1541 MUST reference a Requirement Type or Capability Type that is derived from the one in the
1542 corresponding requirement definition or capability definition of the Node Type derived from.

- 1543 • **Instance States:** The set of instance states of the Node Type under definition consists of the set
1544 union of the instances states defined by the Nodes Type derived from and the instance states
1545 defined by the Node Type under definition. A set of instance states of the same name will be
1546 combined into a single instance state of the same name.
- 1547 • **Interfaces:** The set of interfaces of the Node Type under definition consists of the set union of
1548 interfaces defined by the Node Type derived from and the interfaces defined by the Node Type
1549 under definition.
1550 Two interfaces of the same name will be combined into a single, derived interface with the same
1551 name. The set of operations of the derived interface consists of the set union of operations
1552 defined by both interfaces. An operation defined by the Node Type under definition substitutes an
1553 operation with the same name of the Node Type derived from.

1554 6.4 Example

1555 The following example defines the Node Type “Project”. It is defined in a Definitions document
1556 “MyDefinitions” within the target namespace “http://www.example.com/sample”. Thus, by importing the
1557 corresponding namespace in another Definitions document, the Project Node Type is available for use in
1558 the other document.

```
1559 01 <Definitions id="MyDefinitions" name="My Definitions"
1560 02     targetNamespace="http://www.example.com/sample">
1561 03
1562 04   <NodeType name="Project">
1563 05
1564 06     <documentation xml:lang="EN">
1565 07       A reusable definition of a node type supporting
1566 08       the creation of new projects.
```

```

1567 09     </documentation>
1568 10
1569 11     <PropertiesDefinition element="ProjectProperties"/>
1570 12
1571 13     <InstanceStates>
1572 14         <InstanceState state="www.example.com/active"/>
1573 15         <InstanceState state="www.example.com/onHold"/>
1574 16     </InstanceStates>
1575 17
1576 18     <Interfaces>
1577 19         <Interface name="ProjectInterface">
1578 20             <Operation name="CreateProject">
1579 21                 <InputParameters>
1580 22                     <InputParamter name="ProjectName"
1581 23                         type="xs:string"/>
1582 24                     <InputParamter name="Owner"
1583 25                         type="xs:string"/>
1584 26                     <InputParamter name="AccountID"
1585 27                         type="xs:string"/>
1586 28                 </InputParameters>
1587 29             </Operation>
1588 30         </Interface>
1589 31     </Interfaces>
1590 32 </NodeType>
1591 33
1592 34 </Definitions>

```

1593 The Node Type "Project" has three Node Type Properties defined as an XML element in the `Types`
1594 element definition of the Service Template document: `Owner`, `ProjectName` and `AccountID` which are all
1595 of type `xs:string`. An instance of the Node Type "Project" could be "active" (more precise in state
1596 `www.example.com/active`) or "on hold" (more precise in state `www.example.com/onHold`). A single
1597 Interface is defined for this Node Type, and this Interface is defined by an Operation, i.e. its actual
1598 implementation is defined by the definition of the Operation. The Operation has the name `CreateProject`
1599 and three Input Parameters (exploiting the default value "yes" of the attribute `required` of the
1600 `InputParameter` element). The names of these Input Parameters are `ProjectName`, `Owner` and
1601 `AccountID`, all of type `xs:string`.

7 Node Type Implementations

1602

1603 This chapter specifies how *Node Type Implementations* are defined. A Node Type Implementation
1604 represents the executable code that implements a specific Node Type. It provides a collection of
1605 executables implementing the interface operations of a Node Type (aka implementation artifacts) and the
1606 executables needed to materialize instances of Node Templates referring to a particular Node Type (aka
1607 deployment artifacts). The respective executables are defined as separate Artifact Templates and are
1608 referenced from the implementation artifacts and deployment artifacts of a Node Type Implementation.

1609 While Artifact Templates provide invariant information about an artifact – i.e. information that is context
1610 independent like the file name of the artifact – implementation or deployment artifacts can provide variant
1611 (or context specific) information, such as authentication data or deployment paths for a specific
1612 environment.

1613 Node Type Implementations can specify hints for a TOSCA container that enable proper selection of an
1614 implementation that fits into a particular environment by means of Required Container Features
1615 definitions.

7.1 XML Syntax

1616

1617 The following pseudo schema defines the XML syntax of Node Type Implementations:

```
1618 01 <NodeTypeImplementation name="xs:NCName" targetNamespace="xs:anyURI"?  
1619 02     nodeType="xs:QName"  
1620 03     abstract="yes|no"?  
1621 04     final="yes|no"?>  
1622 05  
1623 06 <Tags>  
1624 07   <Tag name="xs:string" value="xs:string"/> +  
1625 08 </Tags> ?  
1626 09  
1627 10 <DerivedFrom nodeTypeImplementationRef="xs:QName"/> ?  
1628 11  
1629 12 <RequiredContainerFeatures>  
1630 13   <RequiredContainerFeature feature="xs:anyURI"/> +  
1631 14 </RequiredContainerFeatures> ?  
1632 15  
1633 16 <ImplementationArtifacts>  
1634 17   <ImplementationArtifact interfaceName="xs:NCName | xs:anyURI"?  
1635 18     operationName="xs:NCName"?  
1636 19     artifactType="xs:QName"  
1637 20     artifactRef="xs:QName"?>  
1638 21     artifact specific content ?  
1639 22   <ImplementationArtifact> +  
1640 23 </ImplementationArtifacts> ?  
1641 24  
1642 25 <DeploymentArtifacts>  
1643 26   <DeploymentArtifact name="xs:string" artifactType="xs:QName"  
1644 27     artifactRef="xs:QName"?>  
1645 28     artifact specific content ?  
1646 29   <DeploymentArtifact> +  
1647 30 </DeploymentArtifacts> ?  
1648 31  
1649 32 </NodeTypeImplementation>
```

1650 7.2 Properties

1651 The `NodeTypeImplementation` element has the following properties:

- 1652 • `name`: This attribute specifies the name or identifier of the Node Type Implementation, which
1653 MUST be unique within the target namespace.
- 1654 • `targetNamespace`: This OPTIONAL attribute specifies the target namespace to which the
1655 definition of the Node Type Implementation will be added. If not specified, the Node Type
1656 Implementation will be added to the target namespace of the enclosing Definitions document.
- 1657 • `nodeType`: The QName value of this attribute specifies the Node Type implemented by this
1658 Node Type Implementation.
- 1659 • `abstract`: This OPTIONAL attribute specifies that this Node Type Implementation cannot be
1660 used directly as an implementation for the Node Type specified in the `nodeType` attribute.

1661
1662 For example, a Node Type implementer might decide to deliver only part of the implementation of
1663 a specific Node Type (i.e. for only some operations) for re-use purposes and require the
1664 implementation for specific operations to be delivered in a more concrete, derived Node Type
1665 Implementation.

1666
1667 Note: an abstract Node Type Implementation MUST NOT be declared as final.

- 1668 • `final`: This OPTIONAL attribute specifies that other Node Type Implementations MUST NOT
1669 be derived from this Node Type Implementation.

1670
1671 Note: a final Node Type Implementation MUST NOT be declared as abstract.

- 1672 • `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by
1673 the author to describe the Node Type Implementation. Each tag is defined by a separate, nested
1674 Tag element.

1675 The Tag element has the following properties:

- 1676 ○ `name`: This attribute specifies the name of the tag.
- 1677 ○ `value`: This attribute specifies the value of the tag.

1678
1679 Note: The name/value pairs defined in tags have no normative interpretation.

- 1680 • `DerivedFrom`: This is an OPTIONAL reference to another Node Type Implementation from
1681 which this Node Type Implementation derives. See section 7.3 Derivation Rules **Error! Reference
1682 source not found.** for details.

1683 The `DerivedFrom` element has the following properties:

- 1684 ○ `nodeTypeImplementationRef`: The QName specifies the Node Type
1685 Implementation from which this Node Type Implementation derives.

- 1686 • `RequiredContainerFeatures`: An implementation of a Node Type might depend on
1687 certain features of the environment it is executed in, such as specific (potentially proprietary) APIs
1688 of the TOSCA container. For example, an implementation to deploy a virtual machine based on
1689 an image could require access to some API provided by a public cloud, while another
1690 implementation could require an API of a vendor-specific virtual image library. Thus, the contents
1691 of the `RequiredContainerFeatures` element provide “hints” to the TOSCA container
1692 allowing it to select the appropriate Node Type Implementation if multiple alternatives are
1693 provided.

1694 Each such dependency is defined by a separate `RequiredContainerFeature` element.

1695 The `RequiredContainerFeature` element has the following properties:

- 1696 ○ `feature`: The value of this attribute is a URI that denotes the corresponding needed
1697 feature of the environment.

- 1698
- `ImplementationArtifacts`: This element specifies a set of implementation artifacts for interfaces or operations of a Node Type.

1699

1700 The `ImplementationArtifacts` element has the following properties:

- 1701
- `ImplementationArtifact`: This element specifies one implementation artifact of an interface or an operation.

1702

1703

1704 Note: Multiple implementation artifacts might be needed to implement a Node Type according to the attributes defined below. An implementation artifact MAY serve as implementation for all interfaces and all operations defined for the Node Type, it MAY serve as implementation for one interface (and all its operations), or it MAY serve as implementation for only one specific operation.

1705

1706

1707

1708

1709

1710 The `ImplementationArtifact` element has the following properties:

- 1711
- `name`: This attribute specifies the name of the artifact, which SHOULD be unique within the scope of the encompassing Node Type Implementation.
- 1712
- `interfaceName`: This OPTIONAL attribute specifies the name of the interface that is implemented by the actual implementation artifact. If not specified, the implementation artifact is assumed to provide the implementation for all interfaces defined by the Node Type referred to by the `nodeType` attribute of the containing `NodeTypeImplementation`.
- 1713
- `operationName`: This OPTIONAL attribute specifies the name of the operation that is implemented by the actual implementation artifact. If specified, the `interfaceName` MUST be specified and the specified `operationName` MUST refer to an operation of the specified interface. If not specified, the implementation artifact is assumed to provide the implementation for all operations defined within the specified interface.
- 1714
- `artifactType`: This attribute specifies the type of this artifact. The QName value of this attribute SHOULD correspond to the QName of an `ArtifactType` defined in the same Definitions document or in an imported document.
- 1715
- 1716
- 1717

1718

1719

1720

1721

1722

1723

1724

1725

1726

1727

1728

1729 The `artifactType` attribute specifies the artifact type specific content of the `ImplementationArtifact` element body and indicates the type of `Artifact Template` referenced by the `Implementation Artifact` via the `artifactRef` attribute.

1730

1731

1732

- `artifactRef`: This OPTIONAL attribute contains a QName that identifies an `Artifact Template` to be used as implementation artifact. This `Artifact Template` can be defined in the same Definitions document or in a separate, imported document. The type of `Artifact Template` referenced by the `artifactRef` attribute MUST be the same type or a sub-type of the type specified in the `artifactType` attribute.

1733

1734

1735

1736

1737

1738

1739

1740

1741 Note: if no `Artifact Template` is referenced, the artifact type specific content of the `ImplementationArtifact` element alone is assumed to represent the actual artifact. For example, a simple script could be defined in place within the `ImplementationArtifact` element.

1742

1743

1744

- 1745
- `DeploymentArtifacts`: This element specifies a set of deployment artifacts relevant for materializing instances of nodes of the Node Type being implemented.

1746

1747 The `DeploymentArtifacts` element has the following properties:

- 1748
- `DeploymentArtifact`: This element specifies one deployment artifact.
- 1749

1750 Note: Multiple deployment artifacts MAY be defined in a Node Type Implementation. One
1751 reason could be that multiple artifacts (maybe of different types) are needed to
1752 materialize a node as a whole. Another reason could be that alternative artifacts are
1753 provided for use in different contexts (e.g. different installables of a software for use in
1754 different operating systems).

1755
1756 The `DeploymentArtifact` element has the following properties:

- 1757 ▪ `name`: This attribute specifies the name of the artifact, which SHOULD be unique
1758 within the scope of the encompassing Node Type Implementation.
- 1759 ▪ `artifactType`: This attribute specifies the type of this artifact. The QName
1760 value of this attribute SHOULD correspond to the QName of an
1761 `ArtifactType` defined in the same Definitions document or in an imported
1762 document.

1763
1764 The `artifactType` attribute specifies the artifact type specific content of the
1765 `DeploymentArtifact` element body and indicates the type of Artifact
1766 Template referenced by the Deployment Artifact via the `artifactRef`
1767 attribute.

- 1768 ▪ `artifactRef`: This OPTIONAL attribute contains a QName that identifies an
1769 Artifact Template to be used as deployment artifact. This Artifact Template can
1770 be defined in the same Definitions document or in a separate, imported
1771 document.

1772 The type of Artifact Template referenced by the `artifactRef` attribute MUST
1773 be the same type or a sub-type of the type specified in the `artifactType`
1774 attribute.

1775
1776 Note: if no Artifact Template is referenced, the artifact type specific content of the
1777 `DeploymentArtifact` element alone is assumed to represent the actual
1778 artifact. For example, the contents of a simple config file could be defined in
1779 place within the `DeploymentArtifact` element.

1780 7.3 Derivation Rules

1781 The following rules on combining definitions based on `DerivedFrom` apply:

- 1782 • **Implementation Artifacts:** The set of implementation artifacts of a Node Type Implementation
1783 consists of the set union of implementation artifacts defined by the Node Type Implementation
1784 itself and the implementation artifacts defined by any Node Type Implementation the Node Type
1785 Implementation is derived from.
1786 An implementation artifact defined by a Node Type Implementation overrides an implementation
1787 artifact having the same interface name and operation name of a Node Type Implementation the
1788 Node Type Implementation is derived from.
1789 If an implementation artifact defined in a Node Type Implementation specifies only an interface
1790 name, it substitutes implementation artifacts having the same interface name (with or without an
1791 operation name defined) of any Node Type Implementation the Node Type Implementation is
1792 derived from. In this case, the implementation of a complete interface of a Node Type is
1793 overridden.
1794 If an implementation artifact defined in a Node Type Implementation neither defines an interface
1795 name nor an operation name, it overrides all implementation artifacts of any Node Type
1796 Implementation the Node Type Implementation is derived from. In this case, the complete
1797 implementation of a Node Type is overridden.

- Deployment Artifacts: The set of deployment artifacts of a Node Type Implementation consists of the set union of the deployment artifacts defined by the Nodes Type Implementation itself and the deployment artifacts defined by any Node Type Implementation the Node Type Implementation is derived from. A deployment artifact defined by a Node Type Implementation overrides a deployment artifact with the same name and type (or any type it is derived from) of any Node Type Implementation the Node Type Implementation is derived from.

7.4 Example

The following example defines the Node Type Implementation “MyDBMSImplementation”. This is an implementation of a Node Type “DBMS”.

```

1807 01 <Definitions id="MyImpls" name="My Implementations"
1808 02   targetNamespace="http://www.example.com/SampleImplementations"
1809 03   xmlns:bn="http://www.example.com/BaseNodeTypes"
1810 04   xmlns:ba="http://www.example.com/BaseArtifactTypes"
1811 05   xmlns:sa="http://www.example.com/SampleArtifacts">
1812 06
1813 07   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
1814 08     namespace="http://www.example.com/BaseArtifactTypes"/>
1815 09
1816 10   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
1817 11     namespace="http://www.example.com/BaseNodeTypes"/>
1818 12
1819 13   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
1820 14     namespace="http://www.example.com/SampleArtifacts"/>
1821 15
1822 16   <NodeTypeImplementation name="MyDBMSImplementation"
1823 17     nodeType="bn:DBMS">
1824 18
1825 19     <ImplementationArtifacts>
1826 20       <ImplementationArtifact interfaceName="MgmtInterface"
1827 21         artifactType="ba:WARFile"
1828 22         artifactRef="sa:MyMgmtWebApp">
1829 23       </ImplementationArtifact>
1830 24     </ImplementationArtifacts>
1831 25
1832 26     <DeploymentArtifacts>
1833 27       <DeploymentArtifact name="MyDBMS"
1834 28         artifactType="ba:ZipFile"
1835 29         artifactRef="sa:MyInstallable">
1836 30       </DeploymentArtifact>
1837 31     </DeploymentArtifacts>
1838 32
1839 33   </NodeTypeImplementation>
1840 34
1841 35 </Definitions>

```

The Node Type Implementation contains the “MyDBMSManagement” implementation artifact, which is an artifact for the “MgmtInterface” Interface that has been defined for the “DBMS” base Node Type. The type of this artifact is a “WARFile” that has been defined as base Artifact Type. The implementation artifact refers to the “MyMgmtWebApp” Artifact Template that has been defined before.

The Node Type Implementation further contains the “MyDBMS” deployment artifact, which is a software installable used for instantiating the “DBMS” Node Type. This software installable is a “ZipFile” that has been separately defined as the “MyInstallable” Artifact Template before.

1849

8 Relationship Types

1850 This chapter specifies how *Relationship Types* are defined. A Relationship Type is a reusable entity that
1851 defines the type of one or more Relationship Templates between Node Templates. As such, a
1852 Relationship Type can define the structure of observable properties via a *Properties Definition*, i.e. the
1853 names, data types and allowed values the properties defined in Relationship Templates using a
1854 Relationship Type or instances of such Relationship Templates can have.

1855 The operations that can be performed on (an instance of) a corresponding Relationship Template are
1856 defined by the *Interfaces* of the Relationship Type. Furthermore, a Relationship Type defines the potential
1857 states an instance of it might reveal at runtime.

1858 A Relationship Type can inherit the definitions defined in another Relationship Type by means of the
1859 *DerivedFrom* element. Relationship Types might be declared as abstract, meaning that they cannot be
1860 instantiated. The purpose of such abstract Relationship Types is to provide common properties and
1861 behavior for re-use in specialized, derived Relationship Types. Relationship Types might also be declared
1862 as final, meaning that they cannot be derived by other Relationship Types.

1863 8.1 XML Syntax

1864 The following pseudo schema defines the XML syntax of Relationship Types:

```
1865 01 <RelationshipType name="xs:NCName"
1866 02           targetNamespace="xs:anyURI"?
1867 03           abstract="yes|no"?
1868 04           final="yes|no"?> +
1869 05
1870 06   <Tags>
1871 07     <Tag name="xs:string" value="xs:string"/> +
1872 08   </Tags> ?
1873 09
1874 10   <DerivedFrom typeRef="xs:QName"/> ?
1875 11
1876 12   <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
1877 13
1878 14   <InstanceStates>
1879 15     <InstanceState state="xs:anyURI"> +
1880 16   </InstanceStates> ?
1881 17
1882 18   <SourceInterfaces>
1883 19     <Interface name="xs:NCName | xs:anyURI">
1884 20       ...
1885 21     </Interface> +
1886 22   </SourceInterfaces> ?
1887 23
1888 24   <TargetInterfaces>
1889 25     <Interface name="xs:NCName | xs:anyURI">
1890 26       ...
1891 27     </Interface> +
1892 28   </TargetInterfaces> ?
1893 29
1894 30   <ValidSource typeRef="xs:QName"/> ?
1895 31
1896 32   <ValidTarget typeRef="xs:QName"/> ?
1897 33
1898 34 </RelationshipType>
```

1899 8.2 Properties

1900 The `RelationshipType` element has the following properties:

- 1901 • `name`: This attribute specifies the name or identifier of the Relationship Type, which MUST be
1902 unique within the target namespace.
- 1903 • `targetNamespace`: This OPTIONAL attribute specifies the target namespace to which the
1904 definition of the Relationship Type will be added. If not specified, the Relationship Type definition
1905 will be added to the target namespace of the enclosing Definitions document.
- 1906 • `abstract`: This OPTIONAL attribute specifies that no instances can be created from
1907 Relationship Templates that use this Relationship Type as their type.

1908
1909 As a consequence, the corresponding abstract Relationship Type referenced by any Relationship
1910 Template has to be substituted by a Relationship Type derived from the abstract Relationship
1911 Type at the latest during the instantiation time of a Relationship Template.

1912
1913 Note: an abstract Relationship Type MUST NOT be declared as final.

- 1914 • `final`: This OPTIONAL attribute specifies that other Relationship Types MUST NOT be derived
1915 from this Relationship Type.

1916
1917 Note: a final Relationship Type MUST NOT be declared as abstract.

- 1918 • `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by
1919 the author to describe the Relationship Type. Each tag is defined by a separate, nested `Tag`
1920 element.

1921 The `Tag` element has the following properties:

- 1922 ○ `name`: This attribute specifies the name of the tag.
- 1923 ○ `value`: This attribute specifies the value of the tag.

1924
1925 Note: The name/value pairs defined in tags have no normative interpretation.

- 1926 • `DerivedFrom`: This is an OPTIONAL reference to another Relationship Type from which this
1927 Relationship Type is derived. Conflicting definitions are resolved by the rule that local new
1928 definitions always override derived definitions. See section 8.3 Derivation Rules for details.

1929 The `DerivedFrom` element has the following properties:

- 1930 ○ `typeRef`: The QName specifies the Relationship Type from which this Relationship
1931 Type derives its definitions.

