



Topology and Orchestration Specification for Cloud Applications Version 1.0

**Committee Specification ~~Draft 06 /~~
~~Public Review Draft 0101~~**

~~29 November 2012~~

18 March 2013

Specification URIs

This version:

<http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-csprd01.pdf> (Authoritative)
<http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-csprd01.html>
<http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-csprd01.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>

Previous version:

~~(Authoritative)~~

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), Vnomnic
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/cs01/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. ~~29 November 2012-18 March 2013~~. OASIS Committee Specification ~~Draft-06 / Public Review Draft-01~~. <http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html>.

Notices

Copyright © OASIS Open 201~~2~~3. 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.....	94
Appendix E. Sample	111

E.1 Sample Service Topology Definition	111
Appendix F. Revision History	114

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

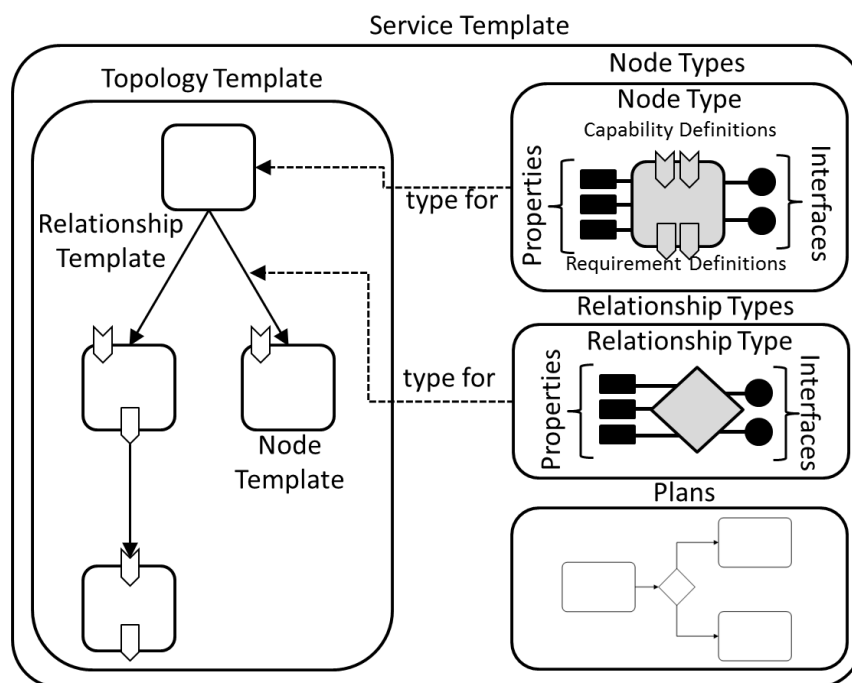
98 The main concepts behind TOSCA are described and some usage patterns of Service Templates are
99 sketched.

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.

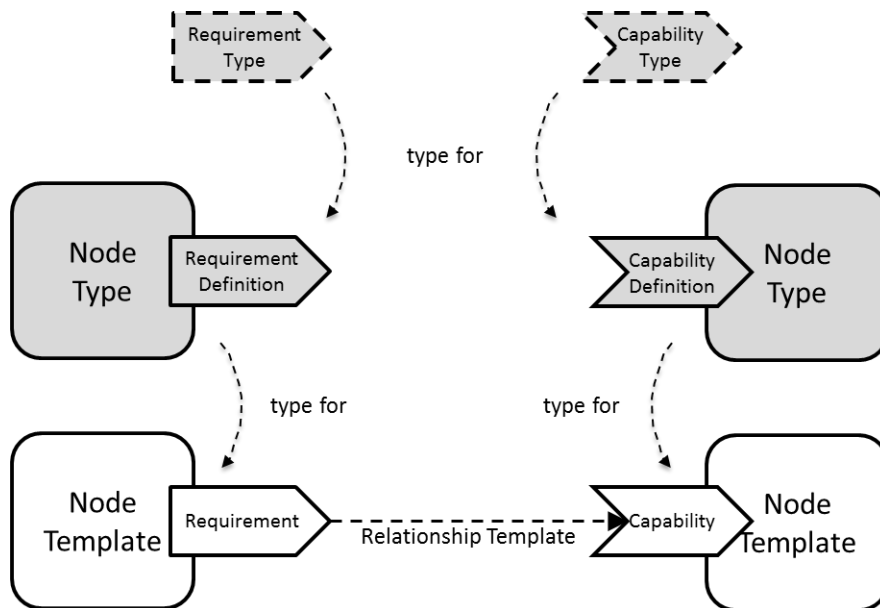
237 The instantiation of a node type can require providing deployment artifacts in the target managed
238 environment. For this purpose, a TOSCA container supports a set of types of deployment artifacts that it
239 can process. A service template that contains (implementation or deployment) artifacts 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

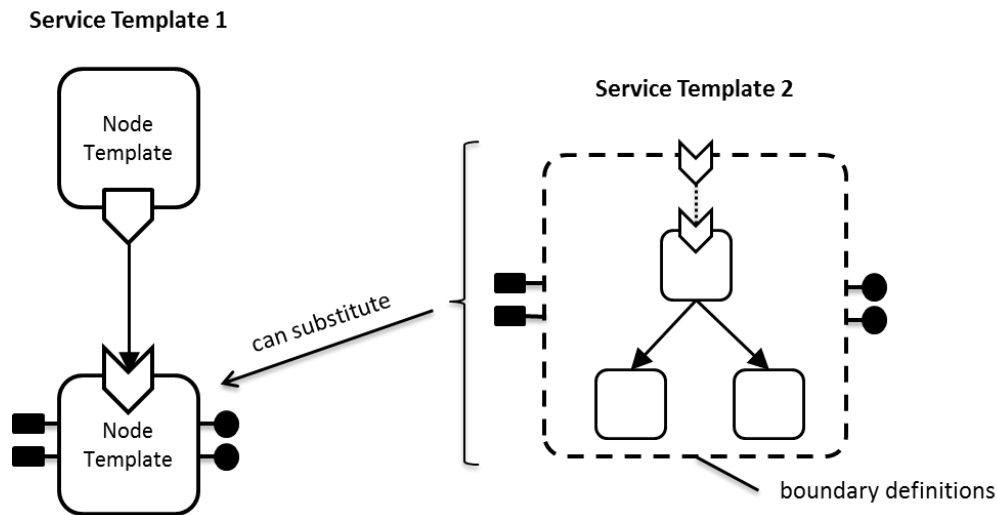
255

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.

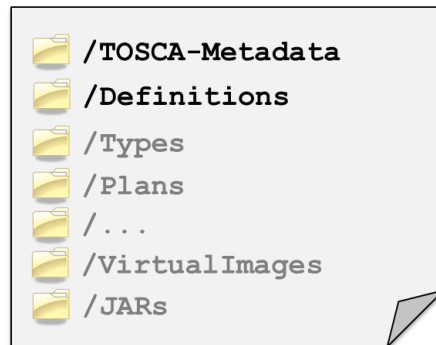
292 A Node Template can be associated with a set of Policies collectively expressing the non-functional
293 behavior or quality-of-services that each instance of the Node Template will expose. Each Policy specifies
294 the actual properties of the non-functional behavior, like the concrete payment information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



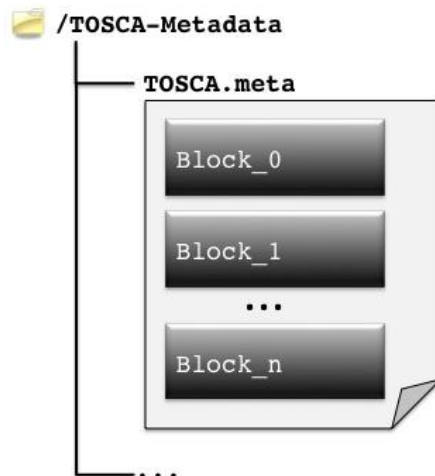
312

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.

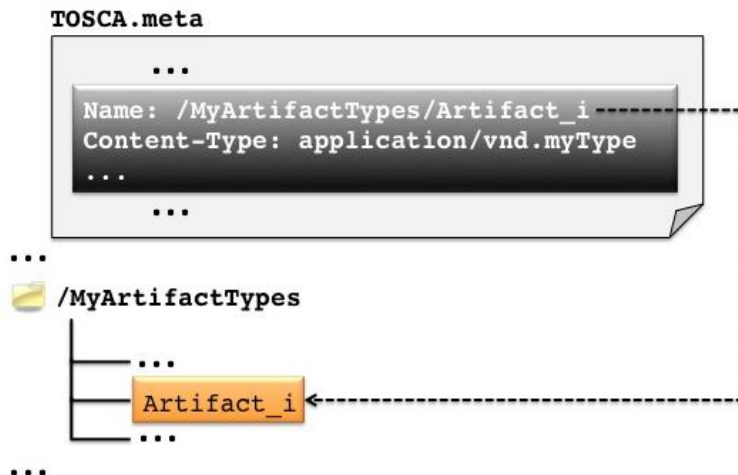
318 Each CSAR MUST contain a subdirectory called *TOSCA-Metadata*. This subdirectory MUST contain a
319 so-called *TOSCA meta file*. This file is named *TOSCA* and has the file extension *.meta*. It represents
320 metadata of the other files in the CSAR. This metadata is given in the format of name/value 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*.



323
324

Figure 5: Structure of the TOSCA Meta File

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:IDREFstring"/> +  
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          <PreEcondition expressionLanguage="xs:anyURI">
717 154              condition
718 155          </PreEcondition> ?
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 • `targetObjectPropertyRef`: This attribute identifies a property of
786 the target object by means of an XPath expression to be evaluated on
787 the XML fragment defining the target object's properties.

789 Note: If a Service Template property is mapped to a property of a
790 component within the Service Template, the XML schema type of the
791 Service Template property and the mapped property MUST be
792 compatible.

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.
- 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 Note: If the property affected by the property constraint is mapped to a property
805 of a component within the Service Template, the property constraint SHOULD be
806 compatible with any property constraint defined for the mapped property.
 - 807 ▪ `constraintType`: This attribute specifies the type of constraint by means of a
808 URI, which defines both the semantic meaning of the constraint as well as the
809 format of the content.
 - 810 ▪ The body of the `PropertyConstraint` element provides the actual
811 constraint.
812 Note: The body MAY be empty in case the `constraintType` URI already
813 specifies the constraint appropriately. For example, a "read-only" constraint could
814 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 The policyType attribute specifies the artifact type specific content of the
849 Policy element body and indicates the type of Policy Template referenced by
850 the Policy via the policyRef attribute.
- 851 ▪ policyRef: The QName value of this OPTIONAL attribute references a Policy
852 Template that is associated to the Service Template. This Policy Template can
853 be defined in the same TOSCA Definitions document, or it can be defined in a
854 separate document that is imported into the current Definitions document. The
855 type of Policy Template referenced by the policyRef attribute MUST be the
856 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 policyRef attribute specifies the artifact type specific content of the
866 Policy element body and indicates the type of Policy Template referenced by
867 the Policy via the policyRef attribute.
- 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.

- 999
- 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.

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 the reference property of the node template under definition has to be

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 `RelationshipTemplate`.

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 `RelationshipTemplate`.

1166
1167 If the `RelationshipType` 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 `RelationshipType`.

1172
1173 In the case where a Node Type is defined as valid source in the
1174 `RelationshipType` 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 `RelationshipType` 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 `RelationshipTemplate`.

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 `RelationshipTemplate`.

1187
1188 If the `RelationshipType` 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 `RelationshipType`.

1193
1194 In case a Node Type is defined as valid target in the `RelationshipType`
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.
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.

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.

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.
 - `PreEcondition`: 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.

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.
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                     <InputParameter name="ProjectName"
1581 23                         type="xs:string"/>
1582 24                     <InputParameter name="Owner"
1583 25                         type="xs:string"/>
1584 26                     <InputParameter 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 name="xs:string"  
1635 1817     interfaceName="xs:NCName | xs:anyURI"?  
1636 1918     operationName="xs:NCName"?  
1637 2019     artifactType="xs:QName"  
1638 2120     artifactRef="xs:QName"?>  
1639 2221     artifact specific content ?  
1640 2322     <ImplementationArtifact> +  
1641 2423 </ImplementationArtifacts> ?  
1642 2524  
1643 2625 <DeploymentArtifacts>  
1644 2726     <DeploymentArtifact name="xs:string" artifactType="xs:QName"  
1645 2827         artifactRef="xs:QName"?>  
1646 2928     artifact specific content ?  
1647 3029     <DeploymentArtifact> +  
1648 3130 </DeploymentArtifacts> ?  
1649 3231  
1650 3332 </NodeTypeImplementation>
```

1651 7.2 Properties

1652 The `NodeTypeImplementation` element has the following properties:

- 1653 • `name`: This attribute specifies the name or identifier of the Node Type Implementation, which
1654 MUST be unique within the target namespace.
- 1655 • `targetNamespace`: This OPTIONAL attribute specifies the target namespace to which the
1656 definition of the Node Type Implementation will be added. If not specified, the Node Type
1657 Implementation will be added to the target namespace of the enclosing Definitions document.
- 1658 • `nodeType`: The QName value of this attribute specifies the Node Type implemented by this
1659 Node Type Implementation.
- 1660 • `abstract`: This OPTIONAL attribute specifies that this Node Type Implementation cannot be
1661 used directly as an implementation for the Node Type specified in the `nodeType` attribute.
1662

1663 For example, a Node Type implementer might decide to deliver only part of the implementation of
1664 a specific Node Type (i.e. for only some operations) for re-use purposes and require the
1665 implementation for specific operations to be delivered in a more concrete, derived Node Type
1666 Implementation.

1667
1668 Note: an abstract Node Type Implementation MUST NOT be declared as final.

- 1669 • `final`: This OPTIONAL attribute specifies that other Node Type Implementations MUST NOT
1670 be derived from this Node Type Implementation.
1671

1672 Note: a final Node Type Implementation MUST NOT be declared as abstract.

- 1673 • `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by
1674 the author to describe the Node Type Implementation. Each tag is defined by a separate, nested
1675 Tag element.

1676 The Tag element has the following properties:

- 1677 ○ `name`: This attribute specifies the name of the tag.
- 1678 ○ `value`: This attribute specifies the value of the tag.
1679

1680 Note: The name/value pairs defined in tags have no normative interpretation.

- 1681 • `DerivedFrom`: This is an OPTIONAL reference to another Node Type Implementation from
1682 which this Node Type Implementation derives. See section 7.3 Derivation Rules **Error! Reference
1683 source not found.** for details.

1684 The `DerivedFrom` element has the following properties:

- 1685 ○ `nodeTypeImplementationRef`: The QName specifies the Node Type
1686 Implementation from which this Node Type Implementation derives.
- 1687 • `RequiredContainerFeatures`: An implementation of a Node Type might depend on
1688 certain features of the environment it is executed in, such as specific (potentially proprietary) APIs
1689 of the TOSCA container. For example, an implementation to deploy a virtual machine based on
1690 an image could require access to some API provided by a public cloud, while another
1691 implementation could require an API of a vendor-specific virtual image library. Thus, the contents
1692 of the `RequiredContainerFeatures` element provide “hints” to the TOSCA container
1693 allowing it to select the appropriate Node Type Implementation if multiple alternatives are
1694 provided.

1695 Each such dependency is defined by a separate `RequiredContainerFeature` element.

1696 The `RequiredContainerFeature` element has the following properties:

- 1697 ○ `feature`: The value of this attribute is a URI that denotes the corresponding needed
1698 feature of the environment.

1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750

- **ImplementationArtifacts**: This element specifies a set of implementation artifacts for interfaces or operations of a Node Type.
The **ImplementationArtifacts** element has the following properties:
 - **ImplementationArtifact**: This element specifies one implementation artifact of an interface or an operation.

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.

The **ImplementationArtifact** element has the following properties:
 - **name**: This attribute specifies the name of the artifact, which SHOULD be unique within the scope of the encompassing Node Type Implementation.
 - **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**.
 - **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.
 - **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 **ImplementationArtifact** element body and indicates the type of **Artifact Template** referenced by the **Implementation Artifact** via the **artifactRef** attribute.
 - **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.

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.
- **DeploymentArtifacts**: This element specifies a set of deployment artifacts relevant for materializing instances of nodes of the Node Type being implemented.
The **DeploymentArtifacts** element has the following properties:
 - **DeploymentArtifact**: This element specifies one deployment artifact.

1751 Note: Multiple deployment artifacts MAY be defined in a Node Type Implementation. One
1752 reason could be that multiple artifacts (maybe of different types) are needed to
1753 materialize a node as a whole. Another reason could be that alternative artifacts are
1754 provided for use in different contexts (e.g. different installables of a software for use in
1755 different operating systems).

1756
1757 The `DeploymentArtifact` element has the following properties:

- 1758 ▪ `name`: This attribute specifies the name of the artifact, which SHOULD be unique
1759 within the scope of the encompassing Node Type Implementation.
- 1760 ▪ `artifactType`: This attribute specifies the type of this artifact. The QName
1761 value of this attribute SHOULD correspond to the QName of an
1762 `ArtifactType` defined in the same Definitions document or in an imported
1763 document.

1764
1765 The `artifactType` attribute specifies the artifact type specific content of the
1766 `DeploymentArtifact` element body and indicates the type of Artifact
1767 Template referenced by the Deployment Artifact via the `artifactRef`
1768 attribute.

- 1769 ▪ `artifactRef`: This OPTIONAL attribute contains a QName that identifies an
1770 Artifact Template to be used as deployment artifact. This Artifact Template can
1771 be defined in the same Definitions document or in a separate, imported
1772 document.

1773 The type of Artifact Template referenced by the `artifactRef` attribute MUST
1774 be the same type or a sub-type of the type specified in the `artifactType`
1775 attribute.

1776
1777 Note: if no Artifact Template is referenced, the artifact type specific content of the
1778 `DeploymentArtifact` element alone is assumed to represent the actual
1779 artifact. For example, the contents of a simple config file could be defined in
1780 place within the `DeploymentArtifact` element.

1781 7.3 Derivation Rules

1782 The following rules on combining definitions based on `DerivedFrom` apply:

- 1783 • **Implementation Artifacts:** The set of implementation artifacts of a Node Type Implementation
1784 consists of the set union of implementation artifacts defined by the Node Type Implementation
1785 itself and the implementation artifacts defined by any Node Type Implementation the Node Type
1786 Implementation is derived from.
1787 An implementation artifact defined by a Node Type Implementation overrides an implementation
1788 artifact having the same interface name and operation name of a Node Type Implementation the
1789 Node Type Implementation is derived from.
1790 If an implementation artifact defined in a Node Type Implementation specifies only an interface
1791 name, it substitutes implementation artifacts having the same interface name (with or without an
1792 operation name defined) of any Node Type Implementation the Node Type Implementation is
1793 derived from. In this case, the implementation of a complete interface of a Node Type is
1794 overridden.
1795 If an implementation artifact defined in a Node Type Implementation neither defines an interface
1796 name nor an operation name, it overrides all implementation artifacts of any Node Type
1797 Implementation the Node Type Implementation is derived from. In this case, the complete
1798 implementation of a Node Type is overridden.

- 1799
- 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.
- 1800
1801
1802
1803
1804

1805 7.4 Example

1806 The following example defines the Node Type Implementation “MyDBMSImplementation”. This is an
1807 implementation of a Node Type “DBMS”.