- 1932 • `PropertiesDefinition`: This element specifies the structure of the observable properties
1933 of the Relationship Type, such as its configuration and state, by means of XML schema.

1934 The `PropertiesDefinition` element has one but not both of the following properties:

- 1935 ○ `element`: This attribute provides the QName of an XML element defining the structure
1936 of the Relationship Type Properties.
- 1937 ○ `type`: This attribute provides the QName of an XML (complex) type defining the
1938 structure of the Relationship Type Properties.

- 1939 • `InstanceStates`: This OPTIONAL element lists the set of states an instance of this
1940 Relationship Type can occupy at runtime. Those states are defined in nested `InstanceState`
1941 elements.

1942 The `InstanceState` element has the following nested properties:

- 1943 ○ `state`: This attribute specifies a URI that identifies a potential state.

- 1944 • `SourceInterfaces`: This OPTIONAL element contains definitions of manageability interfaces
1945 that can be performed on the source of a relationship of this Relationship Type to actually
1946 establish the relationship between the source and the target in the deployed service.

1947 Those interface definitions are contained in nested `Interface` elements, the content of which
1948 is that described for Node Type interfaces (see section 6.2).

1949 • `TargetInterfaces`: This OPTIONAL element contains definitions of manageability interfaces
1950 that can be performed on the target of a relationship of this Relationship Type to actually
1951 establish the relationship between the source and the target in the deployed service.

1952 Those interface definitions are contained in nested `Interface` elements, the content of which
1953 is that described for Node Type interfaces (see section 6.2).

1954 • `ValidSource`: This OPTIONAL element specifies the type of object that is allowed as a valid
1955 origin for relationships defined using the Relationship Type under definition. If not specified, any
1956 Node Type is allowed to be the origin of the relationship.

1957 The `ValidSource` element has the following properties:

1958 ○ `typeRef`: This attribute specifies the QName of a Node Type or Requirement Type that
1959 is allowed as a valid source for relationships defined using the Relationship Type under
1960 definition. Node Types or Requirements Types derived from the specified Node Type or
1961 Requirement Type, respectively, MUST also be accepted as valid relationship source.
1962

1963 Note: If `ValidSource` specifies a Node Type, the `ValidTarget` element (if present)
1964 of the Relationship Type under definition MUST also specify a Node Type.

1965 If `ValidSource` specifies a Requirement Type, the `ValidTarget` element (if
1966 present) of the Relationship Type under definition MUST specify a Capability Type. This
1967 Capability Type MUST match the requirement defined in `ValidSource`, i.e. it MUST
1968 be of the type (or a sub-type of) the capability specified in the
1969 `requiredCapabilityType` attribute of the respective `RequirementType`
1970 definition.

1971 • `ValidTarget`: This OPTIONAL element specifies the type of object that is allowed as a valid
1972 target for relationships defined using the Relationship Type under definition. If not specified, any
1973 Node Type is allowed to be the origin of the relationship.

1974 The `ValidTarget` element has the following properties:

1975 ○ `typeRef`: This attribute specifies the QName of a Node Type or Capability Type that is
1976 allowed as a valid target for relationships defined using the Relationship Type under
1977 definition. Node Types or Capability Types derived from the specified Node Type or
1978 Capability Type, respectively, MUST also be accepted as valid targets of relationships.
1979

1980 Note: If `ValidTarget` specifies a Node Type, the `ValidSource` element (if present)
1981 of the Relationship Type under definition MUST also specify a Node Type.

1982 If `ValidTarget` specifies a Capability Type, the `ValidSource` element (if present)
1983 of the Relationship Type under definition MUST specify a Requirement Type. This
1984 Requirement Type MUST declare it requires the capability defined in `ValidTarget`,
1985 i.e. it MUST declare the type (or a super-type of) the capability in the
1986 `requiredCapabilityType` attribute of the respective `RequirementType`
1987 definition.

1988 8.3 Derivation Rules

1989 The following rules on combining definitions based on `DerivedFrom` apply:

1990 • `Relationship Type Properties`: It is assumed that the XML element (or type) representing the
1991 Relationship Type properties of the Relationship Type under definition extends the XML element
1992 (or type) of the Relationship Type properties of the Relationship Type referenced in the
1993 `DerivedFrom` element.

1994 • `Instance States`: The resulting set of instance states of the Relationship Type under definition
1995 consists of the set union of the instances states defined by the Relationship Type derived from

- 1996 and the instance states explicitly defined by the Relationship Type under definition. Instance
 1997 states with the same state attribute will be combined into a single instance state of the same
 1998 state.
- 1999 • Valid source and target: An object specified as a valid source or target, respectively, of the
 2000 Relationship Type under definition MUST be of a subtype defined as valid source or target,
 2001 respectively, of the Relationship Type derived from.
- 2002
- 2003 If the Relationship Type derived from has no valid source or target defined, the types of object
 2004 being defined in the `ValidSource` or `ValidTarget` elements of the Relationship Type
 2005 under definition are not restricted.
- 2006
- 2007 If the Relationship Type under definition has no source or target defined, only the types of objects
 2008 defined as source or target of the Relationship Type derived from are valid origins or destinations
 2009 of the Relationship Type under definition.
- 2010 • Interfaces: The set of interfaces (both source and target interfaces) of the Relationship Type
 2011 under definition consists of the set union of interfaces defined by the Relationship Type derived
 2012 from and the interfaces defined by the Relationship Type under definition.
 2013 Two interfaces of the same name will be combined into a single, derived interface with the same
 2014 name. The set of operations of the derived interface consists of the set union of operations
 2015 defined by both interfaces. An operation defined by the Relationship Type under definition
 2016 substitutes an operation with the same name of the Relationship Type derived from.

2017 8.4 Example

2018 The following example defines the Relationship Type “processDeployedOn”. The meaning of this
 2019 Relationship Type is that “a process is deployed on a hosting environment”. When the source of an
 2020 instance of a Relationship Template referring to this Relationship Type is deleted, its target is
 2021 automatically deleted as well. The Relationship Type has Relationship Type Properties defined in the
 2022 `Types` section of the same Definitions document as the “ProcessDeployedOnProperties” element. The
 2023 states an instance of this Relationship Type can be in are also listed.

```

2024 01 <RelationshipType name="processDeployedOn">
2025 02
2026 03   <RelationshipTypeProperties element="ProcessDeployedOnProperties"/>
2027 04
2028 05   <InstanceStates>
2029 06     <InstanceState state="www.example.com/successfullyDeployed"/>
2030 07     <InstanceState state="www.example.com/failed"/>
2031 08   </InstanceStates>
2032 09
2033 10 </RelationshipType>
  
```

2034

9 Relationship Type Implementations

2035 This chapter specifies how *Relationship Type Implementations* are defined. A Relationship Type
2036 Implementation represents the runnable code that implements a specific Relationship Type. It provides a
2037 collection of executables implementing the interface operations of a Relationship Type (aka
2038 implementation artifacts). The particular executables are defined as separate Artifact Templates and are
2039 referenced from the implementation artifacts of a Relationship Type Implementation.

2040 While Artifact Templates provide invariant information about an artifact – i.e. information that is context
2041 independent like the file name of the artifact – implementation artifacts can provide variant (or context
2042 specific) information, e.g. authentication data for a specific environment.

2043 Relationship Type Implementations can specify hints for a TOSCA container that enable proper selection
2044 of an implementation that fits into a particular environment by means of Required Container Features
2045 definitions.

2046 Note that there MAY be Relationship Types that do not define any interface operations, i.e. that also do
2047 not require any implementation artifacts. In such cases, no Relationship Type Implementation is needed
2048 but the respective Relationship Types can be used by a TOSCA implementation as is.

9.1 XML Syntax

2049 The following pseudo schema defines the XML syntax of Relationship Type Implementations:

```
2051 01 <RelationshipTypeImplementation name="xs:NCName"
2052 02     targetNamespace="xs:anyURI"?
2053 03     relationshipType="xs:QName"
2054 04     abstract="yes|no"?
2055 05     final="yes|no"?>
2056 06
2057 07 <Tags>
2058 08   <Tag name="xs:string" value="xs:string" /> +
2059 09 </Tags> ?
2060 10
2061 11 <DerivedFrom relationshipTypeImplementationRef="xs:QName" /> ?
2062 12
2063 13 <RequiredContainerFeatures>
2064 14   <RequiredContainerFeature feature="xs:anyURI" /> +
2065 15 </RequiredContainerFeatures> ?
2066 16
2067 17 <ImplementationArtifacts>
2068 18   <ImplementationArtifact interfaceName="xs:NCName | xs:anyURI"?
2069 19     operationName="xs:NCName"?
2070 20     artifactType="xs:QName"
2071 21     artifactRef="xs:QName"?>
2072 22     artifact specific content ?
2073 23   <ImplementationArtifact> +
2074 24 </ImplementationArtifacts> ?
2075 25
2076 26 </RelationshipTypeImplementation>
```

9.2 Properties

2077 The RelationshipTypeImplementation element has the following properties:

- 2078 • name: This attribute specifies the name or identifier of the Relationship Type Implementation,
2080 which MUST be unique within the target namespace.

2081 • `targetNamespace`: This OPTIONAL attribute specifies the target namespace to which the
2082 definition of the Relationship Type Implementation will be added. If not specified, the Relationship
2083 Type Implementation will be added to the target namespace of the enclosing Definitions
2084 document.

2085 • `relationshipType`: The QName value of this attribute specifies the Relationship Type
2086 implemented by this Relationship Type Implementation.

2087 • `abstract`: This OPTIONAL attribute specifies that this Relationship Type Implementation
2088 cannot be used directly as an implementation for the Relationship Type specified in the
2089 `relationshipType` attribute.

2090
2091 For example, a Relationship Type implementer might decide to deliver only part of the
2092 implementation of a specific Relationship Type (i.e. for only some operations) for re-use purposes
2093 and require the implementation for specific operations to be delivered in a more concrete, derived
2094 Relationship Type Implementation.

2095 Note: an abstract Relationship Type Implementation MUST NOT be declared as final.

2097 • `final`: This OPTIONAL attribute specifies that other Relationship Type Implementations MUST
2098 NOT be derived from this Relationship Type Implementation.

2099 Note: a final Relationship Type Implementation MUST NOT be declared as abstract.

2101 • `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by
2102 the author to describe the Relationship Type Implementation. Each tag is defined by a separate,
2103 nested `Tag` element.

2104 The `Tag` element has the following properties:

2105 o `name`: This attribute specifies the name of the tag.

2106 o `value`: This attribute specifies the value of the tag.

2107
2108 Note: The name/value pairs defined in tags have no normative interpretation.

2109 • `DerivedFrom`: This is an OPTIONAL reference to another Relationship Type Implementation
2110 from which this Relationship Type Implementation derives. See section 9.3 Derivation Rules or
2111 details.

2112 The `DerivedFrom` element has the following properties:

2113 o `relationshipTypeImplementationRef`: The QName specifies the Relationship
2114 Type Implementation from which this Relationship Type Implementation derives.

2115 • `RequiredContainerFeatures`: An implementation of a Relationship Type might depend
2116 on certain features of the environment it is executed in, such as specific (potentially proprietary)
2117 APIs of the TOSCA container.

2118 Thus, the contents of the `RequiredContainerFeatures` element provide “hints” to the
2119 TOSCA container allowing it to select the appropriate Relationship Type Implementation if
2120 multiple alternatives are provided.

2121 Each such dependency is defined by a separate `RequiredContainerFeature` element.

2122 The `RequiredContainerFeature` element has the following properties:

2123 o `feature`: The value of this attribute is a URI that denotes the corresponding needed
2124 feature of the environment.

2125 • `ImplementationArtifacts`: This element specifies a set of implementation artifacts for
2126 interfaces or operations of a Relationship Type.

2127 The `ImplementationArtifacts` element has the following properties:

2128 o `ImplementationArtifact`: This element specifies one implementation artifact of
2129 an interface or an operation.

2130

2131 Note: Multiple implementation artifacts might be needed to implement a Relationship
2132 Type according to the attributes defined below. An implementation artifact MAY serve as
2133 implementation for all interfaces and all operations defined for the Relationship Type, it
2134 MAY serve as implementation for one interface (and all its operations), or it MAY serve
2135 as implementation for only one specific operation.
2136

2137 The `ImplementationArtifact` element has the following properties:

- 2138 ▪ `name`: This attribute specifies the name of the artifact, which SHOULD be unique
2139 within the scope of the encompassing Node Type Implementation.
- 2140 ▪ `interfaceName`: This OPTIONAL attribute specifies the name of the interface
2141 that is implemented by the actual implementation artifact. If not specified, the
2142 implementation artifact is assumed to provide the implementation for all
2143 interfaces defined by the Relationship Type referred to by the
2144 `relationshipType` attribute of the containing
2145 `RelationshipTypeImplementation`.

2146 Note that the referenced interface can be defined in either the
2147 `SourceInterfaces` element or the `TargetInterfaces` element of the
2148 Relationship Type implemented by this Relationship Type Implementation.
2149

- 2150 ▪ `operationName`: This OPTIONAL attribute specifies the name of the
2151 operation that is implemented by the actual implementation artifact. If specified,
2152 the `interfaceName` MUST be specified and the specified `operationName`
2153 MUST refer to an operation of the specified interface. If not specified, the
2154 implementation artifact is assumed to provide the implementation for all
2155 operations defined within the specified interface.
- 2156 ▪ `artifactType`: This attribute specifies the type of this artifact. The QName
2157 value of this attribute SHOULD correspond to the QName of an
2158 `ArtifactType` defined in the same Definitions document or in an imported
2159 document.

2160 The `artifactType` attribute specifies the artifact type specific content of the
2161 `ImplementationArtifact` element body and indicates the type of Artifact
2162 Template referenced by the Implementation Artifact via the `artifactRef`
2163 attribute.
2164

- 2165 ▪ `artifactRef`: This OPTIONAL attribute contains a QName that identifies an
2166 Artifact Template to be used as implementation artifact. This Artifact Template
2167 can be defined in the same Definitions document or in a separate, imported
2168 document.
2169 The type of Artifact Template referenced by the `artifactRef` attribute MUST
2170 be the same type or a sub-type of the type specified in the `artifactType`
2171 attribute.

2172 Note: if no Artifact Template is referenced, the artifact type specific content of the
2173 `ImplementationArtifact` element alone is assumed to represent the
2174 actual artifact. For example, a simple script could be defined in place within the
2175 `ImplementationArtifact` element.
2176

2177 9.3 Derivation Rules

2178 The following rules on combining definitions based on `DerivedFrom` apply:

- 2179 • Implementation Artifacts: The set of implementation artifacts of a Relationship Type
2180 Implementation consists of the set union of implementation artifacts defined by the Relationship

2181 Type Implementation itself and the implementation artifacts defined by any Relationship Type
 2182 Implementation the Relationship Type Implementation is derived from.
 2183 An implementation artifact defined by a Node Type Implementation overrides an implementation
 2184 artifact having the same interface name and operation name of a Relationship Type
 2185 Implementation the Relationship Type Implementation is derived from.
 2186 If an implementation artifact defined in a Relationship Type Implementation specifies only an
 2187 interface name, it substitutes implementation artifacts having the same interface name (with or
 2188 without an operation name defined) of any Relationship Type Implementation the Relationship
 2189 Type Implementation is derived from. In this case, the implementation of a complete interface of a
 2190 Relationship Type is overridden.
 2191 If an implementation artifact defined in a Relationship Type Implementation neither defines an
 2192 interface name nor an operation name, it overrides all implementation artifacts of any
 2193 Relationship Type Implementation the Relationship Type Implementation is derived from. In this
 2194 case, the complete implementation of a Relationship Type is overridden.

2195 9.4 Example

2196 The following example defines the Node Type Implementation “MyDBMSImplementation”. This is an
 2197 implementation of a Node Type “DBMS”.

```

2198 01 <Definitions id="MyImpls" name="My Implementations"
2199 02   targetNamespace="http://www.example.com/SampleImplementations"
2200 03   xmlns:bn="http://www.example.com/BaseRelationshipTypes"
2201 04   xmlns:ba="http://www.example.com/BaseArtifactTypes"
2202 05   xmlns:sa="http://www.example.com/SampleArtifacts">
2203 06
2204 07   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
2205 08     namespace="http://www.example.com/BaseArtifactTypes"/>
2206 09
2207 10   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
2208 11     namespace="http://www.example.com/BaseRelationshipTypes"/>
2209 12
2210 13   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
2211 14     namespace="http://www.example.com/SampleArtifacts"/>
2212 15
2213 16   <RelationshipTypeImplementation name="MyDBConnectImplementation"
2214 17     relationshipType="bn:DBConnection">
2215 18
2216 19     <ImplementationArtifacts>
2217 20       <ImplementationArtifact interfaceName="ConnectionInterface"
2218 21         operationName="connectTo"
2219 22         artifactType="ba:ScriptArtifact"
2220 23         artifactRef="sa:MyConnectScript">
2221 24       <ImplementationArtifact>
2222 25     </ImplementationArtifacts>
2223 26
2224 27   </RelationshipTypeImplementation>
2225 28
2226 29 </Definitions>
  
```

2227 The Relationship Type Implementation contains the “MyDBConnectionImpl” implementation artifact,
 2228 which is an artifact for the “ConnectionInterface” interface that has been defined for the “DBConnection”
 2229 base Relationship Type. The type of this artifact is a “ScriptArtifact” that has been defined as base Artifact
 2230 Type. The implementation artifact refers to the “MyConnectScript” Artifact Template that has been defined
 2231 before.

2232

10 Requirement Types

2233 This chapter specifies how *Requirement Types* are defined. A Requirement Type is a reusable entity that
2234 describes a kind of requirement that a Node Type can declare to expose. For example, a Requirement
2235 Type for a database connection can be defined and various Node Types (e.g. a Node Type for an
2236 application) can declare to expose (or “to have”) a requirement for a database connection.

2237 A Requirement Type defines the structure of observable properties via a *Properties Definition*, i.e. the
2238 names, data types and allowed values the properties defined in *Requirements* of Node Templates of a
2239 Node Type can have in cases where the Node Type defines a requirement of the respective Requirement
2240 Type.

2241 A Requirement Type can inherit properties and semantics from another Requirement Type by means of
2242 the *DerivedFrom* element. Requirement Types might be declared as abstract, meaning that they
2243 cannot be instantiated. The purpose of such abstract Requirement Types is to provide common
2244 properties for re-use in specialized, derived Requirement Types. Requirement Types might also be
2245 declared as final, meaning that they cannot be derived by other Requirement Types.

10.1 XML Syntax

2247 The following pseudo schema defines the XML syntax of Requirement Types:

```
2248 01 <RequirementType name="xs:NCName"  
2249 02         targetNamespace="xs:anyURI"?  
2250 03         abstract="yes|no"?  
2251 04         final="yes|no"?  
2252 05         requiredCapabilityType="xs:QName"?>  
2253 06  
2254 07 <Tags>  
2255 08     <Tag name="xs:string" value="xs:string"/> +  
2256 09 </Tags> ?  
2257 10  
2258 11 <DerivedFrom typeRef="xs:QName"/> ?  
2259 12  
2260 13 <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?  
2261 14  
2262 15 </RequirementType>
```

10.2 Properties

2264 The *RequirementType* element has the following properties:

- 2265 • **name**: This attribute specifies the name or identifier of the Requirement Type, which **MUST** be
2266 unique within the target namespace.
- 2267 • **targetNamespace**: This **OPTIONAL** attribute specifies the target namespace to which the
2268 definition of the Requirement Type will be added. If not specified, the Requirement Type definition
2269 will be added to the target namespace of the enclosing Definitions document.
- 2270 • **abstract**: This **OPTIONAL** attribute specifies that no instances can be created from Node
2271 Templates of a Node Type that defines a requirement of this Requirement Type.

2272
2273 As a consequence, a Node Type with a Requirement Definition of an abstract Requirement Type
2274 **MUST** be declared as abstract as well and a derived Node Type that defines a requirement of a
2275 type derived from the abstract Requirement Type has to be defined. For example, an abstract
2276 Node Type “Application” might be defined having a requirement of the abstract type “Container”.
2277 A derived Node Type “Web Application” can then be defined with a more concrete requirement of
2278 type “Web Application Container” which can then be used for defining Node Templates that can

- 2279 be instantiated during the creation of a service according to a Service Template.
 2280
 2281 Note: an abstract Requirement Type MUST NOT be declared as final.
- 2282 • `final`: This OPTIONAL attribute specifies that other Requirement Types MUST NOT be derived
 2283 from this Requirement Type.
 2284
 2285 Note: a final Requirement Type MUST NOT be declared as abstract.
 - 2286 • `requiredCapabilityType`; This OPTIONAL attribute specifies the type of capability
 2287 needed to match the defined Requirement Type. The QName value of this attribute refers to the
 2288 QName of a `CapabilityType` element defined in the same Definitions document or in a
 2289 separate, imported document.
 2290
 2291 Note: The following basic match-making for Requirements and Capabilities MUST be supported
 2292 by each TOSCA implementation. Each Requirement is defined by a Requirement Definition,
 2293 which in turn refers to a Requirement Type that specifies the needed Capability Type by means of
 2294 its `requiredCapabilityType` attribute. The value of this attribute is used for basic type-
 2295 based match-making: a Capability matches a Requirement if the Requirement's Requirement
 2296 Type has a `requiredCapabilityType` value that corresponds to the Capability Type of the
 2297 Capability or one of its super-types.
 2298 Any domain-specific match-making semantics (e.g. based on constraints or properties) has to be
 2299 defined in the cause of specifying the corresponding Requirement Types and Capability Types.
 - 2300 • `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by
 2301 the author to describe the Requirement Type. Each tag is defined by a separate, nested `Tag`
 2302 element.
 2303 The `Tag` element has the following properties:
 - 2304 ○ `name`: This attribute specifies the name of the tag.
 - 2305 ○ `value`: This attribute specifies the value of the tag.
 2306
 2307 Note: The name/value pairs defined in tags have no normative interpretation.
 - 2308 • `DerivedFrom`: This is an OPTIONAL reference to another Requirement Type from which this
 2309 Requirement Type derives. See section 10.3 Derivation Rules for details.
 2310 The `DerivedFrom` element has the following properties:
 - 2311 ○ `typeRef`: The QName specifies the Requirement Type from which this Requirement
 2312 Type derives its definitions and semantics.
 - 2313 • `PropertiesDefinition`: This element specifies the structure of the observable properties
 2314 of the Requirement Type, such as its configuration and state, by means of XML schema.
 2315 The `PropertiesDefinition` element has one but not both of the following properties:
 - 2316 ○ `element`: This attribute provides the QName of an XML element defining the structure
 2317 of the Requirement Type Properties.
 - 2318 ○ `type`: This attribute provides the QName of an XML (complex) type defining the
 2319 structure of the Requirement Type Properties.

2320 10.3 Derivation Rules

2321 The following rules on combining definitions based on `DerivedFrom` apply:

- 2322 • Requirement Type Properties: It is assumed that the XML element (or type) representing the
 2323 Requirement Type Properties extends the XML element (or type) of the Requirement Type
 2324 Properties of the Requirement Type referenced in the `DerivedFrom` element.

2325 10.4 Example

2326 The following example defines the Requirement Type “DatabaseClientEndpoint” that expresses the
2327 requirement of a client for a database connection. It is defined in a Definitions document
2328 “MyRequirements” within the target namespace “http://www.example.com/SampleRequirements”. Thus,
2329 by importing the corresponding namespace into another Definitions document, the
2330 “DatabaseClientEndpoint” Requirement Type is available for use in the other document.

```
2331 01 <Definitions id="MyRequirements" name="My Requirements"  
2332 02   targetNamespace="http://www.example.com/SampleRequirements"  
2333 03   xmlns:br="http://www.example.com/BaseRequirementTypes"  
2334 04   xmlns:mrp="http://www.example.com/SampleRequirementProperties">  
2335 05  
2336 06   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"  
2337 07     namespace="http://www.example.com/BaseRequirementTypes"/>  
2338 08  
2339 09   <Import importType="http://www.w3.org/2001/XMLSchema"  
2340 10     namespace="http://www.example.com/SampleRequirementProperties"/>  
2341 11  
2342 12   <RequirementType name="DatabaseClientEndpoint">  
2343 13     <DerivedFrom typeRef="br:ClientEndpoint"/>  
2344 14     <PropertiesDefinition  
2345 15       element="mrp:DatabaseClientEndpointProperties"/>  
2346 16   </RequirementType>  
2347 17  
2348 18 </Definitions>
```

2349 The Requirement Type “DatabaseClientEndpoint” defined in the example above is derived from another
2350 generic “ClientEndpoint” Requirement Type defined in a separate file by means of the `DerivedFrom`
2351 element. The definitions in that separate Definitions file are imported by means of the first `Import`
2352 element and the namespace of those imported definitions is assigned the prefix “br” in the current file.

2353 The “DatabaseClientEndpoint” Requirement Type defines a set of properties through an XML schema
2354 element definition “DatabaseClientEndpointProperties”. For example, those properties might include the
2355 definition of a port number to be used for client connections. The XML schema definition is stored in a
2356 separate XSD file that is imported by means of the second `Import` element. The namespace of the XML
2357 schema definitions is assigned the prefix “mrp” in the current file.

2358

11 Capability Types

2359 This chapter specifies how *Capability Types* are defined. A Capability Type is a reusable entity that
2360 describes a kind of capability that a Node Type can declare to expose. For example, a Capability Type for
2361 a database server endpoint can be defined and various Node Types (e.g. a Node Type for a database)
2362 can declare to expose (or to “provide”) the capability of serving as a database server endpoint.

2363 A Capability Type defines the structure of observable properties via a *Properties Definition*, i.e. the
2364 names, data types and allowed values the properties defined in *Capabilities* of Node Templates of a Node
2365 Type can have in cases where the Node Type defines a capability of the respective Capability Type.

2366 A Capability Type can inherit properties and semantics from another Capability Type by means of the
2367 *DerivedFrom* element. Capability Types might be declared as abstract, meaning that they cannot be
2368 instantiated. The purpose of such abstract Capability Types is to provide common properties for re-use in
2369 specialized, derived Capability Types. Capability Types might also be declared as final, meaning that they
2370 cannot be derived by other Capability Types.

11.1 XML Syntax

2372 The following pseudo schema defines the XML syntax of Capability Types:

```
2373 01 <CapabilityType name="xs:NCName"  
2374 02           targetNamespace="xs:anyURI"?  
2375 03           abstract="yes|no"?  
2376 04           final="yes|no"?>  
2377 05  
2378 06   <Tags>  
2379 07     <Tag name="xs:string" value="xs:string"/> +  
2380 08   </Tags> ?  
2381 09  
2382 10   <DerivedFrom typeRef="xs:QName"/> ?  
2383 11  
2384 12   <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?  
2385 13  
2386 14 </CapabilityType>
```

11.2 Properties

2388 The *CapabilityType* element has the following properties:

- 2389 • **name**: This attribute specifies the name or identifier of the Capability Type, which **MUST** be
2390 unique within the target namespace.
- 2391 • **targetNamespace**: This **OPTIONAL** attribute specifies the target namespace to which the
2392 definition of the Capability Type will be added. If not specified, the Capability Type definition will
2393 be added to the target namespace of the enclosing Definitions document.
- 2394 • **abstract**: This **OPTIONAL** attribute specifies that no instances can be created from Node
2395 Templates of a Node Type that defines a capability of this Capability Type.

2396
2397 As a consequence, a Node Type with a Capability Definition of an abstract Capability Type **MUST**
2398 be declared as abstract as well and a derived Node Type that defines a capability of a type
2399 derived from the abstract Capability Type has to be defined. For example, an abstract Node Type
2400 “Server” might be defined having a capability of the abstract type “Container”. A derived Node
2401 Type “Web Server” can then be defined with a more concrete capability of type “Web Application
2402 Container” which can then be used for defining Node Templates that can be instantiated during
2403 the creation of a service according to a Service Template.

- 2404
2405 Note: an abstract Capability Type MUST NOT be declared as final.
- 2406 • `final`: This OPTIONAL attribute specifies that other Capability Types MUST NOT be derived
 - 2407 from this Capability Type.
 - 2408
 - 2409 Note: a final Capability Type MUST NOT be declared as abstract.
 - 2410 • `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by
 - 2411 the author to describe the Capability Type. Each tag is defined by a separate, nested `Tag`
 - 2412 element.
 - 2413 The `Tag` element has the following properties:
 - 2414 ◦ `name`: This attribute specifies the name of the tag.
 - 2415 ◦ `value`: This attribute specifies the value of the tag.
 - 2416
 - 2417 Note: The name/value pairs defined in tags have no normative interpretation.
 - 2418 • `DerivedFrom`: This is an OPTIONAL reference to another Capability Type from which this
 - 2419 Capability Type derives. See section 11.3 Derivation Rules for details.
 - 2420 The `DerivedFrom` element has the following properties:
 - 2421 ◦ `typeRef`: The QName specifies the Capability Type from which this Capability Type
 - 2422 derives its definitions and semantics.
 - 2423 • `PropertiesDefinition`: This element specifies the structure of the observable properties
 - 2424 of the Capability Type, such as its configuration and state, by means of XML schema.
 - 2425 The `PropertiesDefinition` element has one but not both of the following properties:
 - 2426 ◦ `element`: This attribute provides the QName of an XML element defining the structure
 - 2427 of the Capability Type Properties.
 - 2428 ◦ `type`: This attribute provides the QName of an XML (complex) type defining the
 - 2429 structure of the Capability Type Properties.

2430 11.3 Derivation Rules

2431 The following rules on combining definitions based on `DerivedFrom` apply:

- 2432 • `Capability Type Properties`: It is assumed that the XML element (or type) representing the
- 2433 Capability Type Properties extends the XML element (or type) of the Capability Type Properties
- 2434 of the Capability Type referenced in the `DerivedFrom` element.

2435 11.4 Example

2436 The following example defines the Capability Type “DatabaseServerEndpoint” that expresses the

2437 capability of a component to serve database connections. It is defined in a Definitions document

2438 “MyCapabilities” within the target namespace “http://www.example.com/SampleCapabilities”. Thus, by

2439 importing the corresponding namespace into another Definitions document, the

2440 “DatabaseServerEndpoint” Capability Type is available for use in the other document.

```

2441 01 <Definitions id="MyCapabilities" name="My Capabilities"
2442 02   targetNamespace="http://www.example.com/SampleCapabilities"
2443 03   xmlns:bc="http://www.example.com/BaseCapabilityTypes"
2444 04   xmlns:mcp="http://www.example.com/SampleCapabilityProperties">
2445 05
2446 06   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
2447 07           namespace="http://www.example.com/BaseCapabilityTypes"/>
2448 08
2449 09   <Import importType="http://www.w3.org/2001/XMLSchema"
2450 10           namespace="http://www.example.com/SampleCapabilityProperties"/>

```

```
2451 11
2452 12 <CapabilityType name="DatabaseServerEndpoint">
2453 13 <DerivedFrom typeRef="bc:ServerEndpoint"/>
2454 14 <PropertiesDefinition
2455 15 element="mcp:DatabaseServerEndpointProperties"/>
2456 16 </CapabilityType>
2457 17
2458 18 </Definitions>
```

2459 The Capability Type “DatabaseServerEndpoint” defined in the example above is derived from another
2460 generic “ServerEndpoint” Capability Type defined in a separate file by means of the `DerivedFrom`
2461 element. The definitions in that separate Definitions file are imported by means of the first `Import`
2462 element and the namespace of those imported definitions is assigned the prefix “bc” in the current file.

2463 The “DatabaseServerEndpoint” Capability Type defines a set of properties through an XML schema
2464 element definition “DatabaseServerEndpointProperties”. For example, those properties might include the
2465 definition of a port number where the server listens for client connections, or credentials to be used by
2466 clients. The XML schema definition is stored in a separate XSD file that is imported by means of the
2467 second `Import` element. The namespace of the XML schema definitions is assigned the prefix “mcp”
2468 in the current file.

2469

12 Artifact Types

2470 This chapter specifies how *Artifact Types* are defined. An Artifact Type is a reusable entity that defines
2471 the type of one or more Artifact Templates which in turn serve as deployment artifacts for Node
2472 Templates or implementation artifacts for Node Type and Relationship Type interface operations. For
2473 example, an Artifact Type “WAR File” might be defined for describing web application archive files. Based
2474 on this Artifact Type, one or more Artifact Templates representing concrete WAR files can be defined and
2475 referenced as deployment or implementation artifacts.

2476 An Artifact Type can define the structure of observable properties via a *Properties Definition*, i.e. the
2477 names, data types and allowed values the properties defined in Artifact Templates using an Artifact Type
2478 or instances of such Artifact Templates can have. Note that properties defined by an Artifact Type are
2479 assumed to be invariant across the contexts in which corresponding artifacts are used – as opposed to
2480 properties that can vary depending on the context. As an example of such an invariant property, an
2481 Artifact Type for a WAR file could define a “signature” property that can hold a hash for validating the
2482 actual artifact proper. In contrast, the path where the web application contained in the WAR file gets
2483 deployed can vary for each place where the WAR file is used.

2484 An Artifact Type can inherit definitions and semantics from another Artifact Type by means of the
2485 *DerivedFrom* element. Artifact Types can be declared as abstract, meaning that they cannot be
2486 instantiated. The purpose of such abstract Artifact Types is to provide common properties for re-use in
2487 specialized, derived Artifact Types. Artifact Types can also be declared as final, meaning that they cannot
2488 be derived by other Artifact Types.

2489

12.1 XML Syntax

2490 The following pseudo schema defines the XML syntax of Artifact Types:

```
2491 01 <ArtifactType name="xs:NCName"  
2492 02     targetNamespace="xs:anyURI"?  
2493 03     abstract="yes|no"?  
2494 04     final="yes|no"?>  
2495 05  
2496 06   <Tags>  
2497 07     <Tag name="xs:string" value="xs:string"/> +  
2498 08   </Tags> ?  
2499 09  
2500 10   <DerivedFrom typeRef="xs:QName"/> ?  
2501 11  
2502 12   <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?  
2503 13  
2504 14 </ArtifactType>
```

2505

12.2 Properties

2506 The *ArtifactType* element has the following properties:

- 2507 • **name**: This attribute specifies the name or identifier of the Artifact Type, which **MUST** be unique
2508 within the target namespace.
- 2509 • **targetNamespace**: This **OPTIONAL** attribute specifies the target namespace to which the
2510 definition of the Artifact Type will be added. If not specified, the Artifact Type definition will be
2511 added to the target namespace of the enclosing Definitions document.
- 2512 • **abstract**: This **OPTIONAL** attribute specifies that no instances can be created from Artifact
2513 Templates of that abstract Artifact Type, i.e. the respective artifacts cannot be used directly as
2514 deployment or implementation artifact in any context.
2515

2516 As a consequence, an Artifact Template of an abstract Artifact Type MUST be replaced by an
2517 artifact of a derived Artifact Type at the latest during deployment of the element that uses the
2518 artifact (i.e. a Node Template or Relationship Template).

2519
2520 Note: an abstract Artifact Type MUST NOT be declared as final.

- 2521 • `final`: This OPTIONAL attribute specifies that other Artifact Types MUST NOT be derived from
2522 this Artifact Type.

2523
2524 Note: a final Artifact Type MUST NOT be declared as abstract.

- 2525 • `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by
2526 the author to describe the Artifact Type. Each tag is defined by a separate, nested `Tag` element.
2527 The `Tag` element has the following properties:

- 2528 ○ `name`: This attribute specifies the name of the tag.
- 2529 ○ `value`: This attribute specifies the value of the tag.

2530
2531 Note: The name/value pairs defined in tags have no normative interpretation.

- 2532 • `DerivedFrom`: This is an OPTIONAL reference to another Artifact Type from which this Artifact
2533 Type derives. See section 12.3 Derivation Rules for details.

2534 The `DerivedFrom` element has the following properties:

- 2535 ○ `typeRef`: The QName specifies the Artifact Type from which this Artifact Type derives
2536 its definitions and semantics.

- 2537 • `PropertiesDefinition`: This element specifies the structure of the observable properties
2538 of the Artifact Type, such as its configuration and state, by means of XML schema.

2539 The `PropertiesDefinition` element has one but not both of the following properties:

- 2540 ○ `element`: This attribute provides the QName of an XML element defining the structure
2541 of the Artifact Type Properties.
- 2542 ○ `type`: This attribute provides the QName of an XML (complex) type defining the
2543 structure of the Artifact Type Properties.

2544 12.3 Derivation Rules

2545 The following rules on combining definitions based on `DerivedFrom` apply:

- 2546 • `Artifact Type Properties`: It is assumed that the XML element (or type) representing the Artifact
2547 Type Properties extends the XML element (or type) of the Artifact Type Properties of the Artifact
2548 Type referenced in the `DerivedFrom` element.

2549 12.4 Example

2550 The following example defines the Artifact Type “RPMPackage” that can be used for describing RPM
2551 packages as deployable artifacts on various Linux distributions. It is defined in a Definitions document
2552 “MyArtifacts” within the target namespace “<http://www.example.com/SampleArtifacts>”. Thus, by importing
2553 the corresponding namespace into another Definitions document, the “RPMPackage” Artifact Type is
2554 available for use in the other document.

```
2555 01 <Definitions id="MyArtifacts" name="My Artifacts"  
2556 02   targetNamespace="http://www.example.com/SampleArtifacts"  
2557 03   xmlns:ba="http://www.example.com/BaseArtifactTypes"  
2558 04   xmlns:map="http://www.example.com/SampleArtifactProperties">  
2559 05  
2560 06   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"  
2561 07     namespace="http://www.example.com/BaseArtifactTypes"/>  
2562 08
```

```
2563 09 <Import importType="http://www.w3.org/2001/XMLSchema"
2564 10     namespace="http://www.example.com/SampleArtifactProperties"/>
2565 11
2566 12 <ArtifactType name="RPMPackage">
2567 13     <DerivedFrom typeRef="ba:OSPackage"/>
2568 14     <PropertiesDefinition element="map:RPMPackageProperties"/>
2569 15 </ArtifactType>
2570 16
2571 17 </Definitions>
```

2572 The Artifact Type “RPMPackage” defined in the example above is derived from another generic
2573 “OSPackage” Artifact Type defined in a separate file by means of the `DerivedFrom` element. The
2574 definitions in that separate Definitions file are imported by means of the first `Import` element and the
2575 namespace of those imported definitions is assigned the prefix “ba” in the current file.

2576 The “RPMPackage” Artifact Type defines a set of properties through an XML schema element definition
2577 “RPMPackageProperties”. For example, those properties might include the definition of the name or
2578 names of one or more RPM packages. The XML schema definition is stored in a separate XSD file that is
2579 imported by means of the second `Import` element. The namespace of the XML schema definitions is
2580 assigned the prefix “map” in the current file.

2581

13 Artifact Templates

2582 This chapter specifies how *Artifact Templates* are defined. An Artifact Template represents an artifact that
2583 can be referenced from other objects in a Service Template as a deployment artifact or implementation
2584 artifact. For example, from Node Types or Node Templates, an Artifact Template for some software
2585 installable could be referenced as a deployment artifact for materializing a specific software component.
2586 As another example, from within interface definitions of Node Types or Relationship Types, an Artifact
2587 Template for a WAR file could be referenced as implementation artifact for a REST operation.

2588 An Artifact Template refers to a specific Artifact Type that defines the structure of observable properties
2589 (metadata) or the artifact. The Artifact Template then typically defines values for those properties inside
2590 the `Properties` element. Note that properties defined by an Artifact Type are assumed to be invariant
2591 across the contexts in which corresponding artifacts are used – as opposed to properties that can vary
2592 depending on the context.

2593 Furthermore, an Artifact Template typically provides one or more references to the actual artifact itself
2594 that can be contained as a file in the CSAR (see section 3.7 and section 14) containing the overall
2595 Service Template or that can be available at a remote location such as an FTP server.

2596 13.1 XML Syntax

2597 The following pseudo schema defines the XML syntax of Artifact Templates:

```
2598 01 <ArtifactTemplate id="xs:ID" name="xs:string"? type="xs:QName">  
2599 02  
2600 03   <Properties>  
2601 04     XML fragment  
2602 05   </Properties> ?  
2603 06  
2604 07   <PropertyConstraints>  
2605 08     <PropertyConstraint property="xs:string"  
2606 09       constraintType="xs:anyURI"> +  
2607 10       constraint ?  
2608 11     </PropertyConstraint>  
2609 12   </PropertyConstraints> ?  
2610 13  
2611 14   <ArtifactReferences>  
2612 15     <ArtifactReference reference="xs:anyURI">  
2613 16       (  
2614 17         <Include pattern="xs:string"/>  
2615 18         |  
2616 19         <Exclude pattern="xs:string"/>  
2617 20       ) *  
2618 21     </ArtifactReference> +  
2619 22   </ArtifactReferences> ?  
2620 23  
2621 24 </ArtifactTemplate>
```

2622 13.2 Properties

2623 The `ArtifactTemplate` element has the following properties:

- 2624 • `id`: This attribute specifies the identifier of the Artifact Template. The identifier of the Artifact
2625 Template MUST be unique within the target namespace.
- 2626 • `name`: This OPTIONAL attribute specifies the name of the Artifact Template.