```
1808 01 <Definitions id="MyImpls" name="My Implementations"  
1809 02   targetNamespace="http://www.example.com/SampleImplementations"  
1810 03   xmlns:bn="http://www.example.com/BaseNodeTypes"  
1811 04   xmlns:ba="http://www.example.com/BaseArtifactTypes"  
1812 05   xmlns:sa="http://www.example.com/SampleArtifacts">  
1813 06  
1814 07   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"  
1815 08     namespace="http://www.example.com/BaseArtifactTypes"/>  
1816 09  
1817 10   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"  
1818 11     namespace="http://www.example.com/BaseNodeTypes"/>  
1819 12  
1820 13   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"  
1821 14     namespace="http://www.example.com/SampleArtifacts"/>  
1822 15  
1823 16   <NodeTypeImplementation name="MyDBMSImplementation"  
1824 17     nodeType="bn:DBMS">  
1825 18  
1826 19     <ImplementationArtifacts>  
1827 20       <ImplementationArtifact name="MyDBMSManagement"  
1828 21         interfaceName="MgmtInterface"  
1829 22         artifactType="ba:WARFile"  
1830 23         artifactRef="sa:MyMgmtWebApp">  
1831 24       </ImplementationArtifact>  
1832 25     </ImplementationArtifacts>  
1833 26  
1834 27     <DeploymentArtifacts>  
1835 28       <DeploymentArtifact name="MyDBMS"  
1836 29         artifactType="ba:ZipFile"  
1837 30         artifactRef="sa:MyInstallable">  
1838 31       </DeploymentArtifact>  
1839 32     </DeploymentArtifacts>  
1840 33   </NodeTypeImplementation>  
1841 34 </Definitions>  
1842  
1843
```

1844 The Node Type Implementation contains the “MyDBMSManagement” implementation artifact, which is an
1845 artifact for the “MgmtInterface” Interface that has been defined for the “DBMS” base Node Type. The type
1846 of this artifact is a “WARFile” that has been defined as base Artifact Type. The implementation artifact
1847 refers to the “MyMgmtWebApp” Artifact Template that has been defined before.

1848 The Node Type Implementation further contains the “MyDBMS” deployment artifact, which is a software
1849 installable used for instantiating the “DBMS” Node Type. This software installable is a “ZipFile” that has
1850 been separately defined as the “MyInstallable” Artifact Template before.

1851

8 Relationship Types

1852 This chapter specifies how *Relationship Types* are defined. A Relationship Type is a reusable entity that
1853 defines the type of one or more Relationship Templates between Node Templates. As such, a
1854 Relationship Type can define the structure of observable properties via a *Properties Definition*, i.e. the
1855 names, data types and allowed values the properties defined in Relationship Templates using a
1856 Relationship Type or instances of such Relationship Templates can have.

1857 The operations that can be performed on (an instance of) a corresponding Relationship Template are
1858 defined by the *Interfaces* of the Relationship Type. Furthermore, a Relationship Type defines the potential
1859 states an instance of it might reveal at runtime.

1860 A Relationship Type can inherit the definitions defined in another Relationship Type by means of the
1861 *DerivedFrom* element. Relationship Types might be declared as abstract, meaning that they cannot be
1862 instantiated. The purpose of such abstract Relationship Types is to provide common properties and
1863 behavior for re-use in specialized, derived Relationship Types. Relationship Types might also be declared
1864 as final, meaning that they cannot be derived by other Relationship Types.

1865 8.1 XML Syntax

1866 The following pseudo schema defines the XML syntax of Relationship Types:

```
1867 01 <RelationshipType name="xs:NCName"
1868 02           targetNamespace="xs:anyURI"?
1869 03           abstract="yes|no"?
1870 04           final="yes|no"?> +
1871 05
1872 06   <Tags>
1873 07     <Tag name="xs:string" value="xs:string"/> +
1874 08   </Tags> ?
1875 09
1876 10   <DerivedFrom typeRef="xs:QName"/> ?
1877 11
1878 12   <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
1879 13
1880 14   <InstanceStates>
1881 15     <InstanceState state="xs:anyURI"> +
1882 16   </InstanceStates> ?
1883 17
1884 18   <SourceInterfaces>
1885 19     <Interface name="xs:NCName | xs:anyURI">
1886 20       ...
1887 21     </Interface> +
1888 22   </SourceInterfaces> ?
1889 23
1890 24   <TargetInterfaces>
1891 25     <Interface name="xs:NCName | xs:anyURI">
1892 26       ...
1893 27     </Interface> +
1894 28   </TargetInterfaces> ?
1895 29
1896 30   <ValidSource typeRef="xs:QName"/> ?
1897 31
1898 32   <ValidTarget typeRef="xs:QName"/> ?
1899 33
1900 34 </RelationshipType>
```

1901 8.2 Properties

1902 The `RelationshipType` element has the following properties:

- 1903 • `name`: This attribute specifies the name or identifier of the Relationship Type, which MUST be
1904 unique within the target namespace.
- 1905 • `targetNamespace`: This OPTIONAL attribute specifies the target namespace to which the
1906 definition of the Relationship Type will be added. If not specified, the Relationship Type definition
1907 will be added to the target namespace of the enclosing Definitions document.
- 1908 • `abstract`: This OPTIONAL attribute specifies that no instances can be created from
1909 Relationship Templates that use this Relationship Type as their type.
1910

1911 As a consequence, the corresponding abstract Relationship Type referenced by any Relationship
1912 Template has to be substituted by a Relationship Type derived from the abstract Relationship
1913 Type at the latest during the instantiation time of a Relationship Template.
1914

1915 Note: an abstract Relationship Type MUST NOT be declared as final.

- 1916 • `final`: This OPTIONAL attribute specifies that other Relationship Types MUST NOT be derived
1917 from this Relationship Type.
1918

1919 Note: a final Relationship Type MUST NOT be declared as abstract.

- 1920 • `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by
1921 the author to describe the Relationship Type. Each tag is defined by a separate, nested `Tag`
1922 element.
1923

1923 The `Tag` element has the following properties:

- 1924 ○ `name`: This attribute specifies the name of the tag.
- 1925 ○ `value`: This attribute specifies the value of the tag.
1926

1927 Note: The name/value pairs defined in tags have no normative interpretation.

- 1928 • `DerivedFrom`: This is an OPTIONAL reference to another Relationship Type from which this
1929 Relationship Type is derived. Conflicting definitions are resolved by the rule that local new
1930 definitions always override derived definitions. See section 8.3 Derivation Rules for details.
1931

1931 The `DerivedFrom` element has the following properties:

- 1932 ○ `typeRef`: The QName specifies the Relationship Type from which this Relationship
1933 Type derives its definitions.

- 1934 • `PropertiesDefinition`: This element specifies the structure of the observable properties
1935 of the Relationship Type, such as its configuration and state, by means of XML schema.
1936

1936 The `PropertiesDefinition` element has one but not both of the following properties:

- 1937 ○ `element`: This attribute provides the QName of an XML element defining the structure
1938 of the Relationship Type Properties.
- 1939 ○ `type`: This attribute provides the QName of an XML (complex) type defining the
1940 structure of the Relationship Type Properties.

- 1941 • `InstanceStates`: This OPTIONAL element lists the set of states an instance of this
1942 Relationship Type can occupy at runtime. Those states are defined in nested `InstanceState`
1943 elements.
1944

1944 The `InstanceState` element has the following nested properties:

- 1945 ○ `state`: This attribute specifies a URI that identifies a potential state.

- 1946 • `SourceInterfaces`: This OPTIONAL element contains definitions of manageability interfaces
1947 that can be performed on the source of a relationship of this Relationship Type to actually
1948 establish the relationship between the source and the target in the deployed service.

1949 Those interface definitions are contained in nested `Interface` elements, the content of which
1950 is that described for Node Type interfaces (see section 6.2).

1951 • `TargetInterfaces`: This OPTIONAL element contains definitions of manageability interfaces
1952 that can be performed on the target of a relationship of this Relationship Type to actually
1953 establish the relationship between the source and the target in the deployed service.

1954 Those interface definitions are contained in nested `Interface` elements, the content of which
1955 is that described for Node Type interfaces (see section 6.2).

1956 • `ValidSource`: This OPTIONAL element specifies the type of object that is allowed as a valid
1957 origin for relationships defined using the Relationship Type under definition. If not specified, any
1958 Node Type is allowed to be the origin of the relationship.

1959 The `ValidSource` element has the following properties:

1960 ○ `typeRef`: This attribute specifies the QName of a Node Type or Requirement Type that
1961 is allowed as a valid source for relationships defined using the Relationship Type under
1962 definition. Node Types or Requirements Types derived from the specified Node Type or
1963 Requirement Type, respectively, MUST also be accepted as valid relationship source.

1964
1965 Note: If `ValidSource` specifies a Node Type, the `ValidTarget` element (if present)
1966 of the Relationship Type under definition MUST also specify a Node Type.

1967 If `ValidSource` specifies a Requirement Type, the `ValidTarget` element (if
1968 present) of the Relationship Type under definition MUST specify a Capability Type. This
1969 Capability Type MUST match the requirement defined in `ValidSource`, i.e. it MUST
1970 be of the type (or a sub-type of) the capability specified in the
1971 `requiredCapabilityType` attribute of the respective `RequirementType`
1972 definition.

1973 • `ValidTarget`: This OPTIONAL element specifies the type of object that is allowed as a valid
1974 target for relationships defined using the Relationship Type under definition. If not specified, any
1975 Node Type is allowed to be the origin of the relationship.

1976 The `ValidTarget` element has the following properties:

1977 ○ `typeRef`: This attribute specifies the QName of a Node Type or Capability Type that is
1978 allowed as a valid target for relationships defined using the Relationship Type under
1979 definition. Node Types or Capability Types derived from the specified Node Type or
1980 Capability Type, respectively, MUST also be accepted as valid targets of relationships.

1981
1982 Note: If `ValidTarget` specifies a Node Type, the `ValidSource` element (if present)
1983 of the Relationship Type under definition MUST also specify a Node Type.

1984 If `ValidTarget` specifies a Capability Type, the `ValidSource` element (if present)
1985 of the Relationship Type under definition MUST specify a Requirement Type. This
1986 Requirement Type MUST declare it requires the capability defined in `ValidTarget`,
1987 i.e. it MUST declare the type (or a super-type of) the capability in the
1988 `requiredCapabilityType` attribute of the respective `RequirementType`
1989 definition.

1990 8.3 Derivation Rules

1991 The following rules on combining definitions based on `DerivedFrom` apply:

1992 • `Relationship Type Properties`: It is assumed that the XML element (or type) representing the
1993 Relationship Type properties of the Relationship Type under definition extends the XML element
1994 (or type) of the Relationship Type properties of the Relationship Type referenced in the
1995 `DerivedFrom` element.

1996 • `Instance States`: The resulting set of instance states of the Relationship Type under definition
1997 consists of the set union of the instances states defined by the Relationship Type derived from

- 1998 and the instance states explicitly defined by the Relationship Type under definition. Instance
 1999 states with the same state attribute will be combined into a single instance state of the same
 2000 state.
- 2001 • Valid source and target: An object specified as a valid source or target, respectively, of the
 2002 Relationship Type under definition MUST be of a subtype defined as valid source or target,
 2003 respectively, of the Relationship Type derived from.
- 2004
- 2005 If the Relationship Type derived from has no valid source or target defined, the types of object
 2006 being defined in the `ValidSource` or `ValidTarget` elements of the Relationship Type
 2007 under definition are not restricted.
- 2008
- 2009 If the Relationship Type under definition has no source or target defined, only the types of objects
 2010 defined as source or target of the Relationship Type derived from are valid origins or destinations
 2011 of the Relationship Type under definition.
- 2012 • Interfaces: The set of interfaces (both source and target interfaces) of the Relationship Type
 2013 under definition consists of the set union of interfaces defined by the Relationship Type derived
 2014 from and the interfaces defined by the Relationship Type under definition.
 2015 Two interfaces of the same name will be combined into a single, derived interface with the same
 2016 name. The set of operations of the derived interface consists of the set union of operations
 2017 defined by both interfaces. An operation defined by the Relationship Type under definition
 2018 substitutes an operation with the same name of the Relationship Type derived from.

2019 8.4 Example

2020 The following example defines the Relationship Type “processDeployedOn”. The meaning of this
 2021 Relationship Type is that “a process is deployed on a hosting environment”. When the source of an
 2022 instance of a Relationship Template referring to this Relationship Type is deleted, its target is
 2023 automatically deleted as well. The Relationship Type has Relationship Type Properties defined in the
 2024 `Types` section of the same Definitions document as the “ProcessDeployedOnProperties” element. The
 2025 states an instance of this Relationship Type can be in are also listed.

```

2026 01 <RelationshipType name="processDeployedOn">
2027 02
2028 03   <RelationshipTypeProperties element="ProcessDeployedOnProperties"/>
2029 04
2030 05   <InstanceStates>
2031 06     <InstanceState state="www.example.com/successfullyDeployed"/>
2032 07     <InstanceState state="www.example.com/failed"/>
2033 08   </InstanceStates>
2034 09
2035 10 </RelationshipType>
  
```

2036

9 Relationship Type Implementations

2037 This chapter specifies how *Relationship Type Implementations* are defined. A Relationship Type
2038 Implementation represents the runnable code that implements a specific Relationship Type. It provides a
2039 collection of executables implementing the interface operations of a Relationship Type (aka
2040 implementation artifacts). The particular executables are defined as separate Artifact Templates and are
2041 referenced from the implementation artifacts of a Relationship Type Implementation.

2042 While Artifact Templates provide invariant information about an artifact – i.e. information that is context
2043 independent like the file name of the artifact – implementation artifacts can provide variant (or context
2044 specific) information, e.g. authentication data for a specific environment.

2045 Relationship Type Implementations can specify hints for a TOSCA container that enable proper selection
2046 of an implementation that fits into a particular environment by means of Required Container Features
2047 definitions.

2048 Note that there MAY be Relationship Types that do not define any interface operations, i.e. that also do
2049 not require any implementation artifacts. In such cases, no Relationship Type Implementation is needed
2050 but the respective Relationship Types can be used by a TOSCA implementation as is.

9.1 XML Syntax

2052 The following pseudo schema defines the XML syntax of Relationship Type Implementations:

```

2053 01 <RelationshipTypeImplementation name="xs:NCName"
2054 02         targetNamespace="xs:anyURI"?
2055 03         relationshipType="xs:QName"
2056 04         abstract="yes|no"?
2057 05         final="yes|no"?>
2058 06
2059 07 <Tags>
2060 08   <Tag name="xs:string" value="xs:string" /> +
2061 09 </Tags> ?
2062 10
2063 11 <DerivedFrom relationshipTypeImplementationRef="xs:QName" /> ?
2064 12
2065 13 <RequiredContainerFeatures>
2066 14   <RequiredContainerFeature feature="xs:anyURI" /> +
2067 15 </RequiredContainerFeatures> ?
2068 16
2069 17 <ImplementationArtifacts>
2070 18-  <ImplementationArtifact name="xs:string"
2071 1918  interfaceName="xs:NCName | xs:anyURI"?
2072 2019  operationName="xs:NCName"?
2073 2120  artifactType="xs:QName"
2074 2221  artifactRef="xs:QName"?>
2075 2322  artifact specific content ?
2076 2423  <ImplementationArtifact> +
2077 2524  </ImplementationArtifacts> ?
2078 2625
2079 2726 </RelationshipTypeImplementation>

```

9.2 Properties

2081 The RelationshipTypeImplementation element has the following properties:

- 2082 • name: This attribute specifies the name or identifier of the Relationship Type Implementation,
2083 which MUST be unique within the target namespace.

2084 • `targetNamespace`: This OPTIONAL attribute specifies the target namespace to which the
2085 definition of the Relationship Type Implementation will be added. If not specified, the Relationship
2086 Type Implementation will be added to the target namespace of the enclosing Definitions
2087 document.

2088 • `relationshipType`: The QName value of this attribute specifies the Relationship Type
2089 implemented by this Relationship Type Implementation.

2090 • `abstract`: This OPTIONAL attribute specifies that this Relationship Type Implementation
2091 cannot be used directly as an implementation for the Relationship Type specified in the
2092 `relationshipType` attribute.

2093
2094 For example, a Relationship Type implementer might decide to deliver only part of the
2095 implementation of a specific Relationship Type (i.e. for only some operations) for re-use purposes
2096 and require the implementation for specific operations to be delivered in a more concrete, derived
2097 Relationship Type Implementation.

2098
2099 Note: an abstract Relationship Type Implementation MUST NOT be declared as final.

2100 • `final`: This OPTIONAL attribute specifies that other Relationship Type Implementations MUST
2101 NOT be derived from this Relationship Type Implementation.

2102
2103 Note: a final Relationship Type Implementation MUST NOT be declared as abstract.

2104 • `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by
2105 the author to describe the Relationship Type Implementation. Each tag is defined by a separate,
2106 nested `Tag` element.

2107 The `Tag` element has the following properties:

2108 o `name`: This attribute specifies the name of the tag.

2109 o `value`: This attribute specifies the value of the tag.

2110
2111 Note: The name/value pairs defined in tags have no normative interpretation.

2112 • `DerivedFrom`: This is an OPTIONAL reference to another Relationship Type Implementation
2113 from which this Relationship Type Implementation derives. See section 9.3 Derivation Rules or
2114 details.

2115 The `DerivedFrom` element has the following properties:

2116 o `relationshipTypeImplementationRef`: The QName specifies the Relationship
2117 Type Implementation from which this Relationship Type Implementation derives.

2118 • `RequiredContainerFeatures`: An implementation of a Relationship Type might depend
2119 on certain features of the environment it is executed in, such as specific (potentially proprietary)
2120 APIs of the TOSCA container.

2121 Thus, the contents of the `RequiredContainerFeatures` element provide “hints” to the
2122 TOSCA container allowing it to select the appropriate Relationship Type Implementation if
2123 multiple alternatives are provided.

2124 Each such dependency is defined by a separate `RequiredContainerFeature` element.

2125 The `RequiredContainerFeature` element has the following properties:

2126 o `feature`: The value of this attribute is a URI that denotes the corresponding needed
2127 feature of the environment.

2128 • `ImplementationArtifacts`: This element specifies a set of implementation artifacts for
2129 interfaces or operations of a Relationship Type.

2130 The `ImplementationArtifacts` element has the following properties:

2131 o `ImplementationArtifact`: This element specifies one implementation artifact of
2132 an interface or an operation.
2133

2134 Note: Multiple implementation artifacts might be needed to implement a Relationship
2135 Type according to the attributes defined below. An implementation artifact MAY serve as
2136 implementation for all interfaces and all operations defined for the Relationship Type, it
2137 MAY serve as implementation for one interface (and all its operations), or it MAY serve
2138 as implementation for only one specific operation.
2139

2140 The `ImplementationArtifact` element has the following properties:

- 2141 ▪ `name`: This attribute specifies the name of the artifact, which SHOULD be unique
2142 within the scope of the encompassing Node Type Implementation.
- 2143 ▪ `interfaceName`: This OPTIONAL attribute specifies the name of the interface
2144 that is implemented by the actual implementation artifact. If not specified, the
2145 implementation artifact is assumed to provide the implementation for all
2146 interfaces defined by the Relationship Type referred to by the
2147 `relationshipType` attribute of the containing
2148 `RelationshipTypeImplementation`.

2149
2150 Note that the referenced interface can be defined in either the
2151 `SourceInterfaces` element or the `TargetInterfaces` element of the
2152 Relationship Type implemented by this Relationship Type Implementation.

- 2153 ▪ `operationName`: This OPTIONAL attribute specifies the name of the
2154 operation that is implemented by the actual implementation artifact. If specified,
2155 the `interfaceName` MUST be specified and the specified `operationName`
2156 MUST refer to an operation of the specified interface. If not specified, the
2157 implementation artifact is assumed to provide the implementation for all
2158 operations defined within the specified interface.
- 2159 ▪ `artifactType`: This attribute specifies the type of this artifact. The QName
2160 value of this attribute SHOULD correspond to the QName of an
2161 `ArtifactType` defined in the same Definitions document or in an imported
2162 document.
2163

2164 The `artifactType` attribute specifies the artifact type specific content of the
2165 `ImplementationArtifact` element body and indicates the type of Artifact
2166 Template referenced by the Implementation Artifact via the `artifactRef`
2167 attribute.

- 2168 ▪ `artifactRef`: This OPTIONAL attribute contains a QName that identifies an
2169 Artifact Template to be used as implementation artifact. This Artifact Template
2170 can be defined in the same Definitions document or in a separate, imported
2171 document.
2172 The type of Artifact Template referenced by the `artifactRef` attribute MUST
2173 be the same type or a sub-type of the type specified in the `artifactType`
2174 attribute.
2175

2176 Note: if no Artifact Template is referenced, the artifact type specific content of the
2177 `ImplementationArtifact` element alone is assumed to represent the
2178 actual artifact. For example, a simple script could be defined in place within the
2179 `ImplementationArtifact` element.

2180 9.3 Derivation Rules

2181 The following rules on combining definitions based on `DerivedFrom` apply:

- 2182 • Implementation Artifacts: The set of implementation artifacts of a Relationship Type
2183 Implementation consists of the set union of implementation artifacts defined by the Relationship

2184 Type Implementation itself and the implementation artifacts defined by any Relationship Type
 2185 Implementation the Relationship Type Implementation is derived from.
 2186 An implementation artifact defined by a Node Type Implementation overrides an implementation
 2187 artifact having the same interface name and operation name of a Relationship Type
 2188 Implementation the Relationship Type Implementation is derived from.
 2189 If an implementation artifact defined in a Relationship Type Implementation specifies only an
 2190 interface name, it substitutes implementation artifacts having the same interface name (with or
 2191 without an operation name defined) of any Relationship Type Implementation the Relationship
 2192 Type Implementation is derived from. In this case, the implementation of a complete interface of a
 2193 Relationship Type is overridden.
 2194 If an implementation artifact defined in a Relationship Type Implementation neither defines an
 2195 interface name nor an operation name, it overrides all implementation artifacts of any
 2196 Relationship Type Implementation the Relationship Type Implementation is derived from. In this
 2197 case, the complete implementation of a Relationship Type is overridden.

2198 9.4 Example

2199 The following example defines the Node Type Implementation “MyDBMSImplementation”. This is an
 2200 implementation of a Node Type “DBMS”.

```

2201 01 <Definitions id="MyImpls" name="My Implementations"
2202 02   targetNamespace="http://www.example.com/SampleImplementations"
2203 03   xmlns:bn="http://www.example.com/BaseRelationshipTypes"
2204 04   xmlns:ba="http://www.example.com/BaseArtifactTypes"
2205 05   xmlns:sa="http://www.example.com/SampleArtifacts">
2206 06
2207 07   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
2208 08     namespace="http://www.example.com/BaseArtifactTypes"/>
2209 09
2210 10   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
2211 11     namespace="http://www.example.com/BaseRelationshipTypes"/>
2212 12
2213 13   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
2214 14     namespace="http://www.example.com/SampleArtifacts"/>
2215 15
2216 16   <RelationshipTypeImplementation name="MyDBConnectImplementation"
2217 17     relationshipType="bn:DBConnection">
2218 18
2219 19     <ImplementationArtifacts>
2220 20     <ImplementationArtifact name="MyDBConnectionImpl"
2221 21       interfaceName="ConnectionInterface"
2222 22       operationName="connectTo"
2223 23       artifactType="ba:ScriptArtifact"
2224 24       artifactRef="sa:MyConnectScript">
2225 25     <ImplementationArtifact>
2226 26     </ImplementationArtifacts>
2227 27   </RelationshipTypeImplementation>
2228 28
2229 29 </Definitions>
  
```

2231 The Relationship Type Implementation contains the “MyDBConnectionImpl” implementation artifact,
 2232 which is an artifact for the “ConnectionInterface” interface that has been defined for the “DBConnection”
 2233 base Relationship Type. The type of this artifact is a “ScriptArtifact” that has been defined as base Artifact
 2234 Type. The implementation artifact refers to the “MyConnectScript” Artifact Template that has been defined
 2235 before.

2236

10 Requirement Types

2237 This chapter specifies how *Requirement Types* are defined. A Requirement Type is a reusable entity that
2238 describes a kind of requirement that a Node Type can declare to expose. For example, a Requirement
2239 Type for a database connection can be defined and various Node Types (e.g. a Node Type for an
2240 application) can declare to expose (or “to have”) a requirement for a database connection.

2241 A Requirement Type defines the structure of observable properties via a *Properties Definition*, i.e. the
2242 names, data types and allowed values the properties defined in *Requirements* of Node Templates of a
2243 Node Type can have in cases where the Node Type defines a requirement of the respective Requirement
2244 Type.

2245 A Requirement Type can inherit properties and semantics from another Requirement Type by means of
2246 the *DerivedFrom* element. Requirement Types might be declared as abstract, meaning that they
2247 cannot be instantiated. The purpose of such abstract Requirement Types is to provide common
2248 properties for re-use in specialized, derived Requirement Types. Requirement Types might also be
2249 declared as final, meaning that they cannot be derived by other Requirement Types.

10.1 XML Syntax

2250 The following pseudo schema defines the XML syntax of Requirement Types:

```
2251 01 <RequirementType name="xs:NCName"  
2252 02     targetNamespace="xs:anyURI"?  
2253 03     abstract="yes|no"?  
2254 04     final="yes|no"?  
2255 05     requiredCapabilityType="xs:QName"?>  
2256 06  
2257 07 <Tags>  
2258 08     <Tag name="xs:string" value="xs:string"/> +  
2259 09 </Tags> ?  
2260 10  
2261 11 <DerivedFrom typeRef="xs:QName"/> ?  
2262 12  
2263 13 <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?  
2264 14  
2265 15 </RequirementType>
```

10.2 Properties

2266 The RequirementType element has the following properties:

- 2269
- 2270 • **name**: This attribute specifies the name or identifier of the Requirement Type, which **MUST** be
unique within the target namespace.
 - 2271 • **targetNamespace**: This **OPTIONAL** attribute specifies the target namespace to which the
2272 definition of the Requirement Type will be added. If not specified, the Requirement Type definition
2273 will be added to the target namespace of the enclosing Definitions document.
 - 2274 • **abstract**: This **OPTIONAL** attribute specifies that no instances can be created from Node
2275 Templates of a Node Type that defines a requirement of this Requirement Type.

2276
2277 As a consequence, a Node Type with a Requirement Definition of an abstract Requirement Type
2278 **MUST** be declared as abstract as well and a derived Node Type that defines a requirement of a
2279 type derived from the abstract Requirement Type has to be defined. For example, an abstract
2280 Node Type “Application” might be defined having a requirement of the abstract type “Container”.
2281 A derived Node Type “Web Application” can then be defined with a more concrete requirement of
2282 type “Web Application Container” which can then be used for defining Node Templates that can

2283 be instantiated during the creation of a service according to a Service Template.
 2284
 2285 Note: an abstract Requirement Type MUST NOT be declared as final.

- 2286 • `final`: This OPTIONAL attribute specifies that other Requirement Types MUST NOT be derived
 2287 from this Requirement Type.
 2288
 2289 Note: a final Requirement Type MUST NOT be declared as abstract.
- 2290 • `requiredCapabilityType`; This OPTIONAL attribute specifies the type of capability
 2291 needed to match the defined Requirement Type. The QName value of this attribute refers to the
 2292 QName of a `CapabilityType` element defined in the same Definitions document or in a
 2293 separate, imported document.
 2294
 2295 Note: The following basic match-making for Requirements and Capabilities MUST be supported
 2296 by each TOSCA implementation. Each Requirement is defined by a Requirement Definition,
 2297 which in turn refers to a Requirement Type that specifies the needed Capability Type by means of
 2298 its `requiredCapabilityType` attribute. The value of this attribute is used for basic type-
 2299 based match-making: a Capability matches a Requirement if the Requirement's Requirement
 2300 Type has a `requiredCapabilityType` value that corresponds to the Capability Type of the
 2301 Capability or one of its super-types.
 2302 Any domain-specific match-making semantics (e.g. based on constraints or properties) has to be
 2303 defined in the cause of specifying the corresponding Requirement Types and Capability Types.
- 2304 • `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by
 2305 the author to describe the Requirement Type. Each tag is defined by a separate, nested `Tag`
 2306 element.
 2307 The `Tag` element has the following properties:
 - 2308 ○ `name`: This attribute specifies the name of the tag.
 - 2309 ○ `value`: This attribute specifies the value of the tag.

2310 Note: The name/value pairs defined in tags have no normative interpretation.

- 2311 • `DerivedFrom`: This is an OPTIONAL reference to another Requirement Type from which this
 2312 Requirement Type derives. See section 10.3 Derivation Rules for details.
 2313 The `DerivedFrom` element has the following properties:
 - 2314 ○ `typeRef`: The QName specifies the Requirement Type from which this Requirement
 2315 Type derives its definitions and semantics.
- 2316 • `PropertiesDefinition`: This element specifies the structure of the observable properties
 2317 of the Requirement Type, such as its configuration and state, by means of XML schema.
 2318 The `PropertiesDefinition` element has one but not both of the following properties:
 - 2319 ○ `element`: This attribute provides the QName of an XML element defining the structure
 2320 of the Requirement Type Properties.
 2321
 - 2322 ○ `type`: This attribute provides the QName of an XML (complex) type defining the
 2323 structure of the Requirement Type Properties.

2324 10.3 Derivation Rules

2325 The following rules on combining definitions based on `DerivedFrom` apply:

- 2326 • Requirement Type Properties: It is assumed that the XML element (or type) representing the
 2327 Requirement Type Properties extends the XML element (or type) of the Requirement Type
 2328 Properties of the Requirement Type referenced in the `DerivedFrom` element.

2329 10.4 Example

2330 The following example defines the Requirement Type “DatabaseClientEndpoint” that expresses the
2331 requirement of a client for a database connection. It is defined in a Definitions document
2332 “MyRequirements” within the target namespace “http://www.example.com/SampleRequirements”. Thus,
2333 by importing the corresponding namespace into another Definitions document, the
2334 “DatabaseClientEndpoint” Requirement Type is available for use in the other document.

```
2335 01 <Definitions id="MyRequirements" name="My Requirements"  
2336 02   targetNamespace="http://www.example.com/SampleRequirements"  
2337 03   xmlns:br="http://www.example.com/BaseRequirementTypes"  
2338 04   xmlns:mrp="http://www.example.com/SampleRequirementProperties">  
2339 05  
2340 06   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"  
2341 07     namespace="http://www.example.com/BaseRequirementTypes"/>  
2342 08  
2343 09   <Import importType="http://www.w3.org/2001/XMLSchema"  
2344 10     namespace="http://www.example.com/SampleRequirementProperties"/>  
2345 11  
2346 12   <RequirementType name="DatabaseClientEndpoint">  
2347 13     <DerivedFrom typeRef="br:ClientEndpoint"/>  
2348 14     <PropertiesDefinition  
2349 15       element="mrp:DatabaseClientEndpointProperties"/>  
2350 16   </RequirementType>  
2351 17  
2352 18 </Definitions>
```

2353 The Requirement Type “DatabaseClientEndpoint” defined in the example above is derived from another
2354 generic “ClientEndpoint” Requirement Type defined in a separate file by means of the `DerivedFrom`
2355 element. The definitions in that separate Definitions file are imported by means of the first `Import`
2356 element and the namespace of those imported definitions is assigned the prefix “br” in the current file.

2357 The “DatabaseClientEndpoint” Requirement Type defines a set of properties through an XML schema
2358 element definition “DatabaseClientEndpointProperties”. For example, those properties might include the
2359 definition of a port number to be used for client connections. The XML schema definition is stored in a
2360 separate XSD file that is imported by means of the second `Import` element. The namespace of the XML
2361 schema definitions is assigned the prefix “mrp” in the current file.

2362

11 Capability Types

2363 This chapter specifies how *Capability Types* are defined. A Capability Type is a reusable entity that
2364 describes a kind of capability that a Node Type can declare to expose. For example, a Capability Type for
2365 a database server endpoint can be defined and various Node Types (e.g. a Node Type for a database)
2366 can declare to expose (or to “provide”) the capability of serving as a database server endpoint.

2367 A Capability Type defines the structure of observable properties via a *Properties Definition*, i.e. the
2368 names, data types and allowed values the properties defined in *Capabilities* of Node Templates of a Node
2369 Type can have in cases where the Node Type defines a capability of the respective Capability Type.

2370 A Capability Type can inherit properties and semantics from another Capability Type by means of the
2371 *DerivedFrom* element. Capability Types might be declared as abstract, meaning that they cannot be
2372 instantiated. The purpose of such abstract Capability Types is to provide common properties for re-use in
2373 specialized, derived Capability Types. Capability Types might also be declared as final, meaning that they
2374 cannot be derived by other Capability Types.

11.1 XML Syntax

2375 The following pseudo schema defines the XML syntax of Capability Types:

```
2377 01 <CapabilityType name="xs:NCName"  
2378 02           targetNamespace="xs:anyURI"?  
2379 03           abstract="yes|no"?  
2380 04           final="yes|no"?>  
2381 05  
2382 06   <Tags>  
2383 07     <Tag name="xs:string" value="xs:string"/> +  
2384 08   </Tags> ?  
2385 09  
2386 10   <DerivedFrom typeRef="xs:QName"/> ?  
2387 11  
2388 12   <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?  
2389 13  
2390 14 </CapabilityType>
```

11.2 Properties

2391 The *CapabilityType* element has the following properties:

- 2393 • **name**: This attribute specifies the name or identifier of the Capability Type, which **MUST** be
2394 unique within the target namespace.
- 2395 • **targetNamespace**: This **OPTIONAL** attribute specifies the target namespace to which the
2396 definition of the Capability Type will be added. If not specified, the Capability Type definition will
2397 be added to the target namespace of the enclosing Definitions document.
- 2398 • **abstract**: This **OPTIONAL** attribute specifies that no instances can be created from Node
2399 Templates of a Node Type that defines a capability of this Capability Type.

2400
2401 As a consequence, a Node Type with a Capability Definition of an abstract Capability Type **MUST**
2402 be declared as abstract as well and a derived Node Type that defines a capability of a type
2403 derived from the abstract Capability Type has to be defined. For example, an abstract Node Type
2404 “Server” might be defined having a capability of the abstract type “Container”. A derived Node
2405 Type “Web Server” can then be defined with a more concrete capability of type “Web Application
2406 Container” which can then be used for defining Node Templates that can be instantiated during
2407 the creation of a service according to a Service Template.

- 2408
- 2409 Note: an abstract Capability Type MUST NOT be declared as final.
- 2410
- 2411
- 2412
- 2413 Note: a final Capability Type MUST NOT be declared as abstract.
- 2414
- 2415
- 2416
- 2417
- **final:** This OPTIONAL attribute specifies that other Capability Types MUST NOT be derived from this Capability Type.
- 2418
- 2419
- 2420
- 2421 Note: a final Capability Type MUST NOT be declared as abstract.
- 2422
- 2423
- 2424
- **Tags:** This OPTIONAL element allows the definition of any number of tags which can be used by the author to describe the Capability Type. Each tag is defined by a separate, nested Tag element.
- 2425
- 2426
- 2427
- 2428
- 2429
- The Tag element has the following properties:
- **name:** This attribute specifies the name of the tag.
 - **value:** This attribute specifies the value of the tag.
- 2430
- 2431
- 2432
- 2433
- Note: The name/value pairs defined in tags have no normative interpretation.
- **DerivedFrom:** This is an OPTIONAL reference to another Capability Type from which this Capability Type derives. See section 11.3 Derivation Rules for details.
- The DerivedFrom element has the following properties:
- **typeRef:** The QName specifies the Capability Type from which this Capability Type derives its definitions and semantics.
 - **PropertiesDefinition:** This element specifies the structure of the observable properties of the Capability 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 Capability Type Properties.
 - **type:** This attribute provides the QName of an XML (complex) type defining the structure of the Capability Type Properties.

2434 11.3 Derivation Rules

2435 The following rules on combining definitions based on DerivedFrom apply:

- **Capability Type Properties:** It is assumed that the XML element (or type) representing the Capability Type Properties extends the XML element (or type) of the Capability Type Properties of the Capability Type referenced in the DerivedFrom element.

2439 11.4 Example

2440 The following example defines the Capability Type “DatabaseServerEndpoint” that expresses the
 2441 capability of a component to serve database connections. It is defined in a Definitions document
 2442 “MyCapabilities” within the target namespace “http://www.example.com/SampleCapabilities”. Thus, by
 2443 importing the corresponding namespace into another Definitions document, the
 2444 “DatabaseServerEndpoint” Capability Type is available for use in the other document.