- 2627
- 2628
- 2629
- 2630
- 2631
- 2632
- 2633
- 2634
- `type`: The QName value of this attribute refers to the Artifact Type providing the type of the Artifact Template.
- Note: If the Artifact Type referenced by the `type` attribute of an Artifact Template is declared as abstract, no instances of the specific Artifact Template can be created, i.e. the artifact cannot be used directly as deployment or implementation artifact. Instead, a substitution of the Artifact Template with one having a specialized, derived Artifact Type has to be done at the latest during the instantiation time of a Service Template.
- 2635
- 2636
- 2637
- 2638
- 2639
- 2640
- 2641
- 2642
- `Properties`: This OPTIONAL element specifies the invariant properties of the Artifact Template, i.e. those properties that will be commonly used across different contexts in which the Artifact Template is used.
- The initial values are specified by providing an instance document of the XML schema of the corresponding Artifact Type Properties. This instance document considers the inheritance structure deduced by the `DerivedFrom` property of the Artifact Type referenced by the `type` attribute of the Artifact Template.
- 2643
- 2644
- 2645
- 2646
- 2647
- `PropertyConstraints`: This OPTIONAL element specifies constraints on the use of one or more of the Artifact Type Properties of the Artifact Type providing the property definitions for the Artifact Template. Each constraint is specified by means of a separate nested `PropertyConstraint` element.
- The `PropertyConstraint` element has the following properties:
- `property`: The string value of this property is an XPath expression pointing to the property within the Artifact Type Properties document that is constrained within the context of the Artifact Template. More than one constraint MUST NOT be defined for each property.
 - `constraintType`: The constraint type is specified by means of a URI, which defines both the semantic meaning of the constraint as well as the format of the content.
- For example, a constraint type of `http://www.example.com/PropertyConstraints/unique` could denote that the reference property of the Artifact Template under definition has to be unique within a certain scope. The constraint type specific content of the respective `PropertyConstraint` element could then define the actual scope in which uniqueness has to be ensured in more detail.
- 2652
- 2653
- 2654
- 2655
- 2656
- 2657
- 2658
- 2659
- `ArtifactReferences`: This OPTIONAL element contains one or more references to the actual artifact proper, each represented by a separate `ArtifactReference` element.
- The `ArtifactReference` element has the following properties:
- `reference`: This attribute contains a URI pointing to an actual artifact. If this URI is a relative URI, it is interpreted relative to the root directory of the CSAR containing the Service Template (see also sections 3.7 and 14).
 - `Include`: This OPTIONAL element can be used to define a pattern of files that are to be included in the artifact reference in case the reference points to a complete directory. The `Include` element has the following properties:
 - `pattern`: This attribute contains a pattern definition for files that are to be included in the overall artifact reference. For example, a pattern of `"*.py"` would include all python scripts contained in a directory.
 - `Exclude`: This OPTIONAL element can be used to define a pattern of files that are to be excluded from the artifact reference in case the reference points to a complete directory. The `Exclude` element has the following properties:

- 2676
- 2677
- 2678
- `pattern`: This attribute contains a pattern definition for files that are to be excluded in the overall artifact reference. For example, a pattern of `"* . sh"` would exclude all bash scripts contained in a directory.

2679 13.3 Example

2680 The following example defines the Artifact Template "MyInstallable" that points to a zip file containing
2681 some software installable. It is defined in a Definitions document "MyArtifacts" within the target
2682 namespace "http://www.example.com/SampleArtifacts". The Artifact Template can be used in the same
2683 document, for example as a deployment artifact for some Node Template representing a software
2684 component, or it can be used in other Definitions documents by importing the corresponding namespace
2685 into another document.

```
2686 01 <Definitions id="MyArtifacts" name="My Artifacts"  
2687 02   targetNamespace="http://www.example.com/SampleArtifacts"  
2688 03   xmlns:ba="http://www.example.com/BaseArtifactTypes">  
2689 04  
2690 05   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"  
2691 06     namespace="http://www.example.com/BaseArtifactTypes"/>  
2692 07  
2693 08   <ArtifactTemplate id="MyInstallable"  
2694 09     name="My installable"  
2695 10     type="ba:ZipFile">  
2696 11     <ArtifactReferences>  
2697 12       <ArtifactReference reference="files/MyInstallable.zip"/>  
2698 13     </ArtifactReferences>  
2699 14   </ArtifactTemplate>  
2700 15  
2701 16 </Definitions>
```

2702 The Artifact Template "MyInstallable" defined in the example above is of type "ZipFile" that is specified in
2703 the `type` attribute of the `ArtifactTemplate` element. This Artifact Type is defined in a separate file,
2704 the definitions of which are imported by means of the `Import` element and the namespace of those
2705 imported definitions is assigned the prefix "ba" in the current file.

2706 The "MyInstallable" Artifact Template provides a reference to a file "MyInstallable.zip" by means of the
2707 `ArtifactReference` element. Since the URI provided in the `reference` attribute is a relative URI,
2708 it is interpreted relative to the root directory of the CSAR containing the Service Template.

2709

14 Policy Types

2710 This chapter specifies how *Policy Types* are defined. A Policy Type is a reusable entity that describes a
2711 kind of non-functional behavior or a kind of quality-of-service (QoS) that a Node Type can declare to
2712 expose. For example, a Policy Type can be defined to express high availability for specific Node Types
2713 (e.g. a Node Type for an application server).

2714 A Policy Type defines the structure of observable properties via a Properties Definition, i.e. the names,
2715 data types and allowed values the properties defined in a corresponding Policy Template can have.

2716 A Policy Type can inherit properties from another Policy Type by means of the `DerivedFrom` element.

2717 A Policy Type declares the set of Node Types it specifies non-functional behavior for via the `AppliesTo`
2718 element. Note that being “applicable to” does not enforce implementation: i.e. in case a Policy Type
2719 expressing high availability is associated with a “Webserver” Node Type, an instance of the Webserver is
2720 not necessarily highly available. Whether or not an instance of a Node Type to which a Policy Type is
2721 applicable will show the specified non-functional behavior, is determined by a Node Template of the
2722 corresponding Node Type.

2723

14.1 XML Syntax

2724 The following pseudo schema defines the XML syntax of Policy Types:

```
2725 01 <PolicyType name="xs:NCName"  
2726 02     policyLanguage="xs:anyURI"?  
2727 03     abstract="yes|no"?  
2728 04     final="yes|no"?  
2729 05     targetNamespace="xs:anyURI"?>  
2730 06   <Tags>  
2731 07     <Tag name="xs:string" value="xs:string"/> +  
2732 08   </Tags> ?  
2733 09  
2734 10   <DerivedFrom typeRef="xs:QName"/> ?  
2735 11  
2736 12   <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?  
2737 13  
2738 14   <AppliesTo>  
2739 15     <NodeTypeReference typeRef="xs:QName"/> +  
2740 16   </AppliesTo> ?  
2741 17  
2742 18   policy type specific content ?  
2743 19  
2744 20 </PolicyType>
```

2745

14.2 Properties

2746 The `PolicyType` element has the following properties:

- 2747 • `name`: This attribute specifies the name or identifier of the Policy Type, which **MUST** be unique
2748 within the target namespace.
- 2749 • `targetNamespace`: This **OPTIONAL** attribute specifies the target namespace to which the
2750 definition of the Policy Type will be added. If not specified, the Policy Type definition will be added
2751 to the target namespace of the enclosing Definitions document.
- 2752 • `policyLanguage`: This **OPTIONAL** attribute specifies the language used to specify the details
2753 of the Policy Type. These details can be defined as policy type specific content of the `PolicyType`
2754 element.

2755 • **abstract**: This OPTIONAL attribute specifies that no instances can be created from Policy
2756 Templates of that abstract Policy Type, i.e. the respective policies cannot be used directly during
2757 the instantiation of a Service Template.
2758

2759 As a consequence, a Policy Template of an abstract Policy Type MUST be replaced by a policy
2760 of a derived Policy Type at the latest during deployment of the element that policy is attached to.

2761 • **final**: This OPTIONAL attribute specifies that other Policy Types MUST NOT be derived from
2762 this Policy Type.
2763

2764 Note: a final Policy Type MUST NOT be declared as abstract.

2765 • **Tags**: This OPTIONAL element allows the definition of any number of tags which can be used by
2766 the author to describe the Policy Type. Each tag is defined by a separate, nested `Tag` element.
2767 The `Tag` element has the following properties:

2768 ○ **name**: This attribute specifies the name of the tag.

2769 ○ **value**: This attribute specifies the value of the tag.
2770

2771 Note: The name/value pairs defined in tags have no normative interpretation.

2772 • **DerivedFrom**: This is an OPTIONAL reference to another Policy Type from which this Policy
2773 Type derives. See section 14.3 Derivation Rules for details.
2774 The `DerivedFrom` element has the following properties:

2775 ○ **typeRef**: The QName specifies the Policy Type from which this Policy Type derives its
2776 definitions from.

2777 • **PropertiesDefinition**: This element specifies the structure of the observable properties
2778 of the Policy Type by means of XML schema.
2779 The `PropertiesDefinition` element has one but not both of the following properties:

2780 ○ **element**: This attribute provides the QName of an XML element defining the structure
2781 of the Policy Type Properties.

2782 ○ **type**: This attribute provides the QName of an XML (complex) type defining the
2783 structure of the Policy Type Properties.

2784 • **AppliesTo**: This OPTIONAL element specifies the set of Node Types the Policy Type is
2785 applicable to, each defined as a separate, nested `NodeTypeReference` element.
2786 The `NodeTypeReference` element has the following property:

2787 ○ **typeRef**: The attribute provides the QName of a Node Type to which the Policy Type
2788 applies.

2789 14.3 Derivation Rules

2790 The following rules on combining definitions based on `DerivedFrom` apply:

2791 • **Properties Definitions**: It is assumed that the XML element (or type) representing the Policy Type
2792 Properties Definitions extends the XML element (or type) of the Policy Type Properties Definitions
2793 of the Policy Type referenced in the `DerivedFrom` element.

2794 • **Applies To**: The set of Node Types the Policy Type is applicable to consist of the set union of
2795 Node Types derived from and Node Types explicitly referenced by the Policy Type by means of
2796 its `AppliesTo` element.

2797 • **Policy Language**: A Policy Type MUST define the same policy language as the Policy Type it
2798 derives from. In case the Policy Type used as basis for derivation has no `policyLanguage`
2799 attribute defined, the deriving Policy Type can define any appropriate policy language.

2800 14.4 Example

2801 The following example defines two Policy Types, the “HighAvailability” Policy Type and the
2802 “ContinuousAvailability” Policy Type. They are defined in a Definitions document “MyPolicyTypes” within
2803 the target namespace “http://www.example.com/SamplePolicyTypes”. Thus, by importing the
2804 corresponding namespace into another Definitions document, both Policy Types are available for use in
2805 the other document.

```
2806 01 <Definitions id="MyPolicyTypes" name="My Policy Types"  
2807 02   targetNamespace="http://www.example.com/SamplePolicyTypes"  
2808 03   xmlns:bnt="http://www.example.com/BaseNodeTypes">  
2809 04   xmlns:spp="http://www.example.com/SamplePolicyProperties">  
2810 05  
2811 06   <Import importType="http://www.w3.org/2001/XMLSchema"  
2812 07     namespace="http://www.example.com/SamplePolicyProperties"/>  
2813 08  
2814 09   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"  
2815 10     namespace="http://www.example.com/BaseNodeTypes"/>  
2816 11  
2817 12  
2818 13   <PolicyType name="HighAvailability">  
2819 14     <PropertiesDefinition element="spp:HAProperties"/>  
2820 15   </PolicyType>  
2821 16  
2822 17   <PolicyType name="ContinuousAvailability">  
2823 18     <DerivedFrom typeRef="HighAvailability"/>  
2824 19     <PropertiesDefinition element="spp:CAProperties"/>  
2825 20     <AppliesTo>  
2826 21       <NodeTypeReference typeRef="bnt:DBMS"/>  
2827 22     </AppliesTo>  
2828 23   </PolicyType>  
2829 24  
2830 25 </Definitions>
```

2831 The Policy Type “HighAvailability” defined in the example above has the “HAProperties” properties that
2832 are defined in a separate namespace as an XML element. The same namespace contains the
2833 “CAProperties” element that defines the properties of the “ContinuousAvailability” Policy Type. This
2834 namespace is imported by means of the first `Import` element and the namespace of those imported
2835 definitions is assigned the prefix “spp” in the current file.

2836 The “ContinuousAvailability” Policy Type is derived from the “HighAvailability” Policy Type. Furthermore, it
2837 is applicable to the “DBMS” Node Type. This Node Type is defined in a separate namespace, which is
2838 imported by means of the second `Import` element and the namespace of those imported definitions is
2839 assigned the prefix “bnt” in the current file.

2840

15 Policy Templates

2841 This chapter specifies how *Policy Templates* are defined. A Policy Template represents a particular non-
2842 functional behavior or quality-of-service that can be referenced by a Node Template. A Policy Template
2843 refers to a specific Policy Type that defines the structure of observable properties (metadata) of the non-
2844 functional behavior. The Policy Template then typically defines values for those properties inside the
2845 *Properties* element. Note that properties defined by a Policy Template are assumed to be invariant
2846 across the contexts in which corresponding behavior is exposed – as opposed to properties defined in
2847 Policies of Node Templates that may vary depending on the context.

2848

15.1 XML Syntax

2849 The following pseudo schema defines the XML syntax of Policy Templates:

```
2850 01 <PolicyTemplate id="xs:ID" name="xs:string"? type="xs:QName">  
2851 02  
2852 03   <Properties>  
2853 04     XML fragment  
2854 05   </Properties> ?  
2855 06  
2856 07   <PropertyConstraints>  
2857 08     <PropertyConstraint property="xs:string"  
2858 09       constraintType="xs:anyURI"> +  
2859 10       constraint ?  
2860 11     </PropertyConstraint>  
2861 12   </PropertyConstraints> ?  
2862 13  
2863 14   policy type specific content ?  
2864 15  
2865 16 </PolicyTemplate>
```

2866

15.2 Properties

2867 The `PolicyTemplate` element has the following properties:

- 2868 • `id`: This attribute specifies the identifier of the Policy Template which **MUST** be unique within the
2869 target namespace.
- 2870 • `name`: This **OPTIONAL** attribute specifies the name of the Policy Template.
- 2871 • `type`: The QName value of this attribute refers to the Policy Type providing the type of the Policy
2872 Template.
- 2873 • `Properties`: This **OPTIONAL** element specifies the invariant properties of the Policy
2874 Template, i.e. those properties that will be commonly used across different contexts in which the
2875 Policy Template is used.

2876

2877 The initial values are specified by providing an instance document of the XML schema of the
2878 corresponding Policy Type Properties. This instance document considers the inheritance
2879 structure deduced by the `DerivedFrom` property of the Policy Type referenced by the `type`
2880 attribute of the Policy Template.

- 2881 • `PropertyConstraints`: This **OPTIONAL** element specifies constraints on the use of one or
2882 more of the Policy Type Properties of the Policy Type providing the property definitions for the
2883 Policy Template. Each constraint is specified by means of a separate nested
2884 `PropertyConstraint` element.

2885 The `PropertyConstraint` element has the following properties:

- 2886
- 2887
- 2888
- 2889
- `property`: The string value of this property is an XPath expression pointing to the property within the Policy Type Properties document that is constrained within the context of the Policy Template. More than one constraint MUST NOT be defined for each property.
 - `constraintType`: The constraint type is specified by means of a URI, which defines both the semantic meaning of the constraint as well as the format of the content.
- 2890
- 2891

2892 15.3 Example

2893 The following example defines a Policy Template “MyHAPolicy”. It is defined in a Definitions document
2894 “MyPolicies” within the target namespace “http://www.example.com/SamplePolicies”. The Policy
2895 Template can be used in the same Definitions document, for example, as a Policy of some Node
2896 Template, or it can be used in other document by importing the corresponding namespace into the other
2897 document.

```
2898 01 <Definitions id="MyPolicies" name="My Policies"  
2899 02   targetNamespace="http://www.example.com/SamplePolicies"  
2900 03   xmlns:spt="http://www.example.com/SamplePolicyTypes">  
2901 04  
2902 05   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"  
2903 06     namespace="http://www.example.com/SamplePolicyTypes"/>  
2904 07  
2905 08   <PolicyTemplate id="MyHAPolicy"  
2906 09     name="My High Availability Policy"  
2907 10     type="bpt:HighAvailability">  
2908 11     <Properties>  
2909 12       <HAProperties>  
2910 13         <AvailabilityClass>4</AvailabilityClass>  
2911 14         <HeartbeatFrequency measuredIn="msec">  
2912 15           250  
2913 16         </HeartbeatFrequency>  
2914 17       </HAProperties>  
2915 18     </Properties>  
2916 19   </PolicyTemplate>  
2917 20  
2918 21 </Definitions>
```

2919 The Policy Template “MyHAPolicy” defined in the example above is of type “HighAvailability” that is
2920 specified in the `type` attribute of the `PolicyTemplate` element. This Policy Type is defined in a
2921 separate file, the definitions of which are imported by means of the `Import` element and the namespace
2922 of those imported definitions is assigned the prefix “spt” in the current file.

2923 The “MyHAPolicy” Policy Template provides values for the properties defined by the Properties Definition
2924 of the “HighAvailability” Policy Type. The `AvailabilityClass` property is set to “4”. The value of the
2925 `HeartbeatFrequency` is “250”, measured in “msec”.

2926

2927 16 Cloud Service Archive (CSAR)

2928 This section defines the metadata of a cloud service archive as well as its overall structure.

2929 16.1 Overall Structure of a CSAR

2930 A CSAR is a zip file containing at least two directories, the *TOSCA-Metadata* directory and the *Definitions*
2931 directory. Beyond that, other directories MAY be contained in a CSAR, i.e. the creator of a CSAR has all
2932 freedom to define the content of a CSAR and the structuring of this content as appropriate for the cloud
2933 application.

2934 The TOSCA-Metadata directory contains metadata describing the other content of the CSAR. This
2935 metadata is referred to as *TOSCA meta file*. This file is named `TOSCA` and has the file extension `.meta`.

2936 The Definitions directory contains one or more TOSCA Definitions documents (file extension `.tosca`).
2937 These Definitions files typically contain definitions related to the cloud application of the CSAR. In
2938 addition, CSARs can contain just the definition of elements for re-use in other contexts. For example, a
2939 CSAR might be used to package a set of Node Types and Relationship Types with their respective
2940 implementations that can then be used by Service Templates provided in other CSARs. In cases where a
2941 complete cloud application is packaged in a CSAR, one of the Definitions documents in the Definitions
2942 directory MUST contain a Service Template definition that defines the structure and behavior of the cloud
2943 application.

2944 16.2 TOSCA Meta File

2945 The TOSCA meta file includes metadata that allows interpreting the various artifacts within the CSAR
2946 properly. The `TOSCA.meta` file is contained in the `TOSCA-Metadata` directory of the CSAR.

2947 A TOSCA meta file consists of name/value pairs. The name-part of a name/value pair is followed by a
2948 colon, followed by a blank, followed by the value-part of the name/value pair. The name MUST NOT
2949 contain a colon. Values that represent binary data MUST be base64 encoded. Values that extend beyond
2950 one line can be spread over multiple lines if each subsequent line starts with at least one space. Such
2951 spaces are then collapsed when the value string is read.

```
2952 01 <name>: <value>
```

2953 Each name/value pair is in a separate line. A list of related name/value pairs, i.e. a list of consecutive
2954 name/value pairs describing a particular file in a CSAR, is called a *block*. Blocks are separated by an
2955 empty line. The first block, called *block_0*, is metadata about the CSAR itself. All other blocks represent
2956 metadata of files in the CSAR.

2957 The structure of `block_0` in the TOSCA meta file is as follows:

```
2958 01 TOSCA-Meta-File-Version: digit.digit  
2959 02 CSAR-Version: digit.digit  
2960 03 Created-By: string  
2961 04 Entry-Definitions: string ?
```

2962 The name/value pairs are as follows:

- 2963 • `TOSCA-Meta-File-Version`: This is the version number of the TOSCA meta file format.
2964 The value MUST be “1.0” in the current version of the TOSCA specification.
- 2965 • `CSAR-Version`: This is the version number of the CSAR specification. The value MUST be
2966 “1.0” in the current version of the TOSCA specification.
- 2967 • `Created-By`: The person or vendor, respectively, who created the CSAR.

- 2968
- `Entry-Definitions`: This OPTIONAL name/value pair references a TOSCA Definitions file from the Definitions directory of the CSAR that SHOULD be used as entry point for processing the contents of the CSAR.
- 2971 Note, that a CSAR may contain multiple Definitions files. One reason for this is completeness, e.g. a Service Template defined in one of the Definitions files could refer to Node Types defined in another Definitions file that might be included in the Definitions directory to avoid importing it from external locations. The `Entry-Definitions` name/value pair is a hint to allow optimized processing of the set of files in the Definitions directory.

2977 The first line of a block (other than `block_0`) MUST be a name/value pair that has the name “Name” and the value of which is the path-name of the file described. The second line MUST be a name/value pair that has the name “Content-Type” describing the type of the file described; the format is that of a MIME type with type/subtype structure. The other name/value pairs that consecutively follow are file-type specific.

```
2982 01 Name: <path-name_1>
2983 02 Content-Type: type_1/subtype_1
2984 03 <name_11>: <value_11>
2985 04 <name_12>: <value_12>
2986 05 ...
2987 06 <name_1n>: <value_1n>
2988 07
2989 08 ...
2990 09
2991 10 Name: <path-name_k>
2992 11 Content-Type: type_k/subtype_k
2993 12 <name_k1>: <value_k1>
2994 13 <name_k2>: <value_k2>
2995 14 ...
2996 15 <name_km>: <value_km>
```

2997 The name/value pairs are as follows:

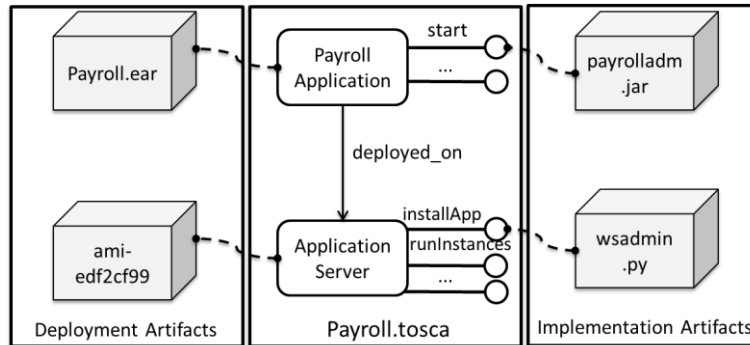
- `Name`: The pathname or pathname pattern of the file(s) or resources described within the actual CSAR.
- 3000 Note, that the file located at this location MAY basically contain a reference to an external file. Such a reference is given by a URI that is of one of the URL schemes “file”, “http”, or “https”.
- `Content-Type`: The type of the file described. This type is a MIME type complying with the type/subtype structure. Vendor defined subtypes SHOULD start as usual with the string “vnd.”.

3005 Note that later directives override earlier directives. This allows for specifying global default directives that can be specialized by later directorives in the TOSCA meta file.

3007 16.3 Example

3008 Figure 7 depicts a sample Definitions file named `Payroll.tosca` containing a Service Template of an application. The application is a payroll application written in Java that MUST be deployed on a proper application server. The Service Template of the application defines the Node Template `Payroll Application`, the Node Template `Application Server`, as well as the Relationship Template `deployed_on`. The `Payroll Application` is associated with an EAR file (named `Payroll.ear`) which is provided as corresponding Deployment Artifact of the `Payroll Application` Node Template. An Amazon Machine Image (AMI) is the Deployment Artifact of the `Application Server` Node Template; this Deployment Artifact is a reference to the image in the Amazon EC2 environment. The Implementation Artifacts of some operations of the Node Templates are

3017 provided too; for example, the start operation of the Payroll Application is implemented by a
 3018 Java API supported by the payrolladm.jar file, the installApp operation of the Application
 3019 Server is realized by the Python script wsadmin.py, while the runInstances operation is a REST
 3020 API available at Amazon for running instances of an AMI. Note, that the runInstances operation is
 3021 not related to a particular implementation artifact because it is available as an Amazon Web Service
 3022 (<https://ec2.amazonaws.com/?Action=RunInstances>); but the details of this REST API are specified with
 3023 the operation of the Application Server Node Type.



3024
 3025 Figure 7: Sample Service Template

3026 The corresponding Node Types and Relationship Types have been defined in the
 3027 PayrollTypes.tosca document, which is imported by the Definitions document containing the
 3028 Payroll Service Template. The following listing provides some of the details:

```

3029 01 <Definitions id="PayrollDefinitions"
3030 02     targetNamespace="http://www.example.com/ste"
3031 03     xmlns:pay="http://www.example.com/ste/Types">
3032 04
3033 05     <Import namespace="http://www.example.com/ste/Types"
3034 06         location="http://www.example.com/ste/Types/PayrollTypes.tosca"
3035 07         importType=" http://docs.oasis-open.org/tosca/ns/2011/12"/>
3036 08
3037 09     <Types>
3038 10         ...
3039 11     </Types>
3040 12
3041 13     <ServiceTemplate id="Payroll" name="Payroll Service Template">
3042 14
3043 15         <TopologyTemplate ID="PayrollTemplate">
3044 16
3045 17             <NodeTemplate id="Payroll Application"
3046 18                 type="pay:ApplicationNodeType">
3047 19                 ...
3048 20
3049 21                 <DeploymentArtifacts>
3050 22                     <DeploymentArtifact name="PayrollEAR"
3051 23                         type="http://www.example.com/
3052 24                             ns/tosca/2011/12/
3053 25                             DeploymentArtifactTypes/CSARref">
3054 26                         EARs/Payroll.ear
3055 27                     </DeploymentArtifact>
3056 28                 </DeploymentArtifacts>
3057 29             </NodeTemplate>
3058 30
3059 31             <NodeTemplate id="Application Server"
3060 32                 type="pay:ApplicationServerNodeType">
3061 33
  
```

```

3062 34      ...
3063 35
3064 36      <DeploymentArtifacts>
3065 37          <DeploymentArtifact name="ApplicationServerImage"
3066 38              type="http://www.example.com/
3067 39                  ns/tosca/2011/12/
3068 40                      DeploymentArtifactTypes/AMIref">
3069 41              ami-edf2cf99
3070 42          </DeploymentArtifact>
3071 43      </DeploymentArtifacts>
3072 44
3073 45  </NodeTemplate>
3074 46
3075 47  <RelationshipTemplate id="deployed_on"
3076 48              type="pay:deployed_on">
3077 49      <SourceElement ref="Payroll Application"/>
3078 50      <TargetElement ref="Application Server"/>
3079 51  </RelationshipTemplate>
3080 52
3081 53  </TopologyTemplate>
3082 54
3083 55 </ServiceTemplate>
3084 56
3085 57 </Definitions>

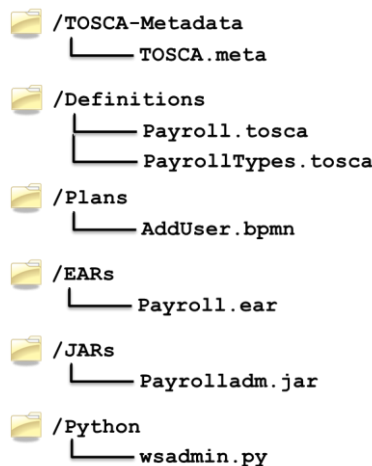
```

3086

3087 The Payroll Application Node Template specifies the deployment artifact PayrollEAR. It is a
3088 reference to the CSAR containing the Payroll.ste file, which is indicated by the .../CSARref type
3089 of the DeploymentArtifact element. The type specific content is a path expression in the directory
3090 structure of the CSAR: it points to the Payroll.ear file in the EARs directory of the CSAR (see Figure
3091 8 for the structure of the corresponding CSAR).

3092 The Application Server Node Template has a DeploymentArtifact called
3093 ApplicationServerImage that is a reference to an AMI (Amazon Machine Image), indicated by an
3094 .../AMIref type.

3095 The corresponding CSAR has the following structure (see Figure 8): The TOSCA.meta file is contained
3096 in the TOSCA-Metadata directory. The Payroll.ste file itself is contained in the Service-
3097 Template directory. Also, the PayrollTypes.ste file is in this directory. The content of the other
3098 directories has been sketched before.



3099

3100 Figure 8: Structure of CSAR Sample

3101 The TOSCA.meta file is as follows:

```
3102 01 TOSCA-Meta-Version: 1.0
3103 02 CSAR-Version: 1.0
3104 03 Created-By: Frank
3105 04
3106 05 Name: Service-Template/Payroll.tosca
3107 06 Content-Type: application/vnd.oasis.tosca.definitions
3108 07
3109 08 Name: Service-Template/PayrollTypes.ste
3110 09 Content-Type: application/vnd.oasis.tosca.definitions
3111 10
3112 11 Name: Plans/AddUser.bpmn
3113 12 Content-Type: application/vnd.oasis.bpmn
3114 13
3115 14 Name: EARs/Payroll.ear
3116 15 Content-Type: application/vnd.oasis.ear
3117 16
3118 17 Name: JARs/Payrolladm.jar
3119 18 Content-Type: application/vnd.oasis.jar
3120 19
3121 20 Name: Python/wsadmin.py
3122 21 Content-Type: application/vnd.oasis.py
```

3123

3124 **17 Security Considerations**

3125 TOSCA does not mandate the use of any specific mechanism or technology for client authentication.
3126 However, a client **MUST** provide a principal or the principal **MUST** be obtainable by the infrastructure.

3127 **18 Conformance**

3128 A TOSCA Definitions document conforms to this specification if it conforms to the TOSCA schema and
3129 follows the syntax and semantics defined in the normative portions of this specification. The TOSCA
3130 schema takes precedence over the TOSCA grammar (pseudo schema as defined in section 2.5), which
3131 in turn takes precedence over normative text, which in turn takes precedence over examples.

3132 An implementation conforms to this specification if it can process a conformant TOSCA Definitions
3133 document according to the rules described in chapters 4 through 16 of this specification.

3134 This specification allows extensions. Each implementation SHALL fully support all required functionality of
3135 the specification exactly as specified. The use of extensions SHALL NOT contradict nor cause the non-
3136 conformance of functionality defined in the specification.

3137

Appendix A. Portability and Interoperability Considerations

3138

3139 This section illustrates the portability and interoperability aspects addressed by Service Templates:

3140 Portability - The ability to take Service Templates created in one vendor's environment and use them in
3141 another vendor's environment.

3142 Interoperability - The capability for multiple components (e.g. a task of a plan and the definition of a
3143 topology node) to interact using well-defined messages and protocols. This enables combining
3144 components from different vendors allowing seamless management of services.

3145 Portability demands support of TOSCA elements.

3146

Appendix B. Acknowledgements

3147 The following individuals have participated in the creation of this specification and are gratefully
3148 acknowledged.

3149 **Participants:**

Aaron Zhang	Huawei Technologies Co., Ltd.
Adolf Hohl	NetApp
Afkham Azeez	WSO2
Al DeLucca	IBM
Alex Heneveld	Cloudsoft Corporation Limited
Allen Bannon	SAP AG
Anthony Rutkowski	Yaana Technologies, LLC
Arvind Srinivasan	IBM
Bryan Haynie	VCE
Bryan Murray	Hewlett-Packard
Chandrasekhar Sundaresh	CA Technologies
Charith Wickramarachchi	WSO2
Colin Hopkinson	3M HIS
Dale Moberg	Axway Software
Debojyoti Dutta	Cisco Systems
Dee Schur	OASIS
Denis Nothern	CenturyLink
Denis Weerasiri	WSO2
Derek Palma	Vnomic
Dhiraj Pathak	PricewaterhouseCoopers LLP:
Diane Mueller	ActiveState Software, Inc.
Doug Davis	IBM
Douglas Neuse	CA Technologies
Duncan Johnston-Watt	Cloudsoft Corporation Limited
Efraim Moscovich	CA Technologies
Frank Leymann	IBM
Gerd Breiter	IBM
James Thomason	Gale Technologies
Jan Ignatius	Nokia Siemens Networks GmbH & Co. KG
Jie Zhu	Huawei Technologies Co., Ltd.
John Wilmes	Individual
Joseph Malek	VCE
Ken Zink	CA Technologies
Kevin Poulter	SAP AG
Kevin Wilson	Hewlett-Packard
Koert Struijk	CA Technologies
Lee Thompson	Morphlabs, Inc.
li peng	Huawei Technologies Co., Ltd.
Marvin Waschke	CA Technologies
Mascot Yu	Huawei Technologies Co., Ltd.
Matthew Dovey	JISC Executive, University of Bristol
Matthew Rutkowski	IBM
Michael Schuster	SAP AG
Mike Edwards	IBM

Naveen Joy	Cisco Systems
Nikki Heron	rPath, Inc.
Paul Fremantle	WSO2
Paul Lipton	CA Technologies
Paul Zhang	Huawei Technologies Co., Ltd.
Rachid Sijelmassi	CA Technologies
Ravi Akireddy	Cisco Systems
Richard Bill	Jericho Systems
Richard Probst	SAP AG
Robert Evans	Zenoss, Inc.
Roland Wartenberg	Citrix Systems
Satoshi Konno	Morphlabs, Inc.
Sean Shen	China Internet Network Information Center(CNNIC)
Selvaratnam Uthaiyashankar	WSO2
Senaka Fernando	WSO2
Sherry Yu	Red Hat
Shumin Cheng	Huawei Technologies Co., Ltd.
Simon Moser	IBM
Srinath Perera	WSO2
Stephen Tyler	CA Technologies
Steve Fanshier	Software AG, Inc.
Steve Jones	Capgemini
Steve Winkler	SAP AG
Tad Deffler	CA Technologies
Ted Streete	VCE
Thilina Buddhika	WSO2
Thomas Spatzier	IBM
Tobias Kunze	Red Hat
Wang Xuan	Primeton Technologies, Inc.
wayne adams	EMC
Wenbo Zhu	Google Inc.
Xiaonan Song	Primeton Technologies, Inc.
YanJiong WANG	Primeton Technologies, Inc.
Zhexuan Song	Huawei Technologies Co., Ltd.

3150

Appendix C. Complete TOSCA Grammar

3152 **Note:** The following is a pseudo EBNF grammar notation meant for documentation purposes only. The
3153 grammar is not intended for machine processing.

```

3154 01 <Definitions id="xs:ID"
3155 02     name="xs:string"?
3156 03     targetNamespace="xs:anyURI">
3157 04
3158 05     <Extensions>
3159 06         <Extension namespace="xs:anyURI"
3160 07             mustUnderstand="yes|no"?/> +
3161 08     </Extensions> ?
3162 09
3163 10     <Import namespace="xs:anyURI"?
3164 11         location="xs:anyURI"?
3165 12         importType="xs:anyURI"/> *
3166 13
3167 14     <Types>
3168 15         <xs:schema .../> *
3169 16     </Types> ?
3170 17
3171 18     (
3172 19         <ServiceTemplate id="xs:ID"
3173 20             name="xs:string"?
3174 21             targetNamespace="xs:anyURI"
3175 22             substitutableNodeType="xs:QName"?>
3176 23
3177 24         <Tags>
3178 25             <Tag name="xs:string" value="xs:string"/> +
3179 26         </Tags> ?
3180 27
3181 28         <BoundaryDefinitions>
3182 29             <Properties>
3183 30                 XML fragment
3184 31                 <PropertyMappings>
3185 32                     <PropertyMapping serviceTemplatePropertyRef="xs:string"
3186 33                         targetObjectRef="xs:IDREF"
3187 34                         targetPropertyRef="xs:IDREF"/> +
3188 35                 </PropertyMappings/> ?
3189 36             </Properties> ?
3190 37
3191 38             <PropertyConstraints>
3192 39                 <PropertyConstraint property="xs:string"
3193 40                     constraintType="xs:anyURI"> +
3194 41                     constraint ?
3195 42                 </PropertyConstraint>
3196 43             </PropertyConstraints> ?
3197 44
3198 45             <Requirements>
3199 46                 <Requirement name="xs:string" ref="xs:IDREF"/> +
3200 47             </Requirements> ?
3201 48
3202 49             <Capabilities>
3203 50                 <Capability name="xs:string" ref="xs:IDREF"/> +
3204 51             </Capabilities> ?

```

```

3205 52
3206 53     <Policies>
3207 54         <Policy name="xs:string"? policyType="xs:QName"
3208 55             policyRef="xs:QName"?>
3209 56             policy specific content ?
3210 57         </Policy> +
3211 58     </Policies> ?
3212 59
3213 60     <Interfaces>
3214 61         <Interface name="xs:NCName">
3215 62             <Operation name="xs:NCName">
3216 63                 (
3217 64                     <NodeOperation nodeRef="xs:IDREF"
3218 65                         interfaceName="xs:anyURI"
3219 66                         operationName="xs:NCName"/>
3220 67                 |
3221 68                 <RelationshipOperation relationshipRef="xs:IDREF"
3222 69                     interfaceName="xs:anyURI"
3223 70                     operationName="xs:NCName"/>
3224 71                 |
3225 72                 <Plan planRef="xs:IDREF"/>
3226 73                 )
3227 74             </Operation> +
3228 75         </Interface> +
3229 76     </Interfaces> ?
3230 77
3231 78 </BoundaryDefinitions> ?
3232 79
3233 80 <TopologyTemplate>
3234 81     (
3235 82         <NodeTemplate id="xs:ID" name="xs:string"? type="xs:QName"
3236 83             minInstances="xs:integer"?
3237 84             maxInstances="xs:integer | xs:string"?>
3238 85             <Properties>
3239 86                 XML fragment
3240 87             </Properties> ?
3241 88
3242 89             <PropertyConstraints>
3243 90                 <PropertyConstraint property="xs:string"
3244 91                     constraintType="xs:anyURI">
3245 92                     constraint ?
3246 93                 </PropertyConstraint> +
3247 94             </PropertyConstraints> ?
3248 95
3249 96             <Requirements>
3250 97                 <Requirement id="xs:ID" name="xs:string" type="xs:QName"> +
3251 98                     <Properties>
3252 99                         XML fragment
3253 100                     <Properties> ?
3254 101                     <PropertyConstraints>
3255 102                         <PropertyConstraint property="xs:string"
3256 103                             constraintType="xs:anyURI"> +
3257 104                             constraint ?
3258 105                         </PropertyConstraint>
3259 106                     </PropertyConstraints> ?
3260 107                 </Requirement>
3261 108             </Requirements> ?
3262 109

```

```

3263 110      <Capabilities>
3264 111          <Capability id="xs:ID" name="xs:string"
3265 112              type="xs:QName"> +
3266 113              <Properties>
3267 114                  XML fragment
3268 115              <Properties> ?
3269 116              <PropertyConstraints>
3270 117                  <PropertyConstraint property="xs:string"
3271 118                      constraintType="xs:anyURI">
3272 119                      constraint ?
3273 120                  </PropertyConstraint> +
3274 121              </PropertyConstraints> ?
3275 122          </Capability>
3276 123      </Capabilities> ?
3277 124
3278 125      <Policies>
3279 126          <Policy name="xs:string"? policyType="xs:QName"
3280 127              policyRef="xs:QName"?>
3281 128              policy specific content ?
3282 129          </Policy> +
3283 130      </Policies> ?
3284 131
3285 132      <DeploymentArtifacts>
3286 133          <DeploymentArtifact name="xs:string"
3287 134              artifactType="xs:QName"
3288 135              artifactRef="xs:QName"?>
3289 136              artifact specific content ?
3290 137          </DeploymentArtifact> +
3291 138      </DeploymentArtifacts> ?
3292 139  </NodeTemplate>
3293 140  |
3294 141  <RelationshipTemplate id="xs:ID" name="xs:string"?
3295 142              type="xs:QName">
3296 143      <Properties>
3297 144          XML fragment
3298 145      </Properties> ?
3299 146
3300 147      <PropertyConstraints>
3301 148          <PropertyConstraint property="xs:string"
3302 149              constraintType="xs:anyURI">
3303 150              constraint ?
3304 151          </PropertyConstraint> +
3305 152      </PropertyConstraints> ?
3306 153
3307 154      <SourceElement ref="xs:IDREF"/>
3308 155      <TargetElement ref="xs:IDREF"/>
3309 156
3310 157      <RelationshipConstraints>
3311 158          <RelationshipConstraint constraintType="xs:anyURI">
3312 159              constraint ?
3313 160          </RelationshipConstraint> +
3314 161      </RelationshipConstraints> ?
3315 162
3316 163      </RelationshipTemplate>
3317 164  ) +
3318 165  </TopologyTemplate>
3319 166
3320 167  <Plans>

```

```

3321 168     <Plan id="xs:ID"
3322 169         name="xs:string"?
3323 170         planType="xs:anyURI"
3324 171         planLanguage="xs:anyURI">
3325 172
3326 173         <Precondition expressionLanguage="xs:anyURI">
3327 174             condition
3328 175         </Precondition> ?
3329 176
3330 177         <InputParameters>
3331 178             <InputParameter name="xs:string" type="xs:string"
3332 179                 required="yes|no"?/> +
3333 180         </InputParameters> ?
3334 181
3335 182         <OutputParameters>
3336 183             <OutputParameter name="xs:string" type="xs:string"
3337 184                 required="yes|no"?/> +
3338 185         </OutputParameters> ?
3339 186
3340 187         (
3341 188             <PlanModel>
3342 189                 actual plan
3343 190             </PlanModel>
3344 191             |
3345 192             <PlanModelReference reference="xs:anyURI"/>
3346 193         )
3347 194
3348 195     </Plan> +
3349 196 </Plans> ?
3350 197
3351 198 </ServiceTemplate>
3352 199 |
3353 200 <NodeType name="xs:NCName" targetNamespace="xs:anyURI"?
3354 201     abstract="yes|no"? final="yes|no"?>
3355 202
3356 203     <DerivedFrom typeRef="xs:QName"/> ?
3357 204
3358 205     <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
3359 206
3360 207     <RequirementDefinitions>
3361 208         <RequirementDefinition name="xs:string"
3362 209             requirementType="xs:QName"
3363 210             lowerBound="xs:integer"?
3364 211             upperBound="xs:integer | xs:string"?>
3365 212             <Constraints>
3366 213                 <Constraint constraintType="xs:anyURI">
3367 214                     constraint type specific content
3368 215                 </Constraint> +
3369 216             </Constraints> ?
3370 217         </RequirementDefinition> +
3371 218     </RequirementDefinitions> ?
3372 219
3373 220     <CapabilityDefinitions>
3374 221         <CapabilityDefinition name="xs:string"
3375 222             capabilityType="xs:QName"
3376 223             lowerBound="xs:integer"?
3377 224             upperBound="xs:integer | xs:string"?>
3378 225     </CapabilityDefinition> +
3378 225     <Constraints>