```

2445 01 <Definitions id="MyCapabilities" name="My Capabilities"
2446 02   targetNamespace="http://www.example.com/SampleCapabilities"
2447 03   xmlns:bc="http://www.example.com/BaseCapabilityTypes"
2448 04   xmlns:mcp="http://www.example.com/SampleCapabilityProperties">
2449 05
2450 06 <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
2451 07   namespace="http://www.example.com/BaseCapabilityTypes"/>
2452 08
2453 09 <Import importType="http://www.w3.org/2001/XMLSchema"
2454 10   namespace="http://www.example.com/SampleCapabilityProperties"/>
  
```

```
2455 11
2456 12 <CapabilityType name="DatabaseServerEndpoint">
2457 13   <DerivedFrom typeRef="bc:ServerEndpoint"/>
2458 14   <PropertiesDefinition
2459 15     element="mcp:DatabaseServerEndpointProperties"/>
2460 16 </CapabilityType>
2461 17
2462 18 </Definitions>
```

2463 The Capability Type "DatabaseServerEndpoint" defined in the example above is derived from another
2464 generic "ServerEndpoint" Capability Type defined in a separate file by means of the `DerivedFrom`
2465 element. The definitions in that separate Definitions file are imported by means of the first `Import`
2466 element and the namespace of those imported definitions is assigned the prefix "bc" in the current file.

2467 The "DatabaseServerEndpoint" Capability Type defines a set of properties through an XML schema
2468 element definition "DatabaseServerEndpointProperties". For example, those properties might include the
2469 definition of a port number where the server listens for client connections, or credentials to be used by
2470 clients. The XML schema definition is stored in a separate XSD file that is imported by means of the
2471 second `Import` element. The namespace of the XML schema definitions is assigned the prefix "mcp"
2472 in the current file.

2473

12 Artifact Types

2474 This chapter specifies how *Artifact Types* are defined. An Artifact Type is a reusable entity that defines
2475 the type of one or more Artifact Templates which in turn serve as deployment artifacts for Node
2476 Templates or implementation artifacts for Node Type and Relationship Type interface operations. For
2477 example, an Artifact Type “WAR File” might be defined for describing web application archive files. Based
2478 on this Artifact Type, one or more Artifact Templates representing concrete WAR files can be defined and
2479 referenced as deployment or implementation artifacts.

2480 An Artifact Type can define the structure of observable properties via a *Properties Definition*, i.e. the
2481 names, data types and allowed values the properties defined in Artifact Templates using an Artifact Type
2482 or instances of such Artifact Templates can have. Note that properties defined by an Artifact Type are
2483 assumed to be invariant across the contexts in which corresponding artifacts are used – as opposed to
2484 properties that can vary depending on the context. As an example of such an invariant property, an
2485 Artifact Type for a WAR file could define a “signature” property that can hold a hash for validating the
2486 actual artifact proper. In contrast, the path where the web application contained in the WAR file gets
2487 deployed can vary for each place where the WAR file is used.

2488 An Artifact Type can inherit definitions and semantics from another Artifact Type by means of the
2489 *DerivedFrom* element. Artifact Types can be declared as abstract, meaning that they cannot be
2490 instantiated. The purpose of such abstract Artifact Types is to provide common properties for re-use in
2491 specialized, derived Artifact Types. Artifact Types can also be declared as final, meaning that they cannot
2492 be derived by other Artifact Types.

2493

12.1 XML Syntax

2494 The following pseudo schema defines the XML syntax of Artifact Types:

```
2495 01 <ArtifactType name="xs:NCName"  
2496 02     targetNamespace="xs:anyURI"?  
2497 03     abstract="yes|no"?  
2498 04     final="yes|no"?>  
2499 05  
2500 06   <Tags>  
2501 07     <Tag name="xs:string" value="xs:string"/> +  
2502 08   </Tags> ?  
2503 09  
2504 10   <DerivedFrom typeRef="xs:QName"/> ?  
2505 11  
2506 12   <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?  
2507 13  
2508 14 </ArtifactType>
```

2509

12.2 Properties

2510 The *ArtifactType* element has the following properties:

- 2511 • **name**: This attribute specifies the name or identifier of the Artifact Type, which MUST be unique
2512 within the target namespace.
- 2513 • **targetNamespace**: This OPTIONAL attribute specifies the target namespace to which the
2514 definition of the Artifact Type will be added. If not specified, the Artifact Type definition will be
2515 added to the target namespace of the enclosing Definitions document.
- 2516 • **abstract**: This OPTIONAL attribute specifies that no instances can be created from Artifact
2517 Templates of that abstract Artifact Type, i.e. the respective artifacts cannot be used directly as
2518 deployment or implementation artifact in any context.
2519

2520 As a consequence, an Artifact Template of an abstract Artifact Type MUST be replaced by an
2521 artifact of a derived Artifact Type at the latest during deployment of the element that uses the
2522 artifact (i.e. a Node Template or Relationship Template).

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

- 2525 • `final`: This OPTIONAL attribute specifies that other Artifact Types MUST NOT be derived from
2526 this Artifact Type.

2527
2528 Note: a final Artifact Type MUST NOT be declared as abstract.

- 2529 • `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by
2530 the author to describe the Artifact Type. Each tag is defined by a separate, nested `Tag` element.
2531 The `Tag` element has the following properties:

- 2532 ○ `name`: This attribute specifies the name of the tag.
- 2533 ○ `value`: This attribute specifies the value of the tag.

2534
2535 Note: The name/value pairs defined in tags have no normative interpretation.

- 2536 • `DerivedFrom`: This is an OPTIONAL reference to another Artifact Type from which this Artifact
2537 Type derives. See section 12.3 Derivation Rules for details.

2538 The `DerivedFrom` element has the following properties:

- 2539 ○ `typeRef`: The QName specifies the Artifact Type from which this Artifact Type derives
2540 its definitions and semantics.

- 2541 • `PropertiesDefinition`: This element specifies the structure of the observable properties
2542 of the Artifact Type, such as its configuration and state, by means of XML schema.

2543 The `PropertiesDefinition` element has one but not both of the following properties:

- 2544 ○ `element`: This attribute provides the QName of an XML element defining the structure
2545 of the Artifact Type Properties.
- 2546 ○ `type`: This attribute provides the QName of an XML (complex) type defining the
2547 structure of the Artifact Type Properties.

2548 12.3 Derivation Rules

2549 The following rules on combining definitions based on `DerivedFrom` apply:

- 2550 • `Artifact Type Properties`: It is assumed that the XML element (or type) representing the Artifact
2551 Type Properties extends the XML element (or type) of the Artifact Type Properties of the Artifact
2552 Type referenced in the `DerivedFrom` element.

2553 12.4 Example

2554 The following example defines the Artifact Type “RPMPackage” that can be used for describing RPM
2555 packages as deployable artifacts on various Linux distributions. It is defined in a Definitions document
2556 “MyArtifacts” within the target namespace “<http://www.example.com/SampleArtifacts>”. Thus, by importing
2557 the corresponding namespace into another Definitions document, the “RPMPackage” Artifact Type is
2558 available for use in the other document.

```
2559 01 <Definitions id="MyArtifacts" name="My Artifacts"  
2560 02   targetNamespace="http://www.example.com/SampleArtifacts"  
2561 03   xmlns:ba="http://www.example.com/BaseArtifactTypes"  
2562 04   xmlns:map="http://www.example.com/SampleArtifactProperties">  
2563 05  
2564 06   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"  
2565 07     namespace="http://www.example.com/BaseArtifactTypes"/>  
2566 08
```

```
2567 09 <Import importType="http://www.w3.org/2001/XMLSchema"
2568 10     namespace="http://www.example.com/SampleArtifactProperties"/>
2569 11
2570 12 <ArtifactType name="RPMPackage">
2571 13     <DerivedFrom typeRef="ba:OSPackage"/>
2572 14     <PropertiesDefinition element="map:RPMPackageProperties"/>
2573 15 </ArtifactType>
2574 16
2575 17 </Definitions>
```

2576 The Artifact Type “RPMPackage” defined in the example above is derived from another generic
2577 “OSPackage” Artifact Type defined in a separate file by means of the `DerivedFrom` element. The
2578 definitions in that separate Definitions file are imported by means of the first `Import` element and the
2579 namespace of those imported definitions is assigned the prefix “ba” in the current file.

2580 The “RPMPackage” Artifact Type defines a set of properties through an XML schema element definition
2581 “RPMPackageProperties”. For example, those properties might include the definition of the name or
2582 names of one or more RPM packages. The XML schema definition is stored in a separate XSD file that is
2583 imported by means of the second `Import` element. The namespace of the XML schema definitions is
2584 assigned the prefix “map” in the current file.

2585

13 Artifact Templates

2586 This chapter specifies how *Artifact Templates* are defined. An Artifact Template represents an artifact that
2587 can be referenced from other objects in a Service Template as a deployment artifact or implementation
2588 artifact. For example, from Node Types or Node Templates, an Artifact Template for some software
2589 installable could be referenced as a deployment artifact for materializing a specific software component.
2590 As another example, from within interface definitions of Node Types or Relationship Types, an Artifact
2591 Template for a WAR file could be referenced as implementation artifact for a REST operation.

2592 An Artifact Template refers to a specific Artifact Type that defines the structure of observable properties
2593 (metadata) or the artifact. The Artifact Template then typically defines values for those properties inside
2594 the `Properties` element. Note that properties defined by an Artifact Type are assumed to be invariant
2595 across the contexts in which corresponding artifacts are used – as opposed to properties that can vary
2596 depending on the context.

2597 Furthermore, an Artifact Template typically provides one or more references to the actual artifact itself
2598 that can be contained as a file in the CSAR (see section 3.7 and section 14) containing the overall
2599 Service Template or that can be available at a remote location such as an FTP server.

2600

13.1 XML Syntax

2601 The following pseudo schema defines the XML syntax of Artifact Templates:

```
2602 01 <ArtifactTemplate id="xs:ID" name="xs:string"? type="xs:QName">  
2603 02  
2604 03   <Properties>  
2605 04     XML fragment  
2606 05   </Properties> ?  
2607 06  
2608 07   <PropertyConstraints>  
2609 08     <PropertyConstraint property="xs:string"  
2610 09       constraintType="xs:anyURI"> +  
2611 10       constraint ?  
2612 11     </PropertyConstraint>  
2613 12   </PropertyConstraints> ?  
2614 13  
2615 14   <ArtifactReferences>  
2616 15     <ArtifactReference reference="xs:anyURI">  
2617 16       (  
2618 17         <Include pattern="xs:string"/>  
2619 18         |  
2620 19         <Exclude pattern="xs:string"/>  
2621 20       )*  
2622 21     </ArtifactReference> +  
2623 22   </ArtifactReferences> ?  
2624 23  
2625 24 </ArtifactTemplate>
```

2626

13.2 Properties

2627 The `ArtifactTemplate` element has the following properties:

- 2628 • `id`: This attribute specifies the identifier of the Artifact Template. The identifier of the Artifact
2629 Template MUST be unique within the target namespace.
- 2630 • `name`: This OPTIONAL attribute specifies the name of the Artifact Template.

- 2631
- 2632
- 2633
- 2634
- 2635
- 2636
- 2637
- 2638
- `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.
- 2639
- 2640
- 2641
- 2642
- 2643
- 2644
- 2645
- 2646
- `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.
- 2647
- 2648
- 2649
- 2650
- 2651
- `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.
- 2664
- 2665
- 2666
- `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:

- 2680
- 2681
- 2682
- `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.

2683 13.3 Example

2684 The following example defines the Artifact Template "MyInstallable" that points to a zip file containing
2685 some software installable. It is defined in a Definitions document "MyArtifacts" within the target
2686 namespace "http://www.example.com/SampleArtifacts". The Artifact Template can be used in the same
2687 document, for example as a deployment artifact for some Node Template representing a software
2688 component, or it can be used in other Definitions documents by importing the corresponding namespace
2689 into another document.

```
2690 01 <Definitions id="MyArtifacts" name="My Artifacts"  
2691 02   targetNamespace="http://www.example.com/SampleArtifacts"  
2692 03   xmlns:ba="http://www.example.com/BaseArtifactTypes">  
2693 04  
2694 05   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"  
2695 06     namespace="http://www.example.com/BaseArtifactTypes"/>  
2696 07  
2697 08   <ArtifactTemplate id="MyInstallable"  
2698 09     name="My installable"  
2699 10     type="ba:ZipFile">  
2700 11     <ArtifactReferences>  
2701 12       <ArtifactReference reference="files/MyInstallable.zip"/>  
2702 13     </ArtifactReferences>  
2703 14   </ArtifactTemplate>  
2704 15  
2705 16 </Definitions>
```

2706 The Artifact Template "MyInstallable" defined in the example above is of type "ZipFile" that is specified in
2707 the `type` attribute of the `ArtifactTemplate` element. This Artifact Type is defined in a separate file,
2708 the definitions of which are imported by means of the `Import` element and the namespace of those
2709 imported definitions is assigned the prefix "ba" in the current file.

2710 The "MyInstallable" Artifact Template provides a reference to a file "MyInstallable.zip" by means of the
2711 `ArtifactReference` element. Since the URI provided in the `reference` attribute is a relative URI,
2712 it is interpreted relative to the root directory of the CSAR containing the Service Template.

2713

14 Policy Types

2714 This chapter specifies how *Policy Types* are defined. A Policy Type is a reusable entity that describes a
2715 kind of non-functional behavior or a kind of quality-of-service (QoS) that a Node Type can declare to
2716 expose. For example, a Policy Type can be defined to express high availability for specific Node Types
2717 (e.g. a Node Type for an application server).

2718 A Policy Type defines the structure of observable properties via a Properties Definition, i.e. the names,
2719 data types and allowed values the properties defined in a corresponding Policy Template can have.

2720 A Policy Type can inherit properties from another Policy Type by means of the `DerivedFrom` element.

2721 A Policy Type declares the set of Node Types it specifies non-functional behavior for via the `AppliesTo`
2722 element. Note that being “applicable to” does not enforce implementation: i.e. in case a Policy Type
2723 expressing high availability is associated with a “Webserver” Node Type, an instance of the Webserver is
2724 not necessarily highly available. Whether or not an instance of a Node Type to which a Policy Type is
2725 applicable will show the specified non-functional behavior, is determined by a Node Template of the
2726 corresponding Node Type.

2727

14.1 XML Syntax

2728 The following pseudo schema defines the XML syntax of Policy Types:

```
2729 01 <PolicyType name="xs:NCName"  
2730 02     policyLanguage="xs:anyURI"?  
2731 03     abstract="yes|no"?  
2732 04     final="yes|no"?  
2733 05     targetNamespace="xs:anyURI"?>  
2734 06   <Tags>  
2735 07     <Tag name="xs:string" value="xs:string"/> +  
2736 08   </Tags> ?  
2737 09  
2738 10   <DerivedFrom typeRef="xs:QName"/> ?  
2739 11  
2740 12   <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?  
2741 13  
2742 14   <AppliesTo>  
2743 15     <NodeTypeReference typeRef="xs:QName"/> +  
2744 16   </AppliesTo> ?  
2745 17  
2746 18   policy type specific content ?  
2747 19  
2748 20 </PolicyType>
```

2749

14.2 Properties

2750 The `PolicyType` element has the following properties:

- 2751 • `name`: This attribute specifies the name or identifier of the Policy Type, which **MUST** be unique
2752 within the target namespace.
- 2753 • `targetNamespace`: This **OPTIONAL** attribute specifies the target namespace to which the
2754 definition of the Policy Type will be added. If not specified, the Policy Type definition will be added
2755 to the target namespace of the enclosing Definitions document.
- 2756 • `policyLanguage`: This **OPTIONAL** attribute specifies the language used to specify the details
2757 of the Policy Type. These details can be defined as policy type specific content of the `PolicyType`
2758 element.

2759 • **abstract**: This OPTIONAL attribute specifies that no instances can be created from Policy
2760 Templates of that abstract Policy Type, i.e. the respective policies cannot be used directly during
2761 the instantiation of a Service Template.

2762
2763 As a consequence, a Policy Template of an abstract Policy Type MUST be replaced by a policy
2764 of a derived Policy Type at the latest during deployment of the element that policy is attached to.

2765 • **final**: This OPTIONAL attribute specifies that other Policy Types MUST NOT be derived from
2766 this Policy Type.

2767
2768 Note: a final Policy Type MUST NOT be declared as abstract.

2769 • **Tags**: This OPTIONAL element allows the definition of any number of tags which can be used by
2770 the author to describe the Policy Type. Each tag is defined by a separate, nested `Tag` element.

2771 The `Tag` element has the following properties:

2772 o **name**: This attribute specifies the name of the tag.

2773 o **value**: This attribute specifies the value of the tag.

2774
2775 Note: The name/value pairs defined in tags have no normative interpretation.

2776 • **DerivedFrom**: This is an OPTIONAL reference to another Policy Type from which this Policy
2777 Type derives. See section 14.3 Derivation Rules for details.

2778 The `DerivedFrom` element has the following properties:

2779 o **typeRef**: The QName specifies the Policy Type from which this Policy Type derives its
2780 definitions from.

2781 • **PropertiesDefinition**: This element specifies the structure of the observable properties
2782 of the Policy Type by means of XML schema.

2783 The `PropertiesDefinition` element has one but not both of the following properties:

2784 o **element**: This attribute provides the QName of an XML element defining the structure
2785 of the Policy Type Properties.

2786 o **type**: This attribute provides the QName of an XML (complex) type defining the
2787 structure of the Policy Type Properties.

2788 • **AppliesTo**: This OPTIONAL element specifies the set of Node Types the Policy Type is
2789 applicable to, each defined as a separate, nested `NodeTypeReference` element.

2790 The `NodeTypeReference` element has the following property:

2791 o **typeRef**: The attribute provides the QName of a Node Type to which the Policy Type
2792 applies.

2793 14.3 Derivation Rules

2794 The following rules on combining definitions based on `DerivedFrom` apply:

2795 • **Properties Definitions**: It is assumed that the XML element (or type) representing the Policy Type
2796 Properties Definitions extends the XML element (or type) of the Policy Type Properties Definitions
2797 of the Policy Type referenced in the `DerivedFrom` element.

2798 • **Applies To**: The set of Node Types the Policy Type is applicable to consist of the set union of
2799 Node Types derived from and Node Types explicitly referenced by the Policy Type by means of
2800 its `AppliesTo` element.

2801 • **Policy Language**: A Policy Type MUST define the same policy language as the Policy Type it
2802 derives from. In case the Policy Type used as basis for derivation has no `policyLanguage`
2803 attribute defined, the deriving Policy Type can define any appropriate policy language.

2804 14.4 Example

2805 The following example defines two Policy Types, the “HighAvailability” Policy Type and the
2806 “ContinuousAvailability” Policy Type. They are defined in a Definitions document “MyPolicyTypes” within
2807 the target namespace “http://www.example.com/SamplePolicyTypes”. Thus, by importing the
2808 corresponding namespace into another Definitions document, both Policy Types are available for use in
2809 the other document.