```



```

3379 226         <Constraint constraintType="xs:anyURI">
3380 227             constraint type specific content
3381 228         </Constraint> +
3382 229     </Constraints> ?
3383 230     </CapabilityDefinition> +
3384 231 </CapabilityDefinitions>
3385 232
3386 233 <InstanceStates>
3387 234     <InstanceState state="xs:anyURI"> +
3388 235 </InstanceState> ?
3389 236
3390 237 <Interfaces>
3391 238     <Interface name="xs:NCName | xs:anyURI">
3392 239         <Operation name="xs:NCName">
3393 240             <InputParameters>
3394 241                 <InputParameter name="xs:string" type="xs:string"
3395 242                     required="yes|no"?/> +
3396 243             </InputParameters> ?
3397 244             <OutputParameters>
3398 245                 <OutputParameter name="xs:string" type="xs:string"
3399 246                     required="yes|no"?/> +
3400 247             </OutputParameters> ?
3401 248         </Operation> +
3402 249     </Interface> +
3403 250 </Interfaces> ?
3404 251
3405 252 </NodeType>
3406 253 |
3407 254 <NodeTypeImplementation name="xs:NCName"
3408 255     targetNamespace="xs:anyURI"?
3409 256     nodeType="xs:QName"
3410 257     abstract="yes|no"?
3411 258     final="yes|no"?>
3412 259
3413 260     <DerivedFrom nodeTypeImplementationRef="xs:QName"/> ?
3414 261
3415 262     <RequiredContainerFeatures>
3416 263         <RequiredContainerFeature feature="xs:anyURI"/> +
3417 264     </RequiredContainerFeatures> ?
3418 265
3419 266     <ImplementationArtifacts>
3420 267         <ImplementationArtifact interfaceName="xs:NCName | xs:anyURI"?
3421 268             operationName="xs:NCName"?
3422 269             artifactType="xs:QName"
3423 270             artifactRef="xs:QName"?>
3424 271             artifact specific content ?
3425 272         <ImplementationArtifact> +
3426 273     </ImplementationArtifacts> ?
3427 274
3428 275     <DeploymentArtifacts>
3429 276         <DeploymentArtifact name="xs:string" artifactType="xs:QName"
3430 277             artifactRef="xs:QName"?>
3431 278             artifact specific content ?
3432 279         <DeploymentArtifact> +
3433 280     </DeploymentArtifacts> ?
3434 281
3435 282 </NodeTypeImplementation>
3436 283 |

```

```

3437 284     <RelationshipType name="xs:NCName"
3438 285         targetNamespace="xs:anyURI"?
3439 286         abstract="yes|no"?
3440 287         final="yes|no"?> +
3441 288
3442 289     <DerivedFrom typeRef="xs:QName"/> ?
3443 290
3444 291     <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
3445 292
3446 293     <InstanceStates>
3447 294         <InstanceState state="xs:anyURI"> +
3448 295     </InstanceStates> ?
3449 296
3450 297     <SourceInterfaces>
3451 298         <Interface name="xs:NCName | xs:anyURI">
3452 299             <Operation name="xs:NCName">
3453 300                 <InputParameters>
3454 301                     <InputParameter name="xs:string" type="xs:string"
3455 302                         required="yes|no"?/> +
3456 303                 </InputParameters> ?
3457 304                 <OutputParameters>
3458 305                     <OutputParameter name="xs:string" type="xs:string"
3459 306                         required="yes|no"?/> +
3460 307                 </OutputParameters> ?
3461 308             </Operation> +
3462 309         </Interface> +
3463 310     </SourceInterfaces> ?
3464 311
3465 312     <TargetInterfaces>
3466 313         <Interface name="xs:NCName | xs:anyURI">
3467 314             <Operation name="xs:NCName">
3468 315                 <InputParameters>
3469 316                     <InputParameter name="xs:string" type="xs:string"
3470 317                         required="yes|no"?/> +
3471 318                 </InputParameters> ?
3472 319                 <OutputParameters>
3473 320                     <OutputParameter name="xs:string" type="xs:string"
3474 321                         required="yes|no"?/> +
3475 322                 </OutputParameters> ?
3476 323             </Operation> +
3477 324         </Interface> +
3478 325     </TargetInterfaces> ?
3479 326
3480 327     <ValidSource typeRef="xs:QName"/> ?
3481 328
3482 329     <ValidTarget typeRef="xs:QName"/> ?
3483 330
3484 331 </RelationshipType>
3485 332 |
3486 333 <RelationshipTypeImplementation name="xs:NCName"
3487 334     targetNamespace="xs:anyURI"?
3488 335     relationshipType="xs:QName"
3489 336     abstract="yes|no"?
3490 337     final="yes|no"?>
3491 338
3492 339     <DerivedFrom relationshipTypeImplementationRef="xs:QName"/> ?
3493 340
3494 341     <RequiredContainerFeatures>

```

```

3495 342         <RequiredContainerFeature feature="xs:anyURI"/> +
3496 343     </RequiredContainerFeatures> ?
3497 344
3498 345     <ImplementationArtifacts>
3499 346         <ImplementationArtifact interfaceName="xs:NCName | xs:anyURI"?
3500 347             operationName="xs:NCName"?
3501 348             artifactType="xs:QName"
3502 349             artifactRef="xs:QName"?>
3503 350         artifact specific content ?
3504 351         <ImplementationArtifact> +
3505 352     </ImplementationArtifacts> ?
3506 353
3507 354 </RelationshipTypeImplementation>
3508 355 |
3509 356 <RequirementType name="xs:NCName"
3510 357     targetNamespace="xs:anyURI"?
3511 358     abstract="yes|no"?
3512 359     final="yes|no"?
3513 360     requiredCapabilityType="xs:QName"?>
3514 361
3515 362     <DerivedFrom typeRef="xs:QName"/> ?
3516 363
3517 364     <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
3518 365
3519 366 </RequirementType>
3520 367 |
3521 368 <CapabilityType name="xs:NCName"
3522 369     targetNamespace="xs:anyURI"?
3523 370     abstract="yes|no"?
3524 371     final="yes|no"?>
3525 372
3526 373     <DerivedFrom typeRef="xs:QName"/> ?
3527 374
3528 375     <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
3529 376
3530 377 </CapabilityType>
3531 378 |
3532 379 <ArtifactType name="xs:NCName"
3533 380     targetNamespace="xs:anyURI"?
3534 381     abstract="yes|no"?
3535 382     final="yes|no"?>
3536 383
3537 384     <DerivedFrom typeRef="xs:QName"/> ?
3538 385
3539 386     <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
3540 387
3541 388 </ArtifactType>
3542 389 |
3543 390 <ArtifactTemplate id="xs:ID" name="xs:string"? type="xs:QName">
3544 391
3545 392     <Properties>
3546 393         XML fragment
3547 394     </Properties> ?
3548 395
3549 396     <PropertyConstraints>
3550 397         <PropertyConstraint property="xs:string"
3551 398             constraintType="xs:anyURI"> +
3552 399     constraint ?

```

```

3553 400         </PropertyConstraint>
3554 401     </PropertyConstraints> ?
3555 402
3556 403     <ArtifactReferences>
3557 404         <ArtifactReference reference="xs:anyURI">
3558 405             (
3559 406                 <Include pattern="xs:string"/>
3560 407                 |
3561 408                 <Exclude pattern="xs:string"/>
3562 409             ) *
3563 410         </ArtifactReference> +
3564 411     </ArtifactReferences> ?
3565 412
3566 413 </ArtifactTemplate>
3567 414 |
3568 415 <PolicyType name="xs:NCName"
3569 416             policyLanguage="xs:anyURI"?
3570 417             abstract="yes|no"?
3571 418             final="yes|no"?
3572 419             targetNamespace="xs:anyURI"?>
3573 420     <Tags>
3574 421         <Tag name="xs:string" value="xs:string"/> +
3575 422     </Tags> ?
3576 423
3577 424     <DerivedFrom typeRef="xs:QName"/> ?
3578 425
3579 426     <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
3580 427
3581 428     <AppliesTo>
3582 429         <NodeTypeReference typeRef="xs:QName"/> +
3583 430     </AppliesTo> ?
3584 431
3585 432     policy type specific content ?
3586 433
3587 434 </PolicyType>
3588 435 |
3589 436 <PolicyTemplate id="xs:ID" name="xs:string"? type="xs:QName">
3590 437
3591 438     <Properties>
3592 439         XML fragment
3593 440     </Properties> ?
3594 441
3595 442     <PropertyConstraints>
3596 443         <PropertyConstraint property="xs:string"
3597 444                             constraintType="xs:anyURI"> +
3598 445             constraint ?
3599 446         </PropertyConstraint>
3600 447     </PropertyConstraints> ?
3601 448
3602 449     policy type specific content ?
3603 450
3604 451 </PolicyTemplate>
3605 452 ) +
3606 453
3607 454 </Definitions>

```

Appendix D. TOSCA Schema

TOSCA-v1.0.xsd:

```
3610 01 <?xml version="1.0" encoding="UTF-8"?>
3611 02 <xs:schema targetNamespace="http://docs.oasis-open.org/tosca/ns/2011/12"
3612 03   elementFormDefault="qualified" attributeFormDefault="unqualified"
3613 04   xmlns="http://docs.oasis-open.org/tosca/ns/2011/12"
3614 05   xmlns:xs="http://www.w3.org/2001/XMLSchema">
3615 06
3616 07   <xs:import namespace="http://www.w3.org/XML/1998/namespace"
3617 08     schemaLocation="http://www.w3.org/2001/xml.xsd"/>
3618 09
3619 10   <xs:element name="documentation" type="tDocumentation"/>
3620 11   <xs:complexType name="tDocumentation" mixed="true">
3621 12     <xs:sequence>
3622 13       <xs:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
3623 14     </xs:sequence>
3624 15     <xs:attribute name="source" type="xs:anyURI"/>
3625 16     <xs:attribute ref="xml:lang"/>
3626 17   </xs:complexType>
3627 18
3628 19   <xs:complexType name="tExtensibleElements">
3629 20     <xs:sequence>
3630 21       <xs:element ref="documentation" minOccurs="0" maxOccurs="unbounded"/>
3631 22       <xs:any namespace="##other" processContents="lax" minOccurs="0"
3632 23         maxOccurs="unbounded"/>
3633 24     </xs:sequence>
3634 25     <xs:anyAttribute namespace="##other" processContents="lax"/>
3635 26   </xs:complexType>
3636 27
3637 28   <xs:complexType name="tImport">
3638 29     <xs:complexContent>
3639 30       <xs:extension base="tExtensibleElements">
3640 31         <xs:attribute name="namespace" type="xs:anyURI"/>
3641 32         <xs:attribute name="location" type="xs:anyURI"/>
3642 33         <xs:attribute name="importType" type="importedURI" use="required"/>
3643 34       </xs:extension>
3644 35     </xs:complexContent>
3645 36   </xs:complexType>
3646 37
3647 38   <xs:element name="Definitions">
3648 39     <xs:complexType>
3649 40       <xs:complexContent>
3650 41         <xs:extension base="tDefinitions"/>
3651 42       </xs:complexContent>
3652 43     </xs:complexType>
3653 44   </xs:element>
3654 45   <xs:complexType name="tDefinitions">
3655 46     <xs:complexContent>
3656 47       <xs:extension base="tExtensibleElements">
3657 48         <xs:sequence>
3658 49           <xs:element name="Extensions" minOccurs="0">
3659 50             <xs:complexType>
3660 51               <xs:sequence>
3661 52                 <xs:element name="Extension" type="tExtension"
```

```

3662 53         maxOccurs="unbounded"/>
3663 54     </xs:sequence>
3664 55 </xs:complexType>
3665 56 </xs:element>
3666 57 <xs:element name="Import" type="tImport" minOccurs="0"
3667 58     maxOccurs="unbounded"/>
3668 59 <xs:element name="Types" minOccurs="0">
3669 60     <xs:complexType>
3670 61         <xs:sequence>
3671 62             <xs:any namespace="##other" processContents="lax" minOccurs="0"
3672 63                 maxOccurs="unbounded"/>
3673 64         </xs:sequence>
3674 65     </xs:complexType>
3675 66 </xs:element>
3676 67 <xs:choice maxOccurs="unbounded">
3677 68     <xs:element name="ServiceTemplate" type="tServiceTemplate"/>
3678 69     <xs:element name="NodeType" type="tNodeType"/>
3679 70     <xs:element name="NodeTypeImplementation"
3680 71         type="tNodeTypeImplementation"/>
3681 72     <xs:element name="RelationshipType" type="tRelationshipType"/>
3682 73     <xs:element name="RelationshipTypeImplementation"
3683 74         type="tRelationshipTypeImplementation"/>
3684 75     <xs:element name="RequirementType" type="tRequirementType"/>
3685 76     <xs:element name="CapabilityType" type="tCapabilityType"/>
3686 77     <xs:element name="ArtifactType" type="tArtifactType"/>
3687 78     <xs:element name="ArtifactTemplate" type="tArtifactTemplate"/>
3688 79     <xs:element name="PolicyType" type="tPolicyType"/>
3689 80     <xs:element name="PolicyTemplate" type="tPolicyTemplate"/>
3690 81 </xs:choice>
3691 82 </xs:sequence>
3692 83 <xs:attribute name="id" type="xs:ID" use="required"/>
3693 84 <xs:attribute name="name" type="xs:string" use="optional"/>
3694 85 <xs:attribute name="targetNamespace" type="xs:anyURI" use="required"/>
3695 86 </xs:extension>
3696 87 </xs:complexContent>
3697 88 </xs:complexType>
3698 89
3699 90 <xs:complexType name="tServiceTemplate">
3700 91     <xs:complexContent>
3701 92         <xs:extension base="tExtensibleElements">
3702 93             <xs:sequence>
3703 94                 <xs:element name="Tags" type="tTags" minOccurs="0"/>
3704 95                 <xs:element name="BoundaryDefinitions" type="tBoundaryDefinitions"
3705 96                     minOccurs="0"/>
3706 97                 <xs:element name="TopologyTemplate" type="tTopologyTemplate"/>
3707 98                 <xs:element name="Plans" type="tPlans" minOccurs="0"/>
3708 99             </xs:sequence>
3709 100             <xs:attribute name="id" type="xs:ID" use="required"/>
3710 101             <xs:attribute name="name" type="xs:string" use="optional"/>
3711 102             <xs:attribute name="targetNamespace" type="xs:anyURI"/>
3712 103             <xs:attribute name="substitutableNodeType" type="xs:QName"
3713 104                 use="optional"/>
3714 105             </xs:extension>
3715 106         </xs:complexContent>
3716 107     </xs:complexType>
3717 108
3718 109 <xs:complexType name="tTags">
3719 110     <xs:sequence>

```

```

3720 111     <xs:element name="Tag" type="tTag" maxOccurs="unbounded"/>
3721 112     </xs:sequence>
3722 113 </xs:complexType>
3723 114
3724 115 <xs:complexType name="tTag">
3725 116     <xs:attribute name="name" type="xs:string" use="required"/>
3726 117     <xs:attribute name="value" type="xs:string" use="required"/>
3727 118 </xs:complexType>
3728 119
3729 120 <xs:complexType name="tBoundaryDefinitions">
3730 121     <xs:sequence>
3731 122         <xs:element name="Properties" minOccurs="0">
3732 123             <xs:complexType>
3733 124                 <xs:sequence>
3734 125                     <xs:any namespace="##other"/>
3735 126                     <xs:element name="PropertyMappings" minOccurs="0">
3736 127                         <xs:complexType>
3737 128                             <xs:sequence>
3738 129                                 <xs:element name="PropertyMapping" type="tPropertyMapping"
3739 130                                     maxOccurs="unbounded"/>
3740 131                             </xs:sequence>
3741 132                         </xs:complexType>
3742 133                     </xs:element>
3743 134                 </xs:sequence>
3744 135             </xs:complexType>
3745 136         </xs:element>
3746 137     <xs:element name="PropertyConstraints" minOccurs="0">
3747 138         <xs:complexType>
3748 139             <xs:sequence>
3749 140                 <xs:element name="PropertyConstraint" type="tPropertyConstraint"
3750 141                     maxOccurs="unbounded"/>
3751 142             </xs:sequence>
3752 143         </xs:complexType>
3753 144     </xs:element>
3754 145 <xs:element name="Requirements" minOccurs="0">
3755 146     <xs:complexType>
3756 147         <xs:sequence>
3757 148             <xs:element name="Requirement" type="tRequirementRef"
3758 149                 maxOccurs="unbounded"/>
3759 150         </xs:sequence>
3760 151     </xs:complexType>
3761 152 </xs:element>
3762 153 <xs:element name="Capabilities" minOccurs="0">
3763 154     <xs:complexType>
3764 155         <xs:sequence>
3765 156             <xs:element name="Capability" type="tCapabilityRef"
3766 157                 maxOccurs="unbounded"/>
3767 158         </xs:sequence>
3768 159     </xs:complexType>
3769 160 </xs:element>
3770 161 <xs:element name="Policies" minOccurs="0">
3771 162     <xs:complexType>
3772 163         <xs:sequence>
3773 164             <xs:element name="Policy" type="tPolicy" maxOccurs="unbounded"/>
3774 165         </xs:sequence>
3775 166     </xs:complexType>
3776 167 </xs:element>
3777 168 <xs:element name="Interfaces" minOccurs="0">

```

```

3778 169     <xs:complexType>
3779 170     <xs:sequence>
3780 171         <xs:element name="Interface" type="tExportedInterface"
3781 172             maxOccurs="unbounded"/>
3782 173     </xs:sequence>
3783 174 </xs:complexType>
3784 175 </xs:element>
3785 176 </xs:sequence>
3786 177 </xs:complexType>
3787 178
3788 179 <xs:complexType name="tPropertyMapping">
3789 180     <xs:attribute name="serviceTemplatePropertyRef" type="xs:string"
3790 181         use="required"/>
3791 182     <xs:attribute name="targetObjectRef" type="xs:IDREF" use="required"/>
3792 183     <xs:attribute name="targetPropertyRef" type="xs:string"
3793 184         use="required"/>
3794 185 </xs:complexType>
3795 186
3796 187 <xs:complexType name="tRequirementRef">
3797 188     <xs:attribute name="name" type="xs:string" use="optional"/>
3798 189     <xs:attribute name="ref" type="xs:IDREF" use="required"/>
3799 190 </xs:complexType>
3800 191
3801 192 <xs:complexType name="tCapabilityRef">
3802 193     <xs:attribute name="name" type="xs:string" use="optional"/>
3803 194     <xs:attribute name="ref" type="xs:IDREF" use="required"/>
3804 195 </xs:complexType>
3805 196
3806 197 <xs:complexType name="tEntityType" abstract="true">
3807 198     <xs:complexContent>
3808 199         <xs:extension base="tExtensibleElements">
3809 200             <xs:sequence>
3810 201                 <xs:element name="Tags" type="tTags" minOccurs="0"/>
3811 202                 <xs:element name="DerivedFrom" minOccurs="0">
3812 203                     <xs:complexType>
3813 204                         <xs:attribute name="typeRef" type="xs:QName" use="required"/>
3814 205                     </xs:complexType>
3815 206                 </xs:element>
3816 207                 <xs:element name="PropertiesDefinition" minOccurs="0">
3817 208                     <xs:complexType>
3818 209                         <xs:attribute name="element" type="xs:QName"/>
3819 210                         <xs:attribute name="type" type="xs:QName"/>
3820 211                     </xs:complexType>
3821 212                 </xs:element>
3822 213             </xs:sequence>
3823 214             <xs:attribute name="name" type="xs:NCName" use="required"/>
3824 215             <xs:attribute name="abstract" type="tBoolean" default="no"/>
3825 216             <xs:attribute name="final" type="tBoolean" default="no"/>
3826 217             <xs:attribute name="targetNamespace" type="xs:anyURI"
3827 218                 use="optional"/>
3828 219         </xs:extension>
3829 220     </xs:complexContent>
3830 221 </xs:complexType>
3831 222
3832 223 <xs:complexType name="tEntityTypeTemplate" abstract="true">
3833 224     <xs:complexContent>
3834 225         <xs:extension base="tExtensibleElements">
3835 226             <xs:sequence>