```
2810 01 <Definitions id="MyPolicyTypes" name="My Policy Types"  
2811 02   targetNamespace="http://www.example.com/SamplePolicyTypes"  
2812 03   xmlns:bnt="http://www.example.com/BaseNodeTypes">  
2813 04   xmlns:spp="http://www.example.com/SamplePolicyProperties">  
2814 05  
2815 06   <Import importType="http://www.w3.org/2001/XMLSchema"  
2816 07     namespace="http://www.example.com/SamplePolicyProperties"/>  
2817 08  
2818 09   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"  
2819 10     namespace="http://www.example.com/BaseNodeTypes"/>  
2820 11  
2821 12  
2822 13   <PolicyType name="HighAvailability">  
2823 14     <PropertiesDefinition element="spp:HAProperties"/>  
2824 15   </PolicyType>  
2825 16  
2826 17   <PolicyType name="ContinuousAvailability">  
2827 18     <DerivedFrom typeRef="HighAvailability"/>  
2828 19     <PropertiesDefinition element="spp:CAProperties"/>  
2829 20     <AppliesTo>  
2830 21       <NodeTypeReference typeRef="bnt:DBMS"/>  
2831 22     </AppliesTo>  
2832 23   </PolicyType>  
2833 24  
2834 25 </Definitions>
```

2835 The Policy Type “HighAvailability” defined in the example above has the “HAProperties” properties that
2836 are defined in a separate namespace as an XML element. The same namespace contains the
2837 “CAProperties” element that defines the properties of the “ContinuousAvailability” Policy Type. This
2838 namespace is imported by means of the first `Import` element and the namespace of those imported
2839 definitions is assigned the prefix “spp” in the current file.

2840 The “ContinuousAvailability” Policy Type is derived from the “HighAvailability” Policy Type. Furthermore, it
2841 is applicable to the “DBMS” Node Type. This Node Type is defined in a separate namespace, which is
2842 imported by means of the second `Import` element and the namespace of those imported definitions is
2843 assigned the prefix “bnt” in the current file.

2844

15 Policy Templates

2845 This chapter specifies how *Policy Templates* are defined. A Policy Template represents a particular non-
2846 functional behavior or quality-of-service that can be referenced by a Node Template. A Policy Template
2847 refers to a specific Policy Type that defines the structure of observable properties (metadata) of the non-
2848 functional behavior. The Policy Template then typically defines values for those properties inside the
2849 *Properties* element. Note that properties defined by a Policy Template are assumed to be invariant
2850 across the contexts in which corresponding behavior is exposed – as opposed to properties defined in
2851 Policies of Node Templates that may vary depending on the context.

2852

15.1 XML Syntax

2853 The following pseudo schema defines the XML syntax of Policy Templates:

```
2854 01 <PolicyTemplate id="xs:ID" name="xs:string"? type="xs:QName">  
2855 02  
2856 03   <Properties>  
2857 04     XML fragment  
2858 05   </Properties> ?  
2859 06  
2860 07   <PropertyConstraints>  
2861 08     <PropertyConstraint property="xs:string"  
2862 09       constraintType="xs:anyURI"> +  
2863 10       constraint ?  
2864 11     </PropertyConstraint>  
2865 12   </PropertyConstraints> ?  
2866 13  
2867 14   policy type specific content ?  
2868 15  
2869 16 </PolicyTemplate>
```

2870

15.2 Properties

2871 The `PolicyTemplate` element has the following properties:

- 2872 • `id`: This attribute specifies the identifier of the Policy Template which **MUST** be unique within the
2873 target namespace.
- 2874 • `name`: This **OPTIONAL** attribute specifies the name of the Policy Template.
- 2875 • `type`: The QName value of this attribute refers to the Policy Type providing the type of the Policy
2876 Template.
- 2877 • `Properties`: This **OPTIONAL** element specifies the invariant properties of the Policy
2878 Template, i.e. those properties that will be commonly used across different contexts in which the
2879 Policy Template is used.

2880

2881 The initial values are specified by providing an instance document of the XML schema of the
2882 corresponding Policy Type Properties. This instance document considers the inheritance
2883 structure deduced by the `DerivedFrom` property of the Policy Type referenced by the `type`
2884 attribute of the Policy Template.

- 2885 • `PropertyConstraints`: This **OPTIONAL** element specifies constraints on the use of one or
2886 more of the Policy Type Properties of the Policy Type providing the property definitions for the
2887 Policy Template. Each constraint is specified by means of a separate nested
2888 `PropertyConstraint` element.

2889 The `PropertyConstraint` element has the following properties:

- 2890 ○ `property`: The string value of this property is an XPath expression pointing to the
2891 property within the Policy Type Properties document that is constrained within the context
2892 of the Policy Template. More than one constraint MUST NOT be defined for each
2893 property.
- 2894 ○ `constraintType`: The constraint type is specified by means of a URI, which defines
2895 both the semantic meaning of the constraint as well as the format of the content.

2896 15.3 Example

2897 The following example defines a Policy Template “MyHAPolicy”. It is defined in a Definitions document
2898 “MyPolicies” within the target namespace “http://www.example.com/SamplePolicies”. The Policy
2899 Template can be used in the same Definitions document, for example, as a Policy of some Node
2900 Template, or it can be used in other document by importing the corresponding namespace into the other
2901 document.

```
2902 01 <Definitions id="MyPolicies" name="My Policies"  
2903 02   targetNamespace="http://www.example.com/SamplePolicies"  
2904 03   xmlns:spt="http://www.example.com/SamplePolicyTypes">  
2905 04  
2906 05   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"  
2907 06     namespace="http://www.example.com/SamplePolicyTypes"/>  
2908 07  
2909 08   <PolicyTemplate id="MyHAPolicy"  
2910 09     name="My High Availability Policy"  
2911 10     type="bpt:HighAvailability">  
2912 11     <Properties>  
2913 12       <HAProperties>  
2914 13         <AvailabilityClass>4</AvailabilityClass>  
2915 14         <HeartbeatFrequency measuredIn="msec">  
2916 15           250  
2917 16         </HeartbeatFrequency>  
2918 17       </HAProperties>  
2919 18     </Properties>  
2920 19   </PolicyTemplate>  
2921 20  
2922 21 </Definitions>
```

2923 The Policy Template “MyHAPolicy” defined in the example above is of type “HighAvailability” that is
2924 specified in the `type` attribute of the `PolicyTemplate` element. This Policy Type is defined in a
2925 separate file, the definitions of which are imported by means of the `Import` element and the namespace
2926 of those imported definitions is assigned the prefix “spt” in the current file.

2927 The “MyHAPolicy” Policy Template provides values for the properties defined by the Properties Definition
2928 of the “HighAvailability” Policy Type. The `AvailabilityClass` property is set to “4”. The value of the
2929 `HeartbeatFrequency` is “250”, measured in “msec”.
2930

2931 16 Cloud Service Archive (CSAR)

2932 This section defines the metadata of a cloud service archive as well as its overall structure.

2933 16.1 Overall Structure of a CSAR

2934 A CSAR is a zip file containing at least two directories, the *TOSCA-Metadata* directory and the *Definitions*
2935 directory. Beyond that, other directories MAY be contained in a CSAR, i.e. the creator of a CSAR has all
2936 freedom to define the content of a CSAR and the structuring of this content as appropriate for the cloud
2937 application.

2938 The TOSCA-Metadata directory contains metadata describing the other content of the CSAR. This
2939 metadata is referred to as *TOSCA meta file*. This file is named `TOSCA` and has the file extension `.meta`.

2940 The Definitions directory contains one or more TOSCA Definitions documents (file extension `.tosca`).
2941 These Definitions files typically contain definitions related to the cloud application of the CSAR. In
2942 addition, CSARs can contain just the definition of elements for re-use in other contexts. For example, a
2943 CSAR might be used to package a set of Node Types and Relationship Types with their respective
2944 implementations that can then be used by Service Templates provided in other CSARs. In cases where a
2945 complete cloud application is packaged in a CSAR, one of the Definitions documents in the Definitions
2946 directory MUST contain a Service Template definition that defines the structure and behavior of the cloud
2947 application.

2948 16.2 TOSCA Meta File

2949 The TOSCA meta file includes metadata that allows interpreting the various artifacts within the CSAR
2950 properly. The `TOSCA.meta` file is contained in the `TOSCA-Metadata` directory of the CSAR.

2951 A TOSCA meta file consists of name/value pairs. The name-part of a name/value pair is followed by a
2952 colon, followed by a blank, followed by the value-part of the name/value pair. The name MUST NOT
2953 contain a colon. Values that represent binary data MUST be base64 encoded. Values that extend beyond
2954 one line can be spread over multiple lines if each subsequent line starts with at least one space. Such
2955 spaces are then collapsed when the value string is read.

```
2956 01 <name>: <value>
```

2957 Each name/value pair is in a separate line. A list of related name/value pairs, i.e. a list of consecutive
2958 name/value pairs describing a particular file in a CSAR, is called a *block*. Blocks are separated by an
2959 empty line. The first block, called *block_0*, is metadata about the CSAR itself. All other blocks represent
2960 metadata of files in the CSAR.

2961 The structure of `block_0` in the TOSCA meta file is as follows:

```
2962 01 TOSCA-Meta-File-Version: digit.digit  
2963 02 CSAR-Version: digit.digit  
2964 03 Created-By: string  
2965 04 Entry-Definitions: string ?
```

2966 The name/value pairs are as follows:

- 2967 • `TOSCA-Meta-File-Version`: This is the version number of the TOSCA meta file format.
2968 The value MUST be “1.0” in the current version of the TOSCA specification.
- 2969 • `CSAR-Version`: This is the version number of the CSAR specification. The value MUST be
2970 “1.0” in the current version of the TOSCA specification.
- 2971 • `Created-By`: The person or vendor, respectively, who created the CSAR.

- 2972
- `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.
- 2973
- 2974
- 2975
- 2976
- 2977
- 2978
- 2979
- 2980
- 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.

2981

2982

2983

2984

2985

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.

```
2986 01 Name: <path-name_1>
2987 02 Content-Type: type_1/subtype_1
2988 03 <name_11>: <value_11>
2989 04 <name_12>: <value_12>
2990 05 ...
2991 06 <name_1n>: <value_1n>
2992 07
2993 08 ...
2994 09
2995 10 Name: <path-name_k>
2996 11 Content-Type: type_k/subtype_k
2997 12 <name_k1>: <value_k1>
2998 13 <name_k2>: <value_k2>
2999 14 ...
3000 15 <name_km>: <value_km>
```

3001

The name/value pairs are as follows:

- `Name`: The pathname or pathname pattern of the file(s) or resources described within the actual CSAR.
- 3002
- 3003
- 3004
- 3005
- 3006
- 3007
- 3008
- 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.”.

3009

3010

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.

3011 16.3 Example

3012

3013

3014

3015

3016

3017

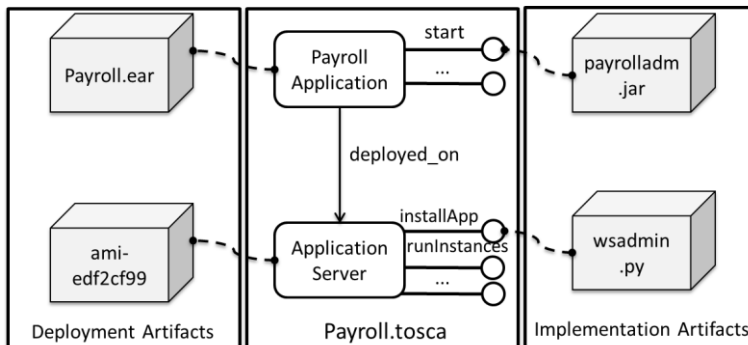
3018

3019

3020

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

3021 provided too; for example, the start operation of the Payroll Application is implemented by a
 3022 Java API supported by the payrolladm.jar file, the installApp operation of the Application
 3023 Server is realized by the Python script wsadmin.py, while the runInstances operation is a REST
 3024 API available at Amazon for running instances of an AMI. Note, that the runInstances operation is
 3025 not related to a particular implementation artifact because it is available as an Amazon Web Service
 3026 (<https://ec2.amazonaws.com/?Action=RunInstances>); but the details of this REST API are specified with
 3027 the operation of the Application Server Node Type.



3028
 3029 Figure 7: Sample Service Template

3030 The corresponding Node Types and Relationship Types have been defined in the
 3031 PayrollTypes.tosca document, which is imported by the Definitions document containing the
 3032 Payroll Service Template. The following listing provides some of the details:

```

3033 01 <Definitions id="PayrollDefinitions"
3034 02     targetNamespace="http://www.example.com/ste"
3035 03     xmlns:pay="http://www.example.com/ste/Types">
3036 04
3037 05     <Import namespace="http://www.example.com/ste/Types"
3038 06         location="http://www.example.com/ste/Types/PayrollTypes.tosca"
3039 07         importType=" http://docs.oasis-open.org/tosca/ns/2011/12"/>
3040 08
3041 09     <Types>
3042 10         ...
3043 11     </Types>
3044 12
3045 13     <ServiceTemplate id="Payroll" name="Payroll Service Template">
3046 14
3047 15         <TopologyTemplate ID="PayrollTemplate">
3048 16
3049 17             <NodeTemplate id="Payroll Application"
3050 18                 type="pay:ApplicationNodeType">
3051 19                 ...
3052 20
3053 21                 <DeploymentArtifacts>
3054 22                     <DeploymentArtifact name="PayrollEAR"
3055 23                         type="http://www.example.com/
3056 24                             ns/tosca/2011/12/
3057 25                             DeploymentArtifactTypes/CSARref">
3058 26                         EARs/Payroll.ear
3059 27                     </DeploymentArtifact>
3060 28                 </DeploymentArtifacts>
3061 29
3062 30             </NodeTemplate>
3063 31
3064 32             <NodeTemplate id="Application Server"
3065 33                 type="pay:ApplicationServerNodeType">
  
```

```

3066 34      ...
3067 35
3068 36      <DeploymentArtifacts>
3069 37          <DeploymentArtifact name="ApplicationServerImage"
3070 38              type="http://www.example.com/
3071 39                  ns/tosca/2011/12/
3072 40                      DeploymentArtifactTypes/AMIref">
3073 41              ami-edf2cf99
3074 42          </DeploymentArtifact>
3075 43      </DeploymentArtifacts>
3076 44
3077 45  </NodeTemplate>
3078 46
3079 47  <RelationshipTemplate id="deployed_on"
3080 48              type="pay:deployed_on">
3081 49      <SourceElement ref="Payroll Application"/>
3082 50      <TargetElement ref="Application Server"/>
3083 51  </RelationshipTemplate>
3084 52
3085 53  </TopologyTemplate>
3086 54
3087 55  </ServiceTemplate>
3088 56
3089 57 </Definitions>

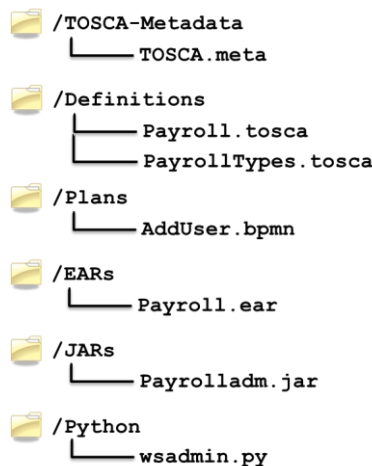
```

3090

3091 The Payroll Application Node Template specifies the deployment artifact PayrollEAR. It is a
3092 reference to the CSAR containing the Payroll.ste file, which is indicated by the .../CSARref type
3093 of the DeploymentArtifact element. The type specific content is a path expression in the directory
3094 structure of the CSAR: it points to the Payroll.ear file in the EARs directory of the CSAR (see Figure
3095 8 for the structure of the corresponding CSAR).

3096 The Application Server Node Template has a DeploymentArtifact called
3097 ApplicationServerImage that is a reference to an AMI (Amazon Machine Image), indicated by an
3098 .../AMIref type.

3099 The corresponding CSAR has the following structure (see Figure 8): The TOSCA.meta file is contained
3100 in the TOSCA-Metadata directory. The Payroll.ste file itself is contained in the Service-
3101 Template directory. Also, the PayrollTypes.ste file is in this directory. The content of the other
3102 directories has been sketched before.



3103

3104 Figure 8: Structure of CSAR Sample

3105 The TOSCA.meta file is as follows:

```
3106 01 TOSCA-Meta-Version: 1.0
3107 02 CSAR-Version: 1.0
3108 03 Created-By: Frank
3109 04
3110 05 Name: Service-Template/Payroll.tosca
3111 06 Content-Type: application/vnd.oasis.tosca.definitions
3112 07
3113 08 Name: Service-Template/PayrollTypes.ste
3114 09 Content-Type: application/vnd.oasis.tosca.definitions
3115 10
3116 11 Name: Plans/AddUser.bpmn
3117 12 Content-Type: application/vnd.oasis.bpmn
3118 13
3119 14 Name: EARs/Payroll.ear
3120 15 Content-Type: application/vnd.oasis.ear
3121 16
3122 17 Name: JARs/Payrolladm.jar
3123 18 Content-Type: application/vnd.oasis.jar
3124 19
3125 20 Name: Python/wsadmin.py
3126 21 Content-Type: application/vnd.oasis.py
3127
```

3128 **17 Security Considerations**

3129 TOSCA does not mandate the use of any specific mechanism or technology for client authentication.
3130 However, a client **MUST** provide a principal or the principal **MUST** be obtainable by the infrastructure.

3131 **18 Conformance**

3132 A TOSCA Definitions document conforms to this specification if it conforms to the TOSCA schema and
3133 follows the syntax and semantics defined in the normative portions of this specification. The TOSCA
3134 schema takes precedence over the TOSCA grammar (pseudo schema as defined in section 2.5), which
3135 in turn takes precedence over normative text, which in turn takes precedence over examples.

3136 An implementation conforms to this specification if it can process a conformant TOSCA Definitions
3137 document according to the rules described in chapters 4 through 16 of this specification.

3138 This specification allows extensions. Each implementation SHALL fully support all required functionality of
3139 the specification exactly as specified. The use of extensions SHALL NOT contradict nor cause the non-
3140 conformance of functionality defined in the specification.

3141 **Appendix A. Portability and Interoperability**
3142 **Considerations**

3143 This section illustrates the portability and interoperability aspects addressed by Service Templates:

3144 Portability - The ability to take Service Templates created in one vendor's environment and use them in
3145 another vendor's environment.

3146 Interoperability - The capability for multiple components (e.g. a task of a plan and the definition of a
3147 topology node) to interact using well-defined messages and protocols. This enables combining
3148 components from different vendors allowing seamless management of services.

3149 Portability demands support of TOSCA elements.

3150

Appendix B. Acknowledgements

3151 The following individuals have participated in the creation of this specification and are gratefully
3152 acknowledged.

3153 **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.

3154

Appendix C. Complete TOSCA Grammar

3156 **Note:** The following is a pseudo EBNF grammar notation meant for documentation purposes only. The
3157 grammar is not intended for machine processing.

```

3158 01 <Definitions id="xs:ID"
3159 02     name="xs:string"?
3160 03     targetNamespace="xs:anyURI">
3161 04
3162 05 <Extensions>
3163 06   <Extension namespace="xs:anyURI"
3164 07     mustUnderstand="yes|no"?/> +
3165 08 </Extensions> ?
3166 09
3167 10 <Import namespace="xs:anyURI"?
3168 11   location="xs:anyURI"?
3169 12   importType="xs:anyURI"/> *
3170 13
3171 14 <Types>
3172 15   <xs:schema .../> *
3173 16 </Types> ?
3174 17
3175 18 (
3176 19   <ServiceTemplate id="xs:ID"
3177 20     name="xs:string"?
3178 21     targetNamespace="xs:anyURI"
3179 22     substitutableNodeType="xs:QName"?>
3180 23
3181 24   <Tags>
3182 25     <Tag name="xs:string" value="xs:string"/> +
3183 26   </Tags> ?
3184 27
3185 28   <BoundaryDefinitions>
3186 29     <Properties>
3187 30       XML fragment
3188 31     <PropertyMappings>
3189 32       <PropertyMapping serviceTemplatePropertyRef="xs:string"
3190 33         targetObjectRef="xs:IDREF"
3191 34         targetPropertyRef="xs:IDREF"/> +
3192 35     </PropertyMappings/> ?
3193 36   </Properties> ?
3194 37
3195 38   <PropertyConstraints>
3196 39     <PropertyConstraint property="xs:string"
3197 40       constraintType="xs:anyURI"> +
3198 41     constraint ?
3199 42   </PropertyConstraint>
3200 43 </PropertyConstraints> ?
3201 44
3202 45   <Requirements>
3203 46     <Requirement name="xs:string" ref="xs:IDREF"/> +
3204 47   </Requirements> ?
3205 48
3206 49   <Capabilities>
3207 50     <Capability name="xs:string" ref="xs:IDREF"/> +
3208 51   </Capabilities> ?

```

```

3209 52
3210 53     <Policies>
3211 54         <Policy name="xs:string"? policyType="xs:QName"
3212 55             policyRef="xs:QName"?>
3213 56             policy specific content ?
3214 57         </Policy> +
3215 58     </Policies> ?
3216 59
3217 60     <Interfaces>
3218 61         <Interface name="xs:NCName">
3219 62             <Operation name="xs:NCName">
3220 63                 (
3221 64                     <NodeOperation nodeRef="xs:IDREF"
3222 65                         interfaceName="xs:anyURI"
3223 66                         operationName="xs:NCName"/>
3224 67                 |
3225 68                     <RelationshipOperation relationshipRef="xs:IDREF"
3226 69                         interfaceName="xs:anyURI"
3227 70                         operationName="xs:NCName"/>
3228 71                 |
3229 72                     <Plan planRef="xs:IDREF"/>
3230 73                 )
3231 74             </Operation> +
3232 75         </Interface> +
3233 76     </Interfaces> ?
3234 77
3235 78 </BoundaryDefinitions> ?
3236 79
3237 80 <TopologyTemplate>
3238 81     (
3239 82         <NodeTemplate id="xs:ID" name="xs:string"? type="xs:QName"
3240 83             minInstances="xs:integer"?
3241 84             maxInstances="xs:integer | xs:string"?>
3242 85             <Properties>
3243 86                 XML fragment
3244 87             </Properties> ?
3245 88
3246 89             <PropertyConstraints>
3247 90                 <PropertyConstraint property="xs:string"
3248 91                     constraintType="xs:anyURI">
3249 92                     constraint ?
3250 93                 </PropertyConstraint> +
3251 94             </PropertyConstraints> ?
3252 95
3253 96             <Requirements>
3254 97                 <Requirement id="xs:ID" name="xs:string" type="xs:QName"> +
3255 98                     <Properties>
3256 99                         XML fragment
3257 100                     <Properties> ?
3258 101                     <PropertyConstraints>
3259 102                         <PropertyConstraint property="xs:string"
3260 103                             constraintType="xs:anyURI"> +
3261 104                             constraint ?
3262 105                         </PropertyConstraint>
3263 106                     </PropertyConstraints> ?
3264 107                 </Requirement>
3265 108             </Requirements> ?
3266 109

```

```

3267 110      <Capabilities>
3268 111          <Capability id="xs:ID" name="xs:string"
3269 112              type="xs:QName"> +
3270 113              <Properties>
3271 114                  XML fragment
3272 115              <Properties> ?
3273 116              <PropertyConstraints>
3274 117                  <PropertyConstraint property="xs:string"
3275 118                      constraintType="xs:anyURI">
3276 119                      constraint ?
3277 120                  </PropertyConstraint> +
3278 121              </PropertyConstraints> ?
3279 122          </Capability>
3280 123      </Capabilities> ?
3281 124
3282 125      <Policies>
3283 126          <Policy name="xs:string"? policyType="xs:QName"
3284 127              policyRef="xs:QName"?>
3285 128              policy specific content ?
3286 129          </Policy> +
3287 130      </Policies> ?
3288 131
3289 132      <DeploymentArtifacts>
3290 133          <DeploymentArtifact name="xs:string"
3291 134              artifactType="xs:QName"
3292 135              artifactRef="xs:QName"?>
3293 136              artifact specific content ?
3294 137          </DeploymentArtifact> +
3295 138      </DeploymentArtifacts> ?
3296 139  </NodeTemplate>
3297 140  |
3298 141  <RelationshipTemplate id="xs:ID" name="xs:string"?
3299 142              type="xs:QName">
3300 143      <Properties>
3301 144          XML fragment
3302 145      </Properties> ?
3303 146
3304 147      <PropertyConstraints>
3305 148          <PropertyConstraint property="xs:string"
3306 149              constraintType="xs:anyURI">
3307 150              constraint ?
3308 151          </PropertyConstraint> +
3309 152      </PropertyConstraints> ?
3310 153
3311 154      <SourceElement ref="xs:IDREF"/>
3312 155      <TargetElement ref="xs:IDREF"/>
3313 156
3314 157      <RelationshipConstraints>
3315 158          <RelationshipConstraint constraintType="xs:anyURI">
3316 159              constraint ?
3317 160          </RelationshipConstraint> +
3318 161      </RelationshipConstraints> ?
3319 162
3320 163      </RelationshipTemplate>
3321 164  ) +
3322 165  </TopologyTemplate>
3323 166
3324 167  <Plans>

```

```

3325 168     <Plan id="xs:ID"
3326 169         name="xs:string"?
3327 170         planType="xs:anyURI"
3328 171         planLanguage="xs:anyURI">
3329 172
3330 173         <PreEcondition expressionLanguage="xs:anyURI">
3331 174             condition
3332 175         </PreEcondition> ?
3333 176
3334 177         <InputParameters>
3335 178             <InputParameter name="xs:string" type="xs:string"
3336 179                 required="yes|no"?/> +
3337 180         </InputParameters> ?
3338 181
3339 182         <OutputParameters>
3340 183             <OutputParameter name="xs:string" type="xs:string"
3341 184                 required="yes|no"?/> +
3342 185         </OutputParameters> ?
3343 186
3344 187         (
3345 188             <PlanModel>
3346 189                 actual plan
3347 190             </PlanModel>
3348 191             |
3349 192             <PlanModelReference reference="xs:anyURI"/>
3350 193         )
3351 194
3352 195     </Plan> +
3353 196 </Plans> ?
3354 197
3355 198 </ServiceTemplate>
3356 199 |
3357 200 <NodeType name="xs:NCName" targetNamespace="xs:anyURI"?
3358 201     abstract="yes|no"? final="yes|no"?>
3359 202
3360 203     <DerivedFrom typeRef="xs:QName"/> ?
3361 204
3362 205     <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
3363 206
3364 207     <RequirementDefinitions>
3365 208         <RequirementDefinition name="xs:string"
3366 209             requirementType="xs:QName"
3367 210             lowerBound="xs:integer"?
3368 211             upperBound="xs:integer | xs:string"?>
3369 212             <Constraints>
3370 213                 <Constraint constraintType="xs:anyURI">
3371 214                     constraint type specific content
3372 215                 </Constraint> +
3373 216             </Constraints> ?
3374 217         </RequirementDefinition> +
3375 218     </RequirementDefinitions> ?
3376 219
3377 220     <CapabilityDefinitions>
3378 221         <CapabilityDefinition name="xs:string"
3379 222             capabilityType="xs:QName"
3380 223             lowerBound="xs:integer"?
3381 224             upperBound="xs:integer | xs:string"?>
3382 225     </CapabilityDefinition>
3382 225     <Constraints>

```



```

3383 226         <Constraint constraintType="xs:anyURI">
3384 227             constraint type specific content
3385 228         </Constraint> +
3386 229     </Constraints> ?
3387 230     </CapabilityDefinition> +
3388 231 </CapabilityDefinitions>
3389 232
3390 233 <InstanceStates>
3391 234     <InstanceState state="xs:anyURI"> +
3392 235 </InstanceState> ?
3393 236
3394 237 <Interfaces>
3395 238     <Interface name="xs:NCName | xs:anyURI">
3396 239         <Operation name="xs:NCName">
3397 240             <InputParameters>
3398 241                 <InputParameter name="xs:string" type="xs:string"
3399 242                     required="yes|no"?/> +
3400 243             </InputParameters> ?
3401 244             <OutputParameters>
3402 245                 <OutputParameter name="xs:string" type="xs:string"
3403 246                     required="yes|no"?/> +
3404 247             </OutputParameters> ?
3405 248         </Operation> +
3406 249     </Interface> +
3407 250 </Interfaces> ?
3408 251
3409 252 </NodeType>
3410 253 |
3411 254 <NodeTypeImplementation name="xs:NCName"
3412 255     targetNamespace="xs:anyURI"?
3413 256     nodeType="xs:QName"
3414 257     abstract="yes|no"?
3415 258     final="yes|no"?>
3416 259
3417 260 <DerivedFrom nodeTypeImplementationRef="xs:QName"/> ?
3418 261
3419 262 <RequiredContainerFeatures>
3420 263     <RequiredContainerFeature feature="xs:anyURI"/> +
3421 264 </RequiredContainerFeatures> ?
3422 265
3423 266 <ImplementationArtifacts>
3424 267     <ImplementationArtifact name="xs:string"
3425 268 interfaceName="xs:NCName |
3426 269 xs:anyURI"?
3427 270 operationName="xs:NCName"?
3428 271 artifactType="xs:QName"
3429 272 artifactRef="xs:QName"??
3430 273     artifact specific content ?
3431 274 <ImplementationArtifact> +
3432 275 </ImplementationArtifacts> ?
3433 276
3434 277 <DeploymentArtifacts>
3435 278     <DeploymentArtifact name="xs:string"
3436 279     artifactType="xs:QName"
3437 280     artifactRef="xs:QName"??>
3438 281     artifact specific content ?
3439 282 <DeploymentArtifact> +
3440 283 </DeploymentArtifacts> ?

```

```

3441 282281
3442 283282 </NodeTypeImplementation>
3443 284283 |
3444 285284 <RelationshipType name="xs:NCName"
3445 286285 targetNamespace="xs:anyURI"?
3446 287286 abstract="yes|no"?
3447 288287 final="yes|no"?> +
3448 289288
3449 290289 <DerivedFrom typeRef="xs:QName"/> ?
3450 291290
3451 292291 <PropertiesDefinition element="xs:QName"?
3452 293292 type="xs:QName"?/> ?
3453 294293
3454 295294 <InstanceStates>
3455 296295 <InstanceState state="xs:anyURI"> +
3456 297296 </InstanceStates> ?
3457 298297
3458 299298 <SourceInterfaces>
3459 300299 <Interface name="xs:NCName | xs:anyURI">
3460 301300 <Operation name="xs:NCName">
3461 302301 <InputParameters>
3462 303302 <InputParameter name="xs:string" type="xs:string"
3463 304303 required="yes|no"?/> +
3464 305304 </InputParameters> ?
3465 306305 <OutputParameters>
3466 307306 <OutputParameter name="xs:string" type="xs:string"
3467 308307 required="yes|no"?/> +
3468 309308 </OutputParameters> ?
3469 310309 </Operation> +
3470 311310 </Interface> +
3471 312311 </SourceInterfaces> ?
3472 313312
3473 314313 <TargetInterfaces>
3474 315314 <Interface name="xs:NCName | xs:anyURI">
3475 316315 <Operation name="xs:NCName">
3476 317316 <InputParameters>
3477 318317 <InputParameter name="xs:string" type="xs:string"
3478 319318 required="yes|no"?/> +
3479 320319 </InputParameters> ?
3480 321320 <OutputParameters>
3481 322321 <OutputParameter name="xs:string" type="xs:string"
3482 323322 required="yes|no"?/> +
3483 324323 </OutputParameters> ?
3484 325324 </Operation> +
3485 326325 </Interface> +
3486 327326 </TargetInterfaces> ?
3487 328327
3488 329328 <ValidSource typeRef="xs:QName"/> ?
3489 330329
3490 331330 <ValidTarget typeRef="xs:QName"/> ?
3491 332331
3492 333332 </RelationshipType>
3493 334333 |
3494 335334 <RelationshipTypeImplementation name="xs:NCName"
3495 336335 targetNamespace="xs:anyURI"?
3496 337336 relationshipType="xs:QName"
3497 338337 abstract="yes|no"?
3498 339338 final="yes|no"?>

```

```

3499 | 339338
3500 | 340339      <DerivedFrom relationshipTypeImplementationRef="xs:QName"/>
3501 |   ?
3502 | 341340
3503 | 342341      <RequiredContainerFeatures>
3504 | 343342      <RequiredContainerFeature feature="xs:anyURI"/> +
3505 | 344343      </RequiredContainerFeatures> ?
3506 | 345344
3507 | 346345      <ImplementationArtifacts>
3508 | 347         <ImplementationArtifact name="xs:string"
3509 | 348346         interfaceName="xs:NCName |
3510 |   xs:anyURI"?
3511 | 349347         operationName="xs:NCName"?
3512 | 350348         artifactType="xs:QName"
3513 | 351349         artifactRef="xs:QName"?>
3514 | 352350      artifact specific content ?
3515 | 353351      <ImplementationArtifact> +
3516 | 354352      </ImplementationArtifacts> ?
3517 | 355353
3518 | 356354      </RelationshipTypeImplementation>
3519 | 357355      |
3520 | 358356      <RequirementType name="xs:NCName"
3521 | 359357         targetNamespace="xs:anyURI"?
3522 | 360358         abstract="yes|no"?
3523 | 361359         final="yes|no"?
3524 | 362360         requiredCapabilityType="xs:QName"?>
3525 | 363361
3526 | 364362      <DerivedFrom typeRef="xs:QName"/> ?
3527 | 365363
3528 | 366364      <PropertiesDefinition element="xs:QName"?
3529 |   type="xs:QName"?/> ?
3530 | 367365
3531 | 368366      </RequirementType>
3532 | 369367      |
3533 | 370368      <CapabilityType name="xs:NCName"
3534 | 371369         targetNamespace="xs:anyURI"?
3535 | 372370         abstract="yes|no"?
3536 | 373371         final="yes|no"?>
3537 | 374372
3538 | 375373      <DerivedFrom typeRef="xs:QName"/> ?
3539 | 376374
3540 | 377375      <PropertiesDefinition element="xs:QName"?
3541 |   type="xs:QName"?/> ?
3542 | 378376
3543 | 379377      </CapabilityType>
3544 | 380378      |
3545 | 381379      <ArtifactType name="xs:NCName"
3546 | 382380         targetNamespace="xs:anyURI"?
3547 | 383381         abstract="yes|no"?
3548 | 384382         final="yes|no"?>
3549 | 385383
3550 | 386384      <DerivedFrom typeRef="xs:QName"/> ?
3551 | 387385
3552 | 388386      <PropertiesDefinition element="xs:QName"?
3553 |   type="xs:QName"?/> ?
3554 | 389387
3555 | 390388      </ArtifactType>
3556 | 391389      |

```

```

3557 | 392390      <ArtifactTemplate id="xs:ID" name="xs:string"?
3558 |         type="xs:QName">
3559 | 393391
3560 | 394392      <Properties>
3561 | 395393      XML fragment
3562 | 396394      </Properties> ?
3563 | 397395
3564 | 398396      <PropertyConstraints>
3565 | 399397      <PropertyConstraint property="xs:string"
3566 | 400398      constraintType="xs:anyURI"> +
3567 | 401399      constraint ?
3568 | 402400      </PropertyConstraint>
3569 | 403401      </PropertyConstraints> ?
3570 | 404402
3571 | 405403      <ArtifactReferences>
3572 | 406404      <ArtifactReference reference="xs:anyURI">
3573 | 407405      (
3574 | 408406      <Include pattern="xs:string"/>
3575 | 409407      |
3576 | 410408      <Exclude pattern="xs:string"/>
3577 | 411409      ) *
3578 | 412410      </ArtifactReference> +
3579 | 413411      </ArtifactReferences> ?
3580 | 414412
3581 | 415413      </ArtifactTemplate>
3582 | 416414      |
3583 | 417415      <PolicyType name="xs:NCName"
3584 | 418416      policyLanguage="xs:anyURI"?
3585 | 419417      abstract="yes|no"?
3586 | 420418      final="yes|no"?
3587 | 421419      targetNamespace="xs:anyURI"?>
3588 | 422420      <Tags>
3589 | 423421      <Tag name="xs:string" value="xs:string"/> +
3590 | 424422      </Tags> ?
3591 | 425423
3592 | 426424      <DerivedFrom typeRef="xs:QName"/> ?
3593 | 427425
3594 | 428426      <PropertiesDefinition element="xs:QName"?
3595 |         type="xs:QName"?/> ?
3596 | 429427
3597 | 430428      <AppliesTo>
3598 | 431429      <NodeTypeReference typeRef="xs:QName"/> +
3599 | 432430      </AppliesTo> ?
3600 | 433431
3601 | 434432      policy type specific content ?
3602 | 435433
3603 | 436434      </PolicyType>
3604 | 437435      |
3605 | 438436      <PolicyTemplate id="xs:ID" name="xs:string"? type="xs:QName">
3606 | 439437
3607 | 440438      <Properties>
3608 | 441439      XML fragment
3609 | 442440      </Properties> ?
3610 | 443441
3611 | 444442      <PropertyConstraints>
3612 | 445443      <PropertyConstraint property="xs:string"
3613 | 446444      constraintType="xs:anyURI"> +
3614 | 447445      constraint ?

```

```
3615 | 448446          </PropertyConstraint>
3616 | 449447          </PropertyConstraints> ?
3617 | 450448
3618 | 451449          policy type specific content ?
3619 | 452450
3620 | 453451          </PolicyTemplate>
3621 | 454452          ) +
3622 | 455453
3623 | 456454          </Definitions>
```

Appendix D. TOSCA Schema

3625 TOSCA-v1.0.xsd:

```
3626 01 <?xml version="1.0" encoding="UTF-8"?>
3627 02 <xs:schema targetNamespace="http://docs.oasis-open.org/tosca/ns/2011/12"
3628 03   elementFormDefault="qualified" attributeFormDefault="unqualified"
3629 04   xmlns="http://docs.oasis-open.org/tosca/ns/2011/12"
3630 05   xmlns:xs="http://www.w3.org/2001/XMLSchema">
3631 06
3632 07   <xs:import namespace="http://www.w3.org/XML/1998/namespace"
3633 08     schemaLocation="http://www.w3.org/2001/xml.xsd"/>
3634 09
3635 10   <xs:element name="documentation" type="tDocumentation"/>
3636 11   <xs:complexType name="tDocumentation" mixed="true">
3637 12     <xs:sequence>
3638 13       <xs:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
3639 14     </xs:sequence>
3640 15     <xs:attribute name="source" type="xs:anyURI"/>
3641 16     <xs:attribute ref="xml:lang"/>
3642 17   </xs:complexType>
3643 18
3644 19   <xs:complexType name="tExtensibleElements">
3645 20     <xs:sequence>
3646 21       <xs:element ref="documentation" minOccurs="0" maxOccurs="unbounded"/>
3647 22       <xs:any namespace="##other" processContents="lax" minOccurs="0"
3648 23         maxOccurs="unbounded"/>
3649 24     </xs:sequence>
3650 25     <xs:anyAttribute namespace="##other" processContents="lax"/>
3651 26   </xs:complexType>
3652 27
3653 28   <xs:complexType name="tImport">
3654 29     <xs:complexContent>
3655 30       <xs:extension base="tExtensibleElements">
3656 31         <xs:attribute name="namespace" type="xs:anyURI"/>
3657 32         <xs:attribute name="location" type="xs:anyURI"/>
3658 33         <xs:attribute name="importType" type="importedURI" use="required"/>
3659 34       </xs:extension>
3660 35     </xs:complexContent>
3661 36   </xs:complexType>
3662 37
3663 38   <xs:element name="Definitions">
3664 39     <xs:complexType>
3665 40       <xs:complexContent>
3666 41         <xs:extension base="tDefinitions"/>
3667 42       </xs:complexContent>
3668 43     </xs:complexType>
3669 44   </xs:element>
3670 45   <xs:complexType name="tDefinitions">
3671 46     <xs:complexContent>
3672 47       <xs:extension base="tExtensibleElements">
3673 48         <xs:sequence>
3674 49           <xs:element name="Extensions" minOccurs="0">
3675 50             <xs:complexType>
3676 51               <xs:sequence>
3677 52                 <xs:element name="Extension" type="tExtension"
```

```

3678 53         maxOccurs="unbounded"/>
3679 54     </xs:sequence>
3680 55 </xs:complexType>
3681 56 </xs:element>
3682 57 <xs:element name="Import" type="tImport" minOccurs="0"
3683 58     maxOccurs="unbounded"/>
3684 59 <xs:element name="Types" minOccurs="0">
3685 60     <xs:complexType>
3686 61         <xs:sequence>
3687 62             <xs:any namespace="##other" processContents="lax" minOccurs="0"
3688 63                 maxOccurs="unbounded"/>
3689 64         </xs:sequence>
3690 65     </xs:complexType>
3691 66 </xs:element>
3692 67 <xs:choice maxOccurs="unbounded">
3693 68     <xs:element name="ServiceTemplate" type="tServiceTemplate"/>
3694 69     <xs:element name="NodeType" type="tNodeType"/>
3695 70     <xs:element name="NodeTypeImplementation"
3696 71         type="tNodeTypeImplementation"/>
3697 72     <xs:element name="RelationshipType" type="tRelationshipType"/>
3698 73     <xs:element name="RelationshipTypeImplementation"
3699 74         type="tRelationshipTypeImplementation"/>
3700 75     <xs:element name="RequirementType" type="tRequirementType"/>
3701 76     <xs:element name="CapabilityType" type="tCapabilityType"/>
3702 77     <xs:element name="ArtifactType" type="tArtifactType"/>
3703 78     <xs:element name="ArtifactTemplate" type="tArtifactTemplate"/>
3704 79     <xs:element name="PolicyType" type="tPolicyType"/>
3705 80     <xs:element name="PolicyTemplate" type="tPolicyTemplate"/>
3706 81 </xs:choice>
3707 82 </xs:sequence>
3708 83 <xs:attribute name="id" type="xs:ID" use="required"/>
3709 84 <xs:attribute name="name" type="xs:string" use="optional"/>
3710 85 <xs:attribute name="targetNamespace" type="xs:anyURI" use="required"/>
3711 86 </xs:extension>
3712 87 </xs:complexContent>
3713 88 </xs:complexType>
3714 89
3715 90 <xs:complexType name="tServiceTemplate">
3716 91     <xs:complexContent>
3717 92         <xs:extension base="tExtensibleElements">
3718 93             <xs:sequence>
3719 94                 <xs:element name="Tags" type="tTags" minOccurs="0"/>
3720 95                 <xs:element name="BoundaryDefinitions" type="tBoundaryDefinitions"
3721 96                     minOccurs="0"/>
3722 97                 <xs:element name="TopologyTemplate" type="tTopologyTemplate"/>
3723 98                 <xs:element name="Plans" type="tPlans" minOccurs="0"/>
3724 99             </xs:sequence>
3725 100             <xs:attribute name="id" type="xs:ID" use="required"/>
3726 101             <xs:attribute name="name" type="xs:string" use="optional"/>
3727 102             <xs:attribute name="targetNamespace" type="xs:anyURI"/>
3728 103             <xs:attribute name="substitutableNodeType" type="xs:QName"
3729 104                 use="optional"/>
3730 105             </xs:extension>
3731 106         </xs:complexContent>
3732 107     </xs:complexType>
3733 108
3734 109 <xs:complexType name="tTags">
3735 110     <xs:sequence>

```

```

3736 111     <xs:element name="Tag" type="tTag" maxOccurs="unbounded"/>
3737 112     </xs:sequence>
3738 113 </xs:complexType>
3739 114
3740 115 <xs:complexType name="tTag">
3741 116     <xs:attribute name="name" type="xs:string" use="required"/>
3742 117     <xs:attribute name="value" type="xs:string" use="required"/>
3743 118 </xs:complexType>
3744 119
3745 120 <xs:complexType name="tBoundaryDefinitions">
3746 121     <xs:sequence>
3747 122         <xs:element name="Properties" minOccurs="0">
3748 123             <xs:complexType>
3749 124                 <xs:sequence>
3750 125                     <xs:any namespace="##other"/>
3751 126                     <xs:element name="PropertyMappings" minOccurs="0">
3752 127                         <xs:complexType>
3753 128                             <xs:sequence>
3754 129                                 <xs:element name="PropertyMapping" type="tPropertyMapping"/>
3755 130                                 maxOccurs="unbounded"/>
3756 130131                                     </xs:sequence>
3757 131132                                     </xs:complexType>
3758 132133                                     </xs:element>
3759 133134                                     </xs:sequence>
3760 134135                                     </xs:complexType>
3761 135136                                     </xs:element>
3762 136137                                     <xs:element name="PropertyConstraints" minOccurs="0">
3763 137138                                         <xs:complexType>
3764 138139                                         <xs:sequence>
3765 139140                                             <xs:element name="PropertyConstraint"
3766 type="tPropertyConstraint"
3767 maxOccurs="unbounded"/>
3768 141142                                     </xs:sequence>
3769 142143                                     </xs:complexType>
3770 143144                                     </xs:element>
3771 144145                                     <xs:element name="Requirements" minOccurs="0">
3772 145146                                         <xs:complexType>
3773 146147                                             <xs:sequence>
3774 147148                                                 <xs:element name="Requirement" type="tRequirementRef"
3775 148149                                                     maxOccurs="unbounded"/>
3776 149150                                                     </xs:sequence>
3777 150151                                                     </xs:complexType>
3778 151152                                                     </xs:element>
3779 152153                                     <xs:element name="Capabilities" minOccurs="0">
3780 153154                                         <xs:complexType>
3781 154155                                             <xs:sequence>
3782 155156                                                 <xs:element name="Capability" type="tCapabilityRef"
3783 156157                                                     maxOccurs="unbounded"/>
3784 157158                                                     </xs:sequence>
3785 158159                                                     </xs:complexType>
3786 159160                                                     </xs:element>
3787 160161                                     <xs:element name="Policies" minOccurs="0">
3788 161162                                         <xs:complexType>
3789 162163                                             <xs:sequence>
3790 163164                                                 <xs:element name="Policy" type="tPolicy"
3791 maxOccurs="unbounded"/>
3792 164165                                                     </xs:sequence>
3793 165166                                                     </xs:complexType>

```



```

3794 | 166167      </xs:element>
3795 | 167168      <xs:element name="Interfaces" minOccurs="0">
3796 | 168169      <xs:complexType>
3797 | 169170      <xs:sequence>
3798 | 170171      <xs:element name="Interface" type="tExportedInterface"
3799 | 171172      maxOccurs="unbounded"/>
3800 | 172173      </xs:sequence>
3801 | 173174      </xs:complexType>
3802 | 174175      </xs:element>
3803 | 175176      </xs:sequence>
3804 | 176177      </xs:complexType>
3805 | 177178
3806 | 178179      <xs:complexType name="tPropertyMapping">
3807 | 179180      <xs:attribute name="serviceTemplatePropertyRef"
3808 |         type="xs:string"
3809 | 180181      use="required"/>
3810 | 181182      <xs:attribute name="targetObjectRef" type="xs:IDREF"
3811 |         use="required"/>
3812 | 182183      <xs:attribute name="targetPropertyRef" type="xs:string"
3813 |         use="required"/>
3814 | 184185      </xs:complexType>
3815 | 185186
3816 | 186187      <xs:complexType name="tRequirementRef">
3817 | 187188      <xs:attribute name="name" type="xs:string" use="optional"/>
3818 | 188189      <xs:attribute name="ref" type="xs:IDREF" use="required"/>
3819 | 189190      </xs:complexType>
3820 | 190191
3821 | 191192      <xs:complexType name="tCapabilityRef">
3822 | 192193      <xs:attribute name="name" type="xs:string" use="optional"/>
3823 | 193194      <xs:attribute name="ref" type="xs:IDREF" use="required"/>
3824 | 194195      </xs:complexType>
3825 | 195196
3826 | 196197      <xs:complexType name="tEntityType" abstract="true">
3827 | 197198      <xs:complexContent>
3828 | 198199      <xs:extension base="tExtensibleElements">
3829 | 199200      <xs:sequence>
3830 | 200201      <xs:element name="Tags" type="tTags" minOccurs="0"/>
3831 | 201202      <xs:element name="DerivedFrom" minOccurs="0">
3832 | 202203      <xs:complexType>
3833 | 203204      <xs:attribute name="typeRef" type="xs:QName"
3834 |         use="required"/>
3835 | 204205      </xs:complexType>
3836 | 205206      </xs:element>
3837 | 206207      <xs:element name="PropertiesDefinition" minOccurs="0">
3838 | 207208      <xs:complexType>
3839 | 208209      <xs:attribute name="element" type="xs:QName"/>
3840 | 209210      <xs:attribute name="type" type="xs:QName"/>
3841 | 210211      </xs:complexType>
3842 | 211212      </xs:element>
3843 | 212213      </xs:sequence>
3844 | 213214      <xs:attribute name="name" type="xs:NCName" use="required"/>
3845 | 214215      <xs:attribute name="abstract" type="tBoolean" default="no"/>
3846 | 215216      <xs:attribute name="final" type="tBoolean" default="no"/>
3847 | 216217      <xs:attribute name="targetNamespace" type="xs:anyURI"
3848 |         use="optional"/>
3849 | 218219      </xs:extension>
3850 | 219220      </xs:complexContent>
3851 | 220221      </xs:complexType>

```

```

3852 221222
3853 222223 <xs:complexType name="tEntityTemplate" abstract="true">
3854 223224   <xs:complexContent>
3855 224225     <xs:extension base="tExtensibleElements">
3856 225226       <xs:sequence>
3857 226227         <xs:element name="Properties" minOccurs="0">
3858 227228           <xs:complexType>
3859 228229             <xs:sequence>
3860 229230               <xs:any namespace="##other" processContents="lax"/>
3861 230231             </xs:sequence>
3862 231232           </xs:complexType>
3863 232233         </xs:element>
3864 233234       <xs:element name="PropertyConstraints" minOccurs="0">
3865 234235         <xs:complexType>
3866 235236           <xs:sequence>
3867 236237             <xs:element name="PropertyConstraint"
3868 237238               type="tPropertyConstraint" maxOccurs="unbounded"/>
3869 238239           </xs:sequence>
3870 239240         </xs:complexType>
3871 240241       </xs:element>
3872 241242     </xs:sequence>
3873 242243     <xs:attribute name="id" type="xs:ID" use="required"/>
3874 243244     <xs:attribute name="type" type="xs:QName" use="required"/>
3875 244245   </xs:extension>
3876 245246 </xs:complexContent>
3877 246247 </xs:complexType>
3878 247248
3879 248249 <xs:complexType name="tNodeTemplate">
3880 249250   <xs:complexContent>
3881 250251     <xs:extension base="tEntityTemplate">
3882 251252       <xs:sequence>
3883 252253         <xs:element name="Requirements" minOccurs="0">
3884 253254           <xs:complexType>
3885 254255             <xs:sequence>
3886 255256               <xs:element name="Requirement" type="tRequirement"
3887 256257                 maxOccurs="unbounded"/>
3888 257258             </xs:sequence>
3889 258259           </xs:complexType>
3890 259260         </xs:element>
3891 260261         <xs:element name="Capabilities" minOccurs="0">
3892 261262           <xs:complexType>
3893 262263             <xs:sequence>
3894 263264               <xs:element name="Capability" type="tCapability"
3895 264265                 maxOccurs="unbounded"/>
3896 265266             </xs:sequence>
3897 266267           </xs:complexType>
3898 267268         </xs:element>
3899 268269         <xs:element name="Policies" minOccurs="0">
3900 269270           <xs:complexType>
3901 270271             <xs:sequence>
3902 271272               <xs:element name="Policy" type="tPolicy"
3903 272273                 maxOccurs="unbounded"/>
3904 273274             </xs:sequence>
3905 274275           </xs:complexType>
3906 275276         </xs:element>
3907 276277         <xs:element name="DeploymentArtifacts"
3908 277278           type="tDeploymentArtifacts"
3909           minOccurs="0"/>

```

```

3910 | 278279      </xs:sequence>
3911 | 279280      <xs:attribute name="name" type="xs:string" use="optional"/>
3912 | 280281      <xs:attribute name="minInstances" type="xs:int"
3913 |           use="optional"
3914 | 281282      default="1"/>
3915 | 282283      <xs:attribute name="maxInstances" use="optional" default="1">
3916 | 283284      <xs:simpleType>
3917 | 284285      <xs:union>
3918 | 285286      <xs:simpleType>
3919 | 286287      <xs:restriction base="xs:nonNegativeInteger">
3920 | 287288      <xs:pattern value="([1-9]+[0-9]*)"/>
3921 | 288289      </xs:restriction>
3922 | 289290      </xs:simpleType>
3923 | 290291      <xs:simpleType>
3924 | 291292      <xs:restriction base="xs:string">
3925 | 292293      <xs:enumeration value="unbounded"/>
3926 | 293294      </xs:restriction>
3927 | 294295      </xs:simpleType>
3928 | 295296      </xs:union>
3929 | 296297      </xs:simpleType>
3930 | 297298      </xs:attribute>
3931 | 298299      </xs:extension>
3932 | 299300      </xs:complexContent>
3933 | 300301      </xs:complexType>
3934 | 301302
3935 | 302303      <xs:complexType name="tTopologyTemplate">
3936 | 303304      <xs:complexContent>
3937 | 304305      <xs:extension base="tExtensibleElements">
3938 | 305306      <xs:choice maxOccurs="unbounded">
3939 | 306307      <xs:element name="NodeTemplate" type="tNodeTemplate"/>
3940 | 307308      <xs:element name="RelationshipTemplate"
3941 | 308309      type="tRelationshipTemplate"/>
3942 | 309310      </xs:choice>
3943 | 310311      </xs:extension>
3944 | 311312      </xs:complexContent>
3945 | 312313      </xs:complexType>
3946 | 313314
3947 | 314315      <xs:complexType name="tRelationshipType">
3948 | 315316      <xs:complexContent>
3949 | 316317      <xs:extension base="tEntityType">
3950 | 317318      <xs:sequence>
3951 | 318319      <xs:element name="InstanceStates"
3952 | 319320      type="tTopologyElementInstanceStates" minOccurs="0"/>
3953 | 320321      <xs:element name="SourceInterfaces" minOccurs="0">
3954 | 321322      <xs:complexType>
3955 | 322323      <xs:sequence>
3956 | 323324      <xs:element name="Interface" type="tInterface"
3957 | 324325      maxOccurs="unbounded"/>
3958 | 325326      </xs:sequence>
3959 | 326327      </xs:complexType>
3960 | 327328      </xs:element>
3961 | 328329      <xs:element name="TargetInterfaces" minOccurs="0">
3962 | 329330      <xs:complexType>
3963 | 330331      <xs:sequence>
3964 | 331332      <xs:element name="Interface" type="tInterface"
3965 | 332333      maxOccurs="unbounded"/>
3966 | 333334      </xs:sequence>
3967 | 334335      </xs:complexType>

```