```



```

3836 227     <xs:element name="Properties" minOccurs="0">
3837 228         <xs:complexType>
3838 229             <xs:sequence>
3839 230                 <xs:any namespace="##other" processContents="lax"/>
3840 231             </xs:sequence>
3841 232         </xs:complexType>
3842 233     </xs:element>
3843 234     <xs:element name="PropertyConstraints" minOccurs="0">
3844 235         <xs:complexType>
3845 236             <xs:sequence>
3846 237                 <xs:element name="PropertyConstraint"
3847 238                     type="tPropertyConstraint" maxOccurs="unbounded"/>
3848 239                 </xs:sequence>
3849 240             </xs:complexType>
3850 241         </xs:element>
3851 242     </xs:sequence>
3852 243     <xs:attribute name="id" type="xs:ID" use="required"/>
3853 244     <xs:attribute name="type" type="xs:QName" use="required"/>
3854 245 </xs:extension>
3855 246 </xs:complexContent>
3856 247 </xs:complexType>
3857 248
3858 249 <xs:complexType name="tNodeTemplate">
3859 250     <xs:complexContent>
3860 251         <xs:extension base="tEntityTemplate">
3861 252             <xs:sequence>
3862 253                 <xs:element name="Requirements" minOccurs="0">
3863 254                     <xs:complexType>
3864 255                         <xs:sequence>
3865 256                             <xs:element name="Requirement" type="tRequirement"
3866 257                                 maxOccurs="unbounded"/>
3867 258                         </xs:sequence>
3868 259                     </xs:complexType>
3869 260                 </xs:element>
3870 261                 <xs:element name="Capabilities" minOccurs="0">
3871 262                     <xs:complexType>
3872 263                         <xs:sequence>
3873 264                             <xs:element name="Capability" type="tCapability"
3874 265                                 maxOccurs="unbounded"/>
3875 266                         </xs:sequence>
3876 267                     </xs:complexType>
3877 268                 </xs:element>
3878 269                 <xs:element name="Policies" minOccurs="0">
3879 270                     <xs:complexType>
3880 271                         <xs:sequence>
3881 272                             <xs:element name="Policy" type="tPolicy"
3882 273                                 maxOccurs="unbounded"/>
3883 274                         </xs:sequence>
3884 275                     </xs:complexType>
3885 276                 </xs:element>
3886 277                 <xs:element name="DeploymentArtifacts" type="tDeploymentArtifacts"
3887 278                     minOccurs="0"/>
3888 279             </xs:sequence>
3889 280             <xs:attribute name="name" type="xs:string" use="optional"/>
3890 281             <xs:attribute name="minInstances" type="xs:int" use="optional"
3891 282                 default="1"/>
3892 283             <xs:attribute name="maxInstances" use="optional" default="1">
3893 284                 <xs:simpleType>

```

```

3894 285     <xs:union>
3895 286     <xs:simpleType>
3896 287         <xs:restriction base="xs:nonNegativeInteger">
3897 288             <xs:pattern value="([1-9]+[0-9]*)"/>
3898 289         </xs:restriction>
3899 290     </xs:simpleType>
3900 291     <xs:simpleType>
3901 292         <xs:restriction base="xs:string">
3902 293             <xs:enumeration value="unbounded"/>
3903 294         </xs:restriction>
3904 295     </xs:simpleType>
3905 296 </xs:union>
3906 297 </xs:simpleType>
3907 298 </xs:attribute>
3908 299 </xs:extension>
3909 300 </xs:complexContent>
3910 301 </xs:complexType>
3911 302
3912 303 <xs:complexType name="tTopologyTemplate">
3913 304     <xs:complexContent>
3914 305         <xs:extension base="tExtensibleElements">
3915 306             <xs:choice maxOccurs="unbounded">
3916 307                 <xs:element name="NodeTemplate" type="tNodeTemplate"/>
3917 308                 <xs:element name="RelationshipTemplate"
3918 309                     type="tRelationshipTemplate"/>
3919 310             </xs:choice>
3920 311         </xs:extension>
3921 312     </xs:complexContent>
3922 313 </xs:complexType>
3923 314
3924 315 <xs:complexType name="tRelationshipType">
3925 316     <xs:complexContent>
3926 317         <xs:extension base="tEntityType">
3927 318             <xs:sequence>
3928 319                 <xs:element name="InstanceStates"
3929 320                     type="tTopologyElementInstanceStates" minOccurs="0"/>
3930 321                 <xs:element name="SourceInterfaces" minOccurs="0">
3931 322                     <xs:complexType>
3932 323                         <xs:sequence>
3933 324                             <xs:element name="Interface" type="tInterface"
3934 325                                 maxOccurs="unbounded"/>
3935 326                         </xs:sequence>
3936 327                     </xs:complexType>
3937 328                 </xs:element>
3938 329                 <xs:element name="TargetInterfaces" minOccurs="0">
3939 330                     <xs:complexType>
3940 331                         <xs:sequence>
3941 332                             <xs:element name="Interface" type="tInterface"
3942 333                                 maxOccurs="unbounded"/>
3943 334                         </xs:sequence>
3944 335                     </xs:complexType>
3945 336                 </xs:element>
3946 337                 <xs:element name="ValidSource" minOccurs="0">
3947 338                     <xs:complexType>
3948 339                         <xs:attribute name="typeRef" type="xs:QName" use="required"/>
3949 340                     </xs:complexType>
3950 341                 </xs:element>
3951 342                 <xs:element name="ValidTarget" minOccurs="0">

```

```

3952 343     <xs:complexType>
3953 344     <xs:attribute name="typeRef" type="xs:QName" use="required"/>
3954 345     </xs:complexType>
3955 346     </xs:element>
3956 347     </xs:sequence>
3957 348     </xs:extension>
3958 349     </xs:complexContent>
3959 350 </xs:complexType>
3960 351
3961 352 <xs:complexType name="tRelationshipTypeImplementation">
3962 353   <xs:complexContent>
3963 354     <xs:extension base="tExtensibleElements">
3964 355       <xs:sequence>
3965 356         <xs:element name="Tags" type="tTags" minOccurs="0"/>
3966 357         <xs:element name="DerivedFrom" minOccurs="0">
3967 358           <xs:complexType>
3968 359             <xs:attribute name="relationshipTypeImplementationRef"
3969 360               type="xs:QName" use="required"/>
3970 361           </xs:complexType>
3971 362         </xs:element>
3972 363         <xs:element name="RequiredContainerFeatures"
3973 364           type="tRequiredContainerFeatures" minOccurs="0"/>
3974 365         <xs:element name="ImplementationArtifacts"
3975 366           type="tImplementationArtifacts" minOccurs="0"/>
3976 367       </xs:sequence>
3977 368       <xs:attribute name="name" type="xs:NCName" use="required"/>
3978 369       <xs:attribute name="targetNamespace" type="xs:anyURI"
3979 370         use="optional"/>
3980 371       <xs:attribute name="relationshipType" type="xs:QName"
3981 372         use="required"/>
3982 373       <xs:attribute name="abstract" type="tBoolean" use="optional"
3983 374         default="no"/>
3984 375       <xs:attribute name="final" type="tBoolean" use="optional"
3985 376         default="no"/>
3986 377     </xs:extension>
3987 378   </xs:complexContent>
3988 379 </xs:complexType>
3989 380
3990 381 <xs:complexType name="tRelationshipTemplate">
3991 382   <xs:complexContent>
3992 383     <xs:extension base="tEntityTemplate">
3993 384       <xs:sequence>
3994 385         <xs:element name="SourceElement">
3995 386           <xs:complexType>
3996 387             <xs:attribute name="ref" type="xs:IDREF" use="required"/>
3997 388           </xs:complexType>
3998 389         </xs:element>
3999 390         <xs:element name="TargetElement">
4000 391           <xs:complexType>
4001 392             <xs:attribute name="ref" type="xs:IDREF" use="required"/>
4002 393           </xs:complexType>
4003 394         </xs:element>
4004 395         <xs:element name="RelationshipConstraints" minOccurs="0">
4005 396           <xs:complexType>
4006 397             <xs:sequence>
4007 398               <xs:element name="RelationshipConstraint"
4008 399                 maxOccurs="unbounded">
4009 400             <xs:complexType>

```

```

4010 401         <xs:sequence>
4011 402         <xs:any namespace="##other" processContents="lax"
4012 403             minOccurs="0"/>
4013 404         </xs:sequence>
4014 405         <xs:attribute name="constraintType" type="xs:anyURI"
4015 406             use="required"/>
4016 407         </xs:complexType>
4017 408     </xs:element>
4018 409 </xs:sequence>
4019 410 </xs:complexType>
4020 411 </xs:element>
4021 412 </xs:sequence>
4022 413 <xs:attribute name="name" type="xs:string" use="optional"/>
4023 414 </xs:extension>
4024 415 </xs:complexContent>
4025 416 </xs:complexType>
4026 417
4027 418 <xs:complexType name="tNodeType">
4028 419 <xs:complexContent>
4029 420 <xs:extension base="tEntityType">
4030 421 <xs:sequence>
4031 422 <xs:element name="RequirementDefinitions" minOccurs="0">
4032 423 <xs:complexType>
4033 424 <xs:sequence>
4034 425 <xs:element name="RequirementDefinition"
4035 426     type="tRequirementDefinition" maxOccurs="unbounded"/>
4036 427 </xs:sequence>
4037 428 </xs:complexType>
4038 429 </xs:element>
4039 430 <xs:element name="CapabilityDefinitions" minOccurs="0">
4040 431 <xs:complexType>
4041 432 <xs:sequence>
4042 433 <xs:element name="CapabilityDefinition"
4043 434     type="tCapabilityDefinition" maxOccurs="unbounded"/>
4044 435 </xs:sequence>
4045 436 </xs:complexType>
4046 437 </xs:element>
4047 438 <xs:element name="InstanceStates"
4048 439     type="tTopologyElementInstanceStates" minOccurs="0"/>
4049 440 <xs:element name="Interfaces" minOccurs="0">
4050 441 <xs:complexType>
4051 442 <xs:sequence>
4052 443 <xs:element name="Interface" type="tInterface"
4053 444     maxOccurs="unbounded"/>
4054 445 </xs:sequence>
4055 446 </xs:complexType>
4056 447 </xs:element>
4057 448 </xs:sequence>
4058 449 </xs:extension>
4059 450 </xs:complexContent>
4060 451 </xs:complexType>
4061 452
4062 453 <xs:complexType name="tNodeTypeImplementation">
4063 454 <xs:complexContent>
4064 455 <xs:extension base="tExtensibleElements">
4065 456 <xs:sequence>
4066 457 <xs:element name="Tags" type="tTags" minOccurs="0"/>
4067 458 <xs:element name="DerivedFrom" minOccurs="0">

```

```

4068 459     <xs:complexType>
4069 460     <xs:attribute name="nodeTypeImplementationRef" type="xs:QName"
4070 461         use="required"/>
4071 462     </xs:complexType>
4072 463 </xs:element>
4073 464 <xs:element name="RequiredContainerFeatures"
4074 465     type="tRequiredContainerFeatures" minOccurs="0"/>
4075 466 <xs:element name="ImplementationArtifacts"
4076 467     type="tImplementationArtifacts" minOccurs="0"/>
4077 468 <xs:element name="DeploymentArtifacts" type="tDeploymentArtifacts"
4078 469     minOccurs="0"/>
4079 470 </xs:sequence>
4080 471 <xs:attribute name="name" type="xs:NCName" use="required"/>
4081 472 <xs:attribute name="targetNamespace" type="xs:anyURI"
4082 473     use="optional"/>
4083 474 <xs:attribute name="nodeType" type="xs:QName" use="required"/>
4084 475 <xs:attribute name="abstract" type="tBoolean" use="optional"
4085 476     default="no"/>
4086 477 <xs:attribute name="final" type="tBoolean" use="optional"
4087 478     default="no"/>
4088 479 </xs:extension>
4089 480 </xs:complexContent>
4090 481 </xs:complexType>
4091 482
4092 483 <xs:complexType name="tRequirementType">
4093 484 <xs:complexContent>
4094 485 <xs:extension base="tEntityType">
4095 486 <xs:attribute name="requiredCapabilityType" type="xs:QName"
4096 487     use="optional"/>
4097 488 </xs:extension>
4098 489 </xs:complexContent>
4099 490 </xs:complexType>
4100 491
4101 492 <xs:complexType name="tRequirementDefinition">
4102 493 <xs:complexContent>
4103 494 <xs:extension base="tExtensibleElements">
4104 495 <xs:sequence>
4105 496 <xs:element name="Constraints" minOccurs="0">
4106 497 <xs:complexType>
4107 498 <xs:sequence>
4108 499 <xs:element name="Constraint" type="tConstraint"
4109 500     maxOccurs="unbounded"/>
4110 501 </xs:sequence>
4111 502 </xs:complexType>
4112 503 </xs:element>
4113 504 </xs:sequence>
4114 505 <xs:attribute name="name" type="xs:string" use="required"/>
4115 506 <xs:attribute name="requirementType" type="xs:QName"
4116 507     use="required"/>
4117 508 <xs:attribute name="lowerBound" type="xs:int" use="optional"
4118 509     default="1"/>
4119 510 <xs:attribute name="upperBound" use="optional" default="1">
4120 511 <xs:simpleType>
4121 512 <xs:union>
4122 513 <xs:simpleType>
4123 514 <xs:restriction base="xs:nonNegativeInteger">
4124 515 <xs:pattern value="([1-9]+[0-9]*)"/>
4125 516 </xs:restriction>

```

```

4126 517         </xs:simpleType>
4127 518         <xs:simpleType>
4128 519             <xs:restriction base="xs:string">
4129 520                 <xs:enumeration value="unbounded"/>
4130 521             </xs:restriction>
4131 522         </xs:simpleType>
4132 523     </xs:union>
4133 524 </xs:simpleType>
4134 525 </xs:attribute>
4135 526 </xs:extension>
4136 527 </xs:complexContent>
4137 528 </xs:complexType>
4138 529
4139 530 <xs:complexType name="tRequirement">
4140 531     <xs:complexContent>
4141 532         <xs:extension base="tEntityType">
4142 533             <xs:attribute name="name" type="xs:string" use="required"/>
4143 534         </xs:extension>
4144 535     </xs:complexContent>
4145 536 </xs:complexType>
4146 537
4147 538 <xs:complexType name="tCapabilityType">
4148 539     <xs:complexContent>
4149 540         <xs:extension base="tEntityType"/>
4150 541     </xs:complexContent>
4151 542 </xs:complexType>
4152 543
4153 544 <xs:complexType name="tCapabilityDefinition">
4154 545     <xs:complexContent>
4155 546         <xs:extension base="tExtensibleElements">
4156 547             <xs:sequence>
4157 548                 <xs:element name="Constraints" minOccurs="0">
4158 549                     <xs:complexType>
4159 550                         <xs:sequence>
4160 551                             <xs:element name="Constraint" type="tConstraint"
4161 552                                 maxOccurs="unbounded"/>
4162 553                         </xs:sequence>
4163 554                     </xs:complexType>
4164 555                 </xs:element>
4165 556             </xs:sequence>
4166 557             <xs:attribute name="name" type="xs:string" use="required"/>
4167 558             <xs:attribute name="capabilityType" type="xs:QName"
4168 559                 use="required"/>
4169 560             <xs:attribute name="lowerBound" type="xs:int" use="optional"
4170 561                 default="1"/>
4171 562             <xs:attribute name="upperBound" use="optional" default="1">
4172 563                 <xs:simpleType>
4173 564                     <xs:union>
4174 565                         <xs:simpleType>
4175 566                             <xs:restriction base="xs:nonNegativeInteger">
4176 567                                 <xs:pattern value="([1-9]+[0-9]*)"/>
4177 568                             </xs:restriction>
4178 569                         </xs:simpleType>
4179 570                         <xs:simpleType>
4180 571                             <xs:restriction base="xs:string">
4181 572                                 <xs:enumeration value="unbounded"/>
4182 573                             </xs:restriction>
4183 574                         </xs:simpleType>

```

```

4184 575     </xs:union>
4185 576     </xs:simpleType>
4186 577     </xs:attribute>
4187 578     </xs:extension>
4188 579     </xs:complexContent>
4189 580 </xs:complexType>
4190 581
4191 582 <xs:complexType name="tCapability">
4192 583   <xs:complexContent>
4193 584     <xs:extension base="tEntityType">
4194 585       <xs:attribute name="name" type="xs:string" use="required"/>
4195 586     </xs:extension>
4196 587   </xs:complexContent>
4197 588 </xs:complexType>
4198 589
4199 590 <xs:complexType name="tArtifactType">
4200 591   <xs:complexContent>
4201 592     <xs:extension base="tEntityType"/>
4202 593   </xs:complexContent>
4203 594 </xs:complexType>
4204 595
4205 596 <xs:complexType name="tArtifactTemplate">
4206 597   <xs:complexContent>
4207 598     <xs:extension base="tEntityTypeTemplate">
4208 599       <xs:sequence>
4209 600         <xs:element name="ArtifactReferences" minOccurs="0">
4210 601           <xs:complexType>
4211 602             <xs:sequence>
4212 603               <xs:element name="ArtifactReference" type="tArtifactReference"
4213 604                 maxOccurs="unbounded"/>
4214 605             </xs:sequence>
4215 606           </xs:complexType>
4216 607         </xs:element>
4217 608       </xs:sequence>
4218 609       <xs:attribute name="name" type="xs:string" use="optional"/>
4219 610     </xs:extension>
4220 611   </xs:complexContent>
4221 612 </xs:complexType>
4222 613
4223 614 <xs:complexType name="tDeploymentArtifacts">
4224 615   <xs:sequence>
4225 616     <xs:element name="DeploymentArtifact" type="tDeploymentArtifact"
4226 617       maxOccurs="unbounded"/>
4227 618   </xs:sequence>
4228 619 </xs:complexType>
4229 620
4230 621 <xs:complexType name="tDeploymentArtifact">
4231 622   <xs:complexContent>
4232 623     <xs:extension base="tExtensibleElements">
4233 624       <xs:attribute name="name" type="xs:string" use="required"/>
4234 625       <xs:attribute name="artifactType" type="xs:QName" use="required"/>
4235 626       <xs:attribute name="artifactRef" type="xs:QName" use="optional"/>
4236 627     </xs:extension>
4237 628   </xs:complexContent>
4238 629 </xs:complexType>
4239 630
4240 631 <xs:complexType name="tImplementationArtifacts">
4241 632   <xs:sequence>

```

```

4242 633     <xs:element name="ImplementationArtifact" maxOccurs="unbounded">
4243 634         <xs:complexType>
4244 635             <xs:complexContent>
4245 636                 <xs:extension base="tImplementationArtifact"/>
4246 637             </xs:complexContent>
4247 638         </xs:complexType>
4248 639     </xs:element>
4249 640 </xs:sequence>
4250 641 </xs:complexType>
4251 642
4252 643 <xs:complexType name="tImplementationArtifact">
4253 644     <xs:complexContent>
4254 645         <xs:extension base="tExtensibleElements">
4255 646             <xs:attribute name="interfaceName" type="xs:anyURI"
4256 647                 use="optional"/>
4257 648             <xs:attribute name="operationName" type="xs:NCName"
4258 649                 use="optional"/>
4259 650             <xs:attribute name="artifactType" type="xs:QName" use="required"/>
4260 651             <xs:attribute name="artifactRef" type="xs:QName" use="optional"/>
4261 652         </xs:extension>
4262 653     </xs:complexContent>
4263 654 </xs:complexType>
4264 655
4265 656 <xs:complexType name="tPlans">
4266 657     <xs:sequence>
4267 658         <xs:element name="Plan" type="tPlan" maxOccurs="unbounded"/>
4268 659     </xs:sequence>
4269 660     <xs:attribute name="targetNamespace" type="xs:anyURI"
4270 661         use="optional"/>
4271 662 </xs:complexType>
4272 663
4273 664 <xs:complexType name="tPlan">
4274 665     <xs:complexContent>
4275 666         <xs:extension base="tExtensibleElements">
4276 667             <xs:sequence>
4277 668                 <xs:element name="Precondition" type="tCondition" minOccurs="0"/>
4278 669                 <xs:element name="InputParameters" minOccurs="0">
4279 670                     <xs:complexType>
4280 671                         <xs:sequence>
4281 672                             <xs:element name="InputParameter" type="tParameter"
4282 673                                 maxOccurs="unbounded"/>
4283 674                         </xs:sequence>
4284 675                     </xs:complexType>
4285 676                 </xs:element>
4286 677                 <xs:element name="OutputParameters" minOccurs="0">
4287 678                     <xs:complexType>
4288 679                         <xs:sequence>
4289 680                             <xs:element name="OutputParameter" type="tParameter"
4290 681                                 maxOccurs="unbounded"/>
4291 682                         </xs:sequence>
4292 683                     </xs:complexType>
4293 684                 </xs:element>
4294 685             <xs:choice>
4295 686                 <xs:element name="PlanModel">
4296 687                     <xs:complexType>
4297 688                         <xs:sequence>
4298 689                             <xs:any namespace="##other" processContents="lax"/>
4299 690                         </xs:sequence>