```

3968 | 335336      </xs:element>
3969 | 336337      <xs:element name="ValidSource" minOccurs="0">
3970 | 337338      <xs:complexType>
3971 | 338339      <xs:attribute name="typeRef" type="xs:QName"
3972 |         use="required"/>
3973 | 339340      </xs:complexType>
3974 | 340341      </xs:element>
3975 | 341342      <xs:element name="ValidTarget" minOccurs="0">
3976 | 342343      <xs:complexType>
3977 | 343344      <xs:attribute name="typeRef" type="xs:QName"
3978 |         use="required"/>
3979 | 344345      </xs:complexType>
3980 | 345346      </xs:element>
3981 | 346347      </xs:sequence>
3982 | 347348      </xs:extension>
3983 | 348349      </xs:complexContent>
3984 | 349350      </xs:complexType>
3985 | 350351
3986 | 351352      <xs:complexType name="tRelationshipTypeImplementation">
3987 | 352353      <xs:complexContent>
3988 | 353354      <xs:extension base="tExtensibleElements">
3989 | 354355      <xs:sequence>
3990 | 355356      <xs:element name="Tags" type="tTags" minOccurs="0"/>
3991 | 356357      <xs:element name="DerivedFrom" minOccurs="0">
3992 | 357358      <xs:complexType>
3993 | 358359      <xs:attribute name="relationshipTypeImplementationRef"
3994 |         type="xs:QName" use="required"/>
3995 | 360361      </xs:complexType>
3996 | 361362      </xs:element>
3997 | 362363      <xs:element name="RequiredContainerFeatures"
3998 |         type="tRequiredContainerFeatures" minOccurs="0"/>
3999 | 364365      <xs:element name="ImplementationArtifacts"
4000 |         type="tImplementationArtifacts" minOccurs="0"/>
4001 | 366367      </xs:sequence>
4002 | 367368      <xs:attribute name="name" type="xs:NCName" use="required"/>
4003 | 368369      <xs:attribute name="targetNamespace" type="xs:anyURI"
4004 |         use="optional"/>
4005 | 370371      <xs:attribute name="relationshipType" type="xs:QName"
4006 |         use="required"/>
4007 | 372373      <xs:attribute name="abstract" type="tBoolean" use="optional"
4008 |         default="no"/>
4009 | 374375      <xs:attribute name="final" type="tBoolean" use="optional"
4010 |         default="no"/>
4011 | 376377      </xs:extension>
4012 | 377378      </xs:complexContent>
4013 | 378379      </xs:complexType>
4014 | 379380
4015 | 380381      <xs:complexType name="tRelationshipTemplate">
4016 | 381382      <xs:complexContent>
4017 | 382383      <xs:extension base="tEntityTemplate">
4018 | 383384      <xs:sequence>
4019 | 384385      <xs:element name="SourceElement">
4020 | 385386      <xs:complexType>
4021 | 386387      <xs:attribute name="ref" type="xs:IDREF" use="required"/>
4022 | 387388      </xs:complexType>
4023 | 388389      </xs:element>
4024 | 389390      <xs:element name="TargetElement">
4025 | 390391      <xs:complexType>

```

```

4026 391392      <xs:attribute name="ref" type="xs:IDREF" use="required"/>
4027 392393      </xs:complexType>
4028 393394      </xs:element>
4029 394395      <xs:element name="RelationshipConstraints" minOccurs="0">
4030 395396      <xs:complexType>
4031 396397      <xs:sequence>
4032 397398      <xs:element name="RelationshipConstraint"
4033 398399      maxOccurs="unbounded">
4034 399400      <xs:complexType>
4035 400401      <xs:sequence>
4036 401402      <xs:any namespace="##other" processContents="lax"
4037 402403      minOccurs="0"/>
4038 403404      </xs:sequence>
4039 404405      <xs:attribute name="constraintType" type="xs:anyURI"
4040 405406      use="required"/>
4041 406407      </xs:complexType>
4042 407408      </xs:element>
4043 408409      </xs:sequence>
4044 409410      </xs:complexType>
4045 410411      </xs:element>
4046 411412      </xs:sequence>
4047 412413      <xs:attribute name="name" type="xs:string" use="optional"/>
4048 413414      </xs:extension>
4049 414415      </xs:complexContent>
4050 415416      </xs:complexType>
4051 416417
4052 417418 <xs:complexType name="tNodeType">
4053 418419 <xs:complexContent>
4054 419420 <xs:extension base="tEntityType">
4055 420421 <xs:sequence>
4056 421422 <xs:element name="RequirementDefinitions" minOccurs="0">
4057 422423 <xs:complexType>
4058 423424 <xs:sequence>
4059 424425 <xs:element name="RequirementDefinition"
4060 425426 type="tRequirementDefinition" maxOccurs="unbounded"/>
4061 426427 </xs:sequence>
4062 427428 </xs:complexType>
4063 428429 </xs:element>
4064 429430 <xs:element name="CapabilityDefinitions" minOccurs="0">
4065 430431 <xs:complexType>
4066 431432 <xs:sequence>
4067 432433 <xs:element name="CapabilityDefinition"
4068 433434 type="tCapabilityDefinition" maxOccurs="unbounded"/>
4069 434435 </xs:sequence>
4070 435436 </xs:complexType>
4071 436437 </xs:element>
4072 437438 <xs:element name="InstanceStates"
4073 438439 type="tTopologyElementInstanceStates" minOccurs="0"/>
4074 439440 <xs:element name="Interfaces" minOccurs="0">
4075 440441 <xs:complexType>
4076 441442 <xs:sequence>
4077 442443 <xs:element name="Interface" type="tInterface"
4078 443444 maxOccurs="unbounded"/>
4079 444445 </xs:sequence>
4080 445446 </xs:complexType>
4081 446447 </xs:element>
4082 447448 </xs:sequence>
4083 448449 </xs:extension>

```

```

4084 449450      </xs:complexContent>
4085 450451      </xs:complexType>
4086 451452
4087 452453      <xs:complexType name="tNodeTypeImplementation">
4088 453454      <xs:complexContent>
4089 454455      <xs:extension base="tExtensibleElements">
4090 455456      <xs:sequence>
4091 456457      <xs:element name="Tags" type="tTags" minOccurs="0"/>
4092 457458      <xs:element name="DerivedFrom" minOccurs="0">
4093 458459      <xs:complexType>
4094 459460      <xs:attribute name="nodeTypeImplementationRef"
4095      type="xs:QName"
4096 460461      use="required"/>
4097 461462      </xs:complexType>
4098 462463      </xs:element>
4099 463464      <xs:element name="RequiredContainerFeatures"
4100 464465      type="tRequiredContainerFeatures" minOccurs="0"/>
4101 465466      <xs:element name="ImplementationArtifacts"
4102 466467      type="tImplementationArtifacts" minOccurs="0"/>
4103 467468      <xs:element name="DeploymentArtifacts"
4104      type="tDeploymentArtifacts"
4105 468469      minOccurs="0"/>
4106 469470      </xs:sequence>
4107 470471      <xs:attribute name="name" type="xs:NCName" use="required"/>
4108 471472      <xs:attribute name="targetNamespace" type="xs:anyURI"
4109 472473      use="optional"/>
4110 473474      <xs:attribute name="nodeType" type="xs:QName"
4111      use="required"/>
4112 474475      <xs:attribute name="abstract" type="tBoolean" use="optional"
4113 475476      default="no"/>
4114 476477      <xs:attribute name="final" type="tBoolean" use="optional"
4115 477478      default="no"/>
4116 478479      </xs:extension>
4117 479480      </xs:complexContent>
4118 480481      </xs:complexType>
4119 481482
4120 482483      <xs:complexType name="tRequirementType">
4121 483484      <xs:complexContent>
4122 484485      <xs:extension base="tEntityType">
4123 485486      <xs:attribute name="requiredCapabilityType" type="xs:QName"
4124 486487      use="optional"/>
4125 487488      </xs:extension>
4126 488489      </xs:complexContent>
4127 489490      </xs:complexType>
4128 490491
4129 491492      <xs:complexType name="tRequirementDefinition">
4130 492493      <xs:complexContent>
4131 493494      <xs:extension base="tExtensibleElements">
4132 494495      <xs:sequence>
4133 495496      <xs:element name="Constraints" minOccurs="0">
4134 496497      <xs:complexType>
4135 497498      <xs:sequence>
4136 498499      <xs:element name="Constraint" type="tConstraint"
4137 499500      maxOccurs="unbounded"/>
4138 500501      </xs:sequence>
4139 501502      </xs:complexType>
4140 502503      </xs:element>
4141 503504      </xs:sequence>

```

```

4142 504505 <xs:attribute name="name" type="xs:string" use="required"/>
4143 505506 <xs:attribute name="requirementType" type="xs:QName"
4144 506507 use="required"/>
4145 507508 <xs:attribute name="lowerBound" type="xs:int" use="optional"
4146 508509 default="1"/>
4147 509510 <xs:attribute name="upperBound" use="optional" default="1">
4148 510511 <xs:simpleType>
4149 511512 <xs:union>
4150 512513 <xs:simpleType>
4151 513514 <xs:restriction base="xs:nonNegativeInteger">
4152 514515 <xs:pattern value="([1-9]+[0-9]*)"/>
4153 515516 </xs:restriction>
4154 516517 </xs:simpleType>
4155 517518 <xs:simpleType>
4156 518519 <xs:restriction base="xs:string">
4157 519520 <xs:enumeration value="unbounded"/>
4158 520521 </xs:restriction>
4159 521522 </xs:simpleType>
4160 522523 </xs:union>
4161 523524 </xs:simpleType>
4162 524525 </xs:attribute>
4163 525526 </xs:extension>
4164 526527 </xs:complexContent>
4165 527528 </xs:complexType>
4166 528529
4167 529530 <xs:complexType name="tRequirement">
4168 530531 <xs:complexContent>
4169 531532 <xs:extension base="tEntityType">
4170 532533 <xs:attribute name="name" type="xs:string" use="required"/>
4171 533534 </xs:extension>
4172 534535 </xs:complexContent>
4173 535536 </xs:complexType>
4174 536537
4175 537538 <xs:complexType name="tCapabilityType">
4176 538539 <xs:complexContent>
4177 539540 <xs:extension base="tEntityType"/>
4178 540541 </xs:complexContent>
4179 541542 </xs:complexType>
4180 542543
4181 543544 <xs:complexType name="tCapabilityDefinition">
4182 544545 <xs:complexContent>
4183 545546 <xs:extension base="tExtensibleElements">
4184 546547 <xs:sequence>
4185 547548 <xs:element name="Constraints" minOccurs="0">
4186 548549 <xs:complexType>
4187 549550 <xs:sequence>
4188 550551 <xs:element name="Constraint" type="tConstraint"
4189 551552 maxOccurs="unbounded"/>
4190 552553 </xs:sequence>
4191 553554 </xs:complexType>
4192 554555 </xs:element>
4193 555556 </xs:sequence>
4194 556557 <xs:attribute name="name" type="xs:string" use="required"/>
4195 557558 <xs:attribute name="capabilityType" type="xs:QName"
4196 558559 use="required"/>
4197 559560 <xs:attribute name="lowerBound" type="xs:int" use="optional"
4198 560561 default="1"/>
4199 561562 <xs:attribute name="upperBound" use="optional" default="1">

```

```

4200 562563 <xs:simpleType>
4201 563564 <xs:union>
4202 564565 <xs:simpleType>
4203 565566 <xs:restriction base="xs:nonNegativeInteger">
4204 566567 <xs:pattern value="([1-9]+[0-9]*)"/>
4205 567568 </xs:restriction>
4206 568569 </xs:simpleType>
4207 569570 <xs:simpleType>
4208 570571 <xs:restriction base="xs:string">
4209 571572 <xs:enumeration value="unbounded"/>
4210 572573 </xs:restriction>
4211 573574 </xs:simpleType>
4212 574575 </xs:union>
4213 575576 </xs:simpleType>
4214 576577 </xs:attribute>
4215 577578 </xs:extension>
4216 578579 </xs:complexContent>
4217 579580 </xs:complexType>
4218 580581
4219 581582 <xs:complexType name="tCapability">
4220 582583 <xs:complexContent>
4221 583584 <xs:extension base="tEntityType">
4222 584585 <xs:attribute name="name" type="xs:string" use="required"/>
4223 585586 </xs:extension>
4224 586587 </xs:complexContent>
4225 587588 </xs:complexType>
4226 588589
4227 589590 <xs:complexType name="tArtifactType">
4228 590591 <xs:complexContent>
4229 591592 <xs:extension base="tEntityType"/>
4230 592593 </xs:complexContent>
4231 593594 </xs:complexType>
4232 594595
4233 595596 <xs:complexType name="tArtifactTemplate">
4234 596597 <xs:complexContent>
4235 597598 <xs:extension base="tEntityType">
4236 598599 <xs:sequence>
4237 599600 <xs:element name="ArtifactReferences" minOccurs="0">
4238 600601 <xs:complexType>
4239 601602 <xs:sequence>
4240 602603 <xs:element name="ArtifactReference"
4241 type="tArtifactReference"
4242 maxOccurs="unbounded"/>
4243 604605 </xs:sequence>
4244 605606 </xs:complexType>
4245 606607 </xs:element>
4246 607608 </xs:sequence>
4247 608609 <xs:attribute name="name" type="xs:string" use="optional"/>
4248 609610 </xs:extension>
4249 610611 </xs:complexContent>
4250 611612 </xs:complexType>
4251 612613
4252 613614 <xs:complexType name="tDeploymentArtifacts">
4253 614615 <xs:sequence>
4254 615616 <xs:element name="DeploymentArtifact"
4255 type="tDeploymentArtifact"
4256 maxOccurs="unbounded"/>
4257 617618 </xs:sequence>

```



```

4258 | 618619      </xs:complexType>
4259 | 619620
4260 | 620621      <xs:complexType name="tDeploymentArtifact">
4261 | 621622      <xs:complexContent>
4262 | 622623      <xs:extension base="tExtensibleElements">
4263 | 623624      <xs:attribute name="name" type="xs:string" use="required"/>
4264 | 624625      <xs:attribute name="artifactType" type="xs:QName"
4265 |         use="required"/>
4266 | 625626      <xs:attribute name="artifactRef" type="xs:QName"
4267 |         use="optional"/>
4268 | 626627      </xs:extension>
4269 | 627628      </xs:complexContent>
4270 | 628629      </xs:complexType>
4271 | 629630
4272 | 630631      <xs:complexType name="tImplementationArtifacts">
4273 | 631632      <xs:sequence>
4274 | 632633      <xs:element name="ImplementationArtifact"
4275 |         maxOccurs="unbounded">
4276 | 633634      <xs:complexType>
4277 | 634635      <xs:complexContent>
4278 | 635636      <xs:extension base="tImplementationArtifact"/>
4279 | 636637      </xs:complexContent>
4280 | 637638      </xs:complexType>
4281 | 638639      </xs:element>
4282 | 639640      </xs:sequence>
4283 | 640641      </xs:complexType>
4284 | 641642
4285 | 642643      <xs:complexType name="tImplementationArtifact">
4286 | 643644      <xs:complexContent>
4287 | 644645      <xs:extension base="tExtensibleElements">
4288 | 645646      <xs:attribute name="interfaceName" type="xs:anyURI"
4289 |         use="optional"/>
4290 | 647648      <xs:attribute name="operationName" type="xs:NCName"
4291 |         use="optional"/>
4292 | 649650      <xs:attribute name="artifactType" type="xs:QName"
4293 |         use="required"/>
4294 | 650651      <xs:attribute name="artifactRef" type="xs:QName"
4295 |         use="optional"/>
4296 | 651652      </xs:extension>
4297 | 652653      </xs:complexContent>
4298 | 653654      </xs:complexType>
4299 | 654655
4300 | 655656      <xs:complexType name="tPlans">
4301 | 656657      <xs:sequence>
4302 | 657658      <xs:element name="Plan" type="tPlan" maxOccurs="unbounded"/>
4303 | 658659      </xs:sequence>
4304 | 659660      <xs:attribute name="targetNamespace" type="xs:anyURI"/>
4305 | 661        use="optional"/>
4306 | 660662      </xs:complexType>
4307 | 661663
4308 | 662664      <xs:complexType name="tPlan">
4309 | 663665      <xs:complexContent>
4310 | 664666      <xs:extension base="tExtensibleElements">
4311 | 665667      <xs:sequence>
4312 | 666668      <xs:element name="Precondition" type="tCondition"
4313 |         minOccurs="0"/>
4314 | 667669      <xs:element name="InputParameters" minOccurs="0">
4315 | 668670      <xs:complexType>

```

```

4316 | 669671      <xs:sequence>
4317 | 670672      <xs:element name="InputParameter" type="tParameter"
4318 | 671673      maxOccurs="unbounded"/>
4319 | 672674      </xs:sequence>
4320 | 673675      </xs:complexType>
4321 | 674676      </xs:element>
4322 | 675677      <xs:element name="OutputParameters" minOccurs="0">
4323 | 676678      <xs:complexType>
4324 | 677679      <xs:sequence>
4325 | 678680      <xs:element name="OutputParameter" type="tParameter"
4326 | 679681      maxOccurs="unbounded"/>
4327 | 680682      </xs:sequence>
4328 | 681683      </xs:complexType>
4329 | 682684      </xs:element>
4330 | 683685      <xs:choice>
4331 | 684686      <xs:element name="PlanModel">
4332 | 685687      <xs:complexType>
4333 | 686688      <xs:sequence>
4334 | 687689      <xs:any namespace="##other" processContents="lax"/>
4335 | 688690      </xs:sequence>
4336 | 689691      </xs:complexType>
4337 | 690692      </xs:element>
4338 | 691693      <xs:element name="PlanModelReference">
4339 | 692694      <xs:complexType>
4340 | 693695      <xs:attribute name="reference" type="xs:anyURI"
4341 | 694696      use="required"/>
4342 | 695697      </xs:complexType>
4343 | 696698      </xs:element>
4344 | 697699      </xs:choice>
4345 | 698700      </xs:sequence>
4346 | 699701      <xs:attribute name="id" type="xs:ID" use="required"/>
4347 | 700702      <xs:attribute name="name" type="xs:string" use="optional"/>
4348 | 701703      <xs:attribute name="planType" type="xs:anyURI"
4349 |          use="required"/>
4350 | 702704      <xs:attribute name="planLanguage" type="xs:anyURI"
4351 |          use="required"/>
4352 | 703705      </xs:extension>
4353 | 704706      </xs:complexContent>
4354 | 705707      </xs:complexType>
4355 | 706708
4356 | 707709      <xs:complexType name="tPolicyType">
4357 | 708710      <xs:complexContent>
4358 | 709711      <xs:extension base="tEntityType">
4359 | 710712      <xs:sequence>
4360 | 711713      <xs:element name="AppliesTo" type="tAppliesTo"
4361 |          minOccurs="0"/>
4362 | 712714      </xs:sequence>
4363 | 713715      <xs:attribute name="policyLanguage" type="xs:anyURI"
4364 | 714716      use="optional"/>
4365 | 715717      </xs:extension