```



```

4300 691         </xs:complexType>
4301 692     </xs:element>
4302 693     <xs:element name="PlanModelReference">
4303 694         <xs:complexType>
4304 695             <xs:attribute name="reference" type="xs:anyURI"
4305 696                 use="required"/>
4306 697         </xs:complexType>
4307 698     </xs:element>
4308 699 </xs:choice>
4309 700 </xs:sequence>
4310 701 <xs:attribute name="id" type="xs:ID" use="required"/>
4311 702 <xs:attribute name="name" type="xs:string" use="optional"/>
4312 703 <xs:attribute name="planType" type="xs:anyURI" use="required"/>
4313 704 <xs:attribute name="planLanguage" type="xs:anyURI" use="required"/>
4314 705 </xs:extension>
4315 706 </xs:complexContent>
4316 707 </xs:complexType>
4317 708
4318 709 <xs:complexType name="tPolicyType">
4319 710     <xs:complexContent>
4320 711         <xs:extension base="tEntityType">
4321 712             <xs:sequence>
4322 713                 <xs:element name="AppliesTo" type="tAppliesTo" minOccurs="0"/>
4323 714             </xs:sequence>
4324 715             <xs:attribute name="policyLanguage" type="xs:anyURI"
4325 716                 use="optional"/>
4326 717         </xs:extension>
4327 718     </xs:complexContent>
4328 719 </xs:complexType>
4329 720
4330 721 <xs:complexType name="tPolicyTemplate">
4331 722     <xs:complexContent>
4332 723         <xs:extension base="tEntityTemplate">
4333 724             <xs:attribute name="name" type="xs:string" use="optional"/>
4334 725         </xs:extension>
4335 726     </xs:complexContent>
4336 727 </xs:complexType>
4337 728
4338 729 <xs:complexType name="tAppliesTo">
4339 730     <xs:sequence>
4340 731         <xs:element name="NodeTypeReference" maxOccurs="unbounded">
4341 732             <xs:complexType>
4342 733                 <xs:attribute name="typeRef" type="xs:QName" use="required"/>
4343 734             </xs:complexType>
4344 735         </xs:element>
4345 736     </xs:sequence>
4346 737 </xs:complexType>
4347 738
4348 739 <xs:complexType name="tPolicy">
4349 740     <xs:complexContent>
4350 741         <xs:extension base="tExtensibleElements">
4351 742             <xs:attribute name="name" type="xs:string" use="optional"/>
4352 743             <xs:attribute name="policyType" type="xs:QName" use="required"/>
4353 744             <xs:attribute name="policyRef" type="xs:QName" use="optional"/>
4354 745         </xs:extension>
4355 746     </xs:complexContent>
4356 747 </xs:complexType>
4357 748

```

```

4358 749 <xs:complexType name="tConstraint">
4359 750   <xs:sequence>
4360 751     <xs:any namespace="##other" processContents="lax"/>
4361 752   </xs:sequence>
4362 753   <xs:attribute name="constraintType" type="xs:anyURI" use="required"/>
4363 754 </xs:complexType>
4364 755
4365 756 <xs:complexType name="tPropertyConstraint">
4366 757   <xs:complexContent>
4367 758     <xs:extension base="tConstraint">
4368 759       <xs:attribute name="property" type="xs:string" use="required"/>
4369 760     </xs:extension>
4370 761   </xs:complexContent>
4371 762 </xs:complexType>
4372 763
4373 764 <xs:complexType name="tExtensions">
4374 765   <xs:complexContent>
4375 766     <xs:extension base="tExtensibleElements">
4376 767       <xs:sequence>
4377 768         <xs:element name="Extension" type="tExtension"
4378 769           maxOccurs="unbounded"/>
4379 770       </xs:sequence>
4380 771     </xs:extension>
4381 772   </xs:complexContent>
4382 773 </xs:complexType>
4383 774
4384 775 <xs:complexType name="tExtension">
4385 776   <xs:complexContent>
4386 777     <xs:extension base="tExtensibleElements">
4387 778       <xs:attribute name="namespace" type="xs:anyURI" use="required"/>
4388 779       <xs:attribute name="mustUnderstand" type="tBoolean" use="optional"
4389 780         default="yes"/>
4390 781     </xs:extension>
4391 782   </xs:complexContent>
4392 783 </xs:complexType>
4393 784
4394 785 <xs:complexType name="tParameter">
4395 786   <xs:attribute name="name" type="xs:string" use="required"/>
4396 787   <xs:attribute name="type" type="xs:string" use="required"/>
4397 788   <xs:attribute name="required" type="tBoolean" use="optional"
4398 789     default="yes"/>
4399 790 </xs:complexType>
4400 791
4401 792 <xs:complexType name="tInterface">
4402 793   <xs:sequence>
4403 794     <xs:element name="Operation" type="tOperation"
4404 795       maxOccurs="unbounded"/>
4405 796   </xs:sequence>
4406 797   <xs:attribute name="name" type="xs:anyURI" use="required"/>
4407 798 </xs:complexType>
4408 799
4409 800 <xs:complexType name="tExportedInterface">
4410 801   <xs:sequence>
4411 802     <xs:element name="Operation" type="tExportedOperation"
4412 803       maxOccurs="unbounded"/>
4413 804   </xs:sequence>
4414 805   <xs:attribute name="name" type="xs:anyURI" use="required"/>
4415 806 </xs:complexType>

```

```

4416 807
4417 808 <xs:complexType name="tOperation">
4418 809   <xs:complexContent>
4419 810     <xs:extension base="tExtensibleElements">
4420 811       <xs:sequence>
4421 812         <xs:element name="InputParameters" minOccurs="0">
4422 813           <xs:complexType>
4423 814             <xs:sequence>
4424 815               <xs:element name="InputParameter" type="tParameter"
4425 816                 minOccurs="unbounded"/>
4426 817             </xs:sequence>
4427 818           </xs:complexType>
4428 819         </xs:element>
4429 820         <xs:element name="OutputParameters" minOccurs="0">
4430 821           <xs:complexType>
4431 822             <xs:sequence>
4432 823               <xs:element name="OutputParameter" type="tParameter"
4433 824                 minOccurs="unbounded"/>
4434 825             </xs:sequence>
4435 826           </xs:complexType>
4436 827         </xs:element>
4437 828       </xs:sequence>
4438 829       <xs:attribute name="name" type="xs:NCName" use="required"/>
4439 830     </xs:extension>
4440 831   </xs:complexContent>
4441 832 </xs:complexType>
4442 833
4443 834 <xs:complexType name="tExportedOperation">
4444 835   <xs:choice>
4445 836     <xs:element name="NodeOperation">
4446 837       <xs:complexType>
4447 838         <xs:attribute name="nodeRef" type="xs:IDREF" use="required"/>
4448 839         <xs:attribute name="interfaceName" type="xs:anyURI"
4449 840           use="required"/>
4450 841         <xs:attribute name="operationName" type="xs:NCName"
4451 842           use="required"/>
4452 843       </xs:complexType>
4453 844     </xs:element>
4454 845     <xs:element name="RelationshipOperation">
4455 846       <xs:complexType>
4456 847         <xs:attribute name="relationshipRef" type="xs:IDREF"
4457 848           use="required"/>
4458 849         <xs:attribute name="interfaceName" type="xs:anyURI"
4459 850           use="required"/>
4460 851         <xs:attribute name="operationName" type="xs:NCName"
4461 852           use="required"/>
4462 853       </xs:complexType>
4463 854     </xs:element>
4464 855     <xs:element name="Plan">
4465 856       <xs:complexType>
4466 857         <xs:attribute name="planRef" type="xs:IDREF" use="required"/>
4467 858       </xs:complexType>
4468 859     </xs:element>
4469 860   </xs:choice>
4470 861   <xs:attribute name="name" type="xs:NCName" use="required"/>
4471 862 </xs:complexType>
4472 863
4473 864 <xs:complexType name="tCondition">

```

```

4474 865     <xs:sequence>
4475 866     <xs:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
4476 867     </xs:sequence>
4477 868     <xs:attribute name="expressionLanguage" type="xs:anyURI"
4478 869         use="required"/>
4479 870 </xs:complexType>
4480 871
4481 872 <xs:complexType name="tTopologyElementInstanceStates">
4482 873     <xs:sequence>
4483 874         <xs:element name="InstanceState" maxOccurs="unbounded">
4484 875             <xs:complexType>
4485 876                 <xs:attribute name="state" type="xs:anyURI" use="required"/>
4486 877             </xs:complexType>
4487 878         </xs:element>
4488 879     </xs:sequence>
4489 880 </xs:complexType>
4490 881
4491 882 <xs:complexType name="tArtifactReference">
4492 883     <xs:choice minOccurs="0" maxOccurs="unbounded">
4493 884         <xs:element name="Include">
4494 885             <xs:complexType>
4495 886                 <xs:attribute name="pattern" type="xs:string" use="required"/>
4496 887             </xs:complexType>
4497 888         </xs:element>
4498 889         <xs:element name="Exclude">
4499 890             <xs:complexType>
4500 891                 <xs:attribute name="pattern" type="xs:string" use="required"/>
4501 892             </xs:complexType>
4502 893         </xs:element>
4503 894     </xs:choice>
4504 895     <xs:attribute name="reference" type="xs:anyURI" use="required"/>
4505 896 </xs:complexType>
4506 897
4507 898 <xs:complexType name="tRequiredContainerFeatures">
4508 899     <xs:sequence>
4509 900         <xs:element name="RequiredContainerFeature"
4510 901             type="tRequiredContainerFeature" maxOccurs="unbounded"/>
4511 902     </xs:sequence>
4512 903 </xs:complexType>
4513 904
4514 905 <xs:complexType name="tRequiredContainerFeature">
4515 906     <xs:attribute name="feature" type="xs:anyURI" use="required"/>
4516 907 </xs:complexType>
4517 908
4518 909 <xs:simpleType name="tBoolean">
4519 910     <xs:restriction base="xs:string">
4520 911         <xs:enumeration value="yes"/>
4521 912         <xs:enumeration value="no"/>
4522 913     </xs:restriction>
4523 914 </xs:simpleType>
4524 915
4525 916 <xs:simpleType name="importedURI">
4526 917     <xs:restriction base="xs:anyURI"/>
4527 918 </xs:simpleType>
4528 919
4529 920 </xs:schema>

```

4530

Appendix E. Sample

4531

This appendix contains the full sample used in this specification.

4532

E.1 Sample Service Topology Definition

4533

```
01 <Definitions name="MyServiceTemplateDefinition"
4534 02     targetNamespace="http://www.example.com/sample">
4535 03     <Types>
4536 04         <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
4537 05             elementFormDefault="qualified"
4538 06             attributeFormDefault="unqualified">
4539 07             <xs:element name="ApplicationProperties">
4540 08                 <xs:complexType>
4541 09                     <xs:sequence>
4542 10                         <xs:element name="Owner" type="xs:string"/>
4543 11                         <xs:element name="InstanceName" type="xs:string"/>
4544 12                         <xs:element name="AccountID" type="xs:string"/>
4545 13                     </xs:sequence>
4546 14                 </xs:complexType>
4547 15             </xs:element>
4548 16             <xs:element name="AppServerProperties">
4549 17                 <xs:complexType>
4550 18                     <xs:sequence>
4551 19                         <element name="HostName" type="xs:string"/>
4552 20                         <element name="IPAddress" type="xs:string"/>
4553 21                         <element name="HeapSize" type="xs:positiveInteger"/>
4554 22                         <element name="SoapPort" type="xs:positiveInteger"/>
4555 23                     </xs:sequence>
4556 24                 </xs:complexType>
4557 25             </xs:element>
4558 26         </xs:schema>
4559 27     </Types>
4560 28
4561 29     <ServiceTemplate id="MyServiceTemplate">
4562 30
4563 31         <Tags>
4564 32             <Tag name="author" value="someone@example.com"/>
4565 33         </Tags>
4566 34
4567 35         <TopologyTemplate id="SampleApplication">
4568 36
4569 37             <NodeTemplate id="MyApplication"
4570 38                 name="My Application"
4571 39                 nodeType="abc:Application">
4572 40                 <Properties>
4573 41                     <ApplicationProperties>
4574 42                         <Owner>Frank</Owner>
4575 43                         <InstanceName>Thomas' favorite application</InstanceName>
4576 44                     </ApplicationProperties>
4577 45                 </Properties>
4578 46             </NodeTemplate>
4579 47
4580 48             <NodeTemplate id="MyAppServer"
4581 49                 name="My Application Server"
```

```

4582 50         nodeType="abc:ApplicationServer"
4583 51         minInstances="0"
4584 52         maxInstances="unbounded"/>
4585 53
4586 54     <RelationshipTemplate id="MyDeploymentRelationship"
4587 55         relationshipType="abc:deployedOn">
4588 56         <SourceElement id="MyApplication"/>
4589 57         <TargetElement id="MyAppServer"/>
4590 58     </RelationshipTemplate>
4591 59
4592 60 </TopologyTemplate>
4593 61
4594 62 <Plans>
4595 63     <Plan id="DeployApplication"
4596 64         name="Sample Application Build Plan"
4597 65         planType="http://docs.oasis-
4598 66             open.org/tosca/ns/2011/12/PlanTypes/BuildPlan"
4599 67         planLanguage="http://www.omg.org/spec/BPMN/20100524/MODEL">
4600 68
4601 69         <Precondition expressionLanguage="www.example.com/text"> ?
4602 70             Run only if funding is available
4603 71         </Precondition>
4604 72
4605 73         <PlanModel>
4606 74             <process name="DeployNewApplication" id="p1">
4607 75                 <documentation>This process deploys a new instance of the
4608 76                     sample application.
4609 77                 </documentation>
4610 78
4611 79                 <task id="t1" name="CreateAccount"/>
4612 80
4613 81                 <task id="t2" name="AcquireNetworkAddresses"
4614 82                     isSequential="false"
4615 83                     loopDataInput="t2Input.LoopCounter"/>
4616 84                 <documentation>Assumption: t2 gets data of type "input"
4617 85                     as input and this data has a field names "LoopCounter"
4618 86                     that contains the actual multiplicity of the task.
4619 87                 </documentation>
4620 88
4621 89                 <task id="t3" name="DeployApplicationServer"
4622 90                     isSequential="false"
4623 91                     loopDataInput="t3Input.LoopCounter"/>
4624 92
4625 93                 <task id="t4" name="DeployApplication"
4626 94                     isSequential="false"
4627 95                     loopDataInput="t4Input.LoopCounter"/>
4628 96
4629 97                 <sequenceFlow id="s1" targetRef="t2" sourceRef="t1"/>
4630 98                 <sequenceFlow id="s2" targetRef="t3" sourceRef="t2"/>
4631 99                 <sequenceFlow id="s3" targetRef="t4" sourceRef="t3"/>
4632 100             </process>
4633 101         </PlanModel>
4634 102     </Plan>
4635 103
4636 104     <Plan id="RemoveApplication"
4637 105         planType="http://docs.oasis-
4638 106             open.org/tosca/ns/2011/12/PlanTypes/TerminationPlan"
4639 107         planLanguage="http://docs.oasis-

```

```

4640 108         open.org/wsbpel/2.0/process/executable">
4641 109         <PlanModelReference reference="prj:RemoveApp"/>
4642 110     </Plan>
4643 111 </Plans>
4644 112
4645 113 </ServiceTemplate>
4646 114
4647 115 <NodeType name="Application">
4648 116     <documentation xml:lang="EN">
4649 117         A reusable definition of a node type representing an
4650 118         application that can be deployed on application servers.
4651 119     </documentation>
4652 120     <NodeTypeProperties element="ApplicationProperties"/>
4653 121     <InstanceStates>
4654 122         <InstanceState state="http://www.example.com/started"/>
4655 123         <InstanceState state="http://www.example.com/stopped"/>
4656 124     </InstanceStates>
4657 125     <Interfaces>
4658 126         <Interface name="DeploymentInterface">
4659 127             <Operation name="DeployApplication">
4660 128                 <InputParameters>
4661 129                     <InputParamter name="InstanceName"
4662 130                         type="xs:string"/>
4663 131                     <InputParamter name="AppServerHostname"
4664 132                         type="xs:string"/>
4665 133                     <InputParamter name="ContextRoot"
4666 134                         type="xs:string"/>
4667 135                 </InputParameters>
4668 136             </Operation>
4669 137         </Interface>
4670 138     </Interfaces>
4671 139 </NodeType>
4672 140
4673 141 <NodeType name="ApplicationServer"
4674 142     targetNamespace="http://www.example.com/sample">
4675 143     <NodeTypeProperties element="AppServerProperties"/>
4676 144     <Interfaces>
4677 145         <Interface name="MyAppServerInterface">
4678 146             <Operation name="AcquireNetworkAddress"/>
4679 147             <Operation name="DeployApplicationServer"/>
4680 148         </Interface>
4681 149     </Interfaces>
4682 150 </NodeType>
4683 151
4684 152 <RelationshipType name="deployedOn">
4685 153     <documentation xml:lang="EN">
4686 154         A reusable definition of relation that expresses deployment of
4687 155         an artifact on a hosting environment.
4688 156     </documentation>
4689 157 </RelationshipType>
4690 158
4691 159 </Definitions>

```

Appendix F. Revision History

Revision	Date	Editor	Changes Made
wd-01	2012-01-26	Thomas Spatzier	Changes for JIRA Issue TOSCA-1: Initial working draft based on input spec delivered to TOSCA TC. Copied all content from input spec and just changed namespace. Added line numbers to whole document.
wd-02	2012-02-23	Thomas Spatzier	Changes for JIRA Issue TOSCA-6: Reviewed and adapted normative statement keywords according to RFC2119.
wd-03	2012-03-06	Arvind Srinivasan, Thomas Spatzier	Changes for JIRA Issue TOSCA-10: Marked all occurrences of keywords from the TOSCA language (element and attribute names) in Courier New font.
wd-04	2012-03-22	Thomas Spatzier	Changes for JIRA Issue TOSCA-4: Changed definition of <code>NodeType Interfaces</code> element; adapted text and examples
wd-05	2012-03-30	Thomas Spatzier	Changes for JIRA Issue TOSCA-5: Changed definition of <code>NodeTemplate</code> to include <code>ImplementationArtifact</code> element; adapted text Added Acknowledgements section in Appendix
wd-06	2012-05-03	Thomas Spatzier, Derek Palma	Changes for JIRA Issue TOSCA-15: Added clarifying section about artifacts (see section 3.2); Implemented editorial changes according to OASIS staff recommendations; updated Acknowledgements section
wd-07	2012-06-15	Thomas Spatzier	Changes for JIRA Issue TOSCA-20: Added <code>abstract</code> attribute to <code>NodeType</code> for sub-issue 2; Added <code>final</code> attribute to <code>NodeType</code> for sub-issue 4; Added explanatory text on Node Type properties for sub-issue 8
wd-08	2012-06-29	Thomas Spatzier, Derek Palma	Changes for JIRA Issue TOSCA-23: Added interfaces and introduced inheritance for <code>RelationshipType</code> ; based on wd-07 Added reference to XML element and attribute naming scheme used in this spec

wd-09	2012-07-16	Thomas Spatzier	Changes for JIRA Issue TOSCA-17: Specifies the format of a CSAR file; Explained CSAR concept in the corresponding section.
wd-10	2012-07-30	Thomas Spatzier, Derek Palma	Changes for JIRA Issue TOSCA-18 and related issues: Introduced concept of Requirements and Capabilities; Restructuring of some paragraphs to improve readability
wd-11	2012-08-25	Thomas Spatzier, Derek Palma	Changes for JIRA Issue TOSCA-13: Clarifying rewording of introduction Changes for JIRA Issue TOSCA-38: Add <code>substitutableNodeType</code> attribute and <code>BoundaryDefinitions</code> to Service Template to allow for Service Template composition. Changes for JIRA Issue TOSCA-41: Add Tags to Service Template as simple means for Service Template versioning; Changes for JIRA Issue TOSCA-47: Use <code>name</code> and <code>targetNamespace</code> for uniquely identifying TOSCA types; Changes for JIRA Issue TOSCA-48 (partly): implement notational conventions in pseudo schemas
wd-12	2012-09-29	Thomas Spatzier, Derek Palma	Editorial changes for TOSCA-10: Formatting corrections according to OASIS feedback Changes for JIRA Issue TOSCA-28,29: Added Node Type Implementation (with deployment artifacts and implementation artifacts) that points to a Node Type it realizes; added Relationship Type Implementation analogously for Relationship Types Changes for JIRA Issue TOSCA-38: Added <code>Interfaces</code> to <code>BoundaryDefinitions</code> . Changes for JIRA Issue TOSCA-52: Removal of <code>GroupTemplate</code> Changes for JIRA Issue TOSCA-54: Clarifying rewording in section 3.5 Changes for JIRA Issue TOSCA-56: Clarifying rewording in section 2.8.2 Changes for JIRA Issue TOSCA-58: Clarifying rewording in section 13 Updated roster as of 2012-09-29

wd-13	2012-10-26	Thomas Spatzier, Derek Palma	<p>Changes for JIRA Issue TOSCA-10: More fixes to formatting and references in document according to OASIS staff comments</p> <p>Changes for JIRA Issues TOSCA-36/37: Added <code>PolicyType</code> and <code>PolicyTemplate</code> elements to allow for reusable definitions of policies.</p> <p>Changes for JIRA Issue TOSCA-57: Restructure TOSCA schema to allow for better modular definitions and separation of concerns.</p> <p>Changes for JIRA Issue TOSCA-59: Rewording to clarify overriding of deployment artifacts of Node Templates.</p> <p>Some additional minor changes in wording.</p> <p>Changes for JIRA Issue TOSCA-63: clarifying rewording</p>
wd-14	2012-11-19	Thomas Spatzier	<p>Changes for JIRA Issue TOSCA-76: Add <code>Entry-Definitions</code> property for <code>TOSCA.meta</code> file.</p> <p>Multiple general editorial fixes: Typos, namespaces and MIME types used in examples</p> <p>Fixed schema problems in <code>tPolicyTemplate</code> and <code>tPolicyType</code></p> <p>Added text to Conformance section.</p>
wd-15	2013-02-26	Thomas Spatzier	<p>Changes for JIRA Issue TOSCA-79: Handle public review comments: fixes of typos and other non-material changes like inconsistencies between the specification document and the schema in this document and the TOSCA schema</p>

4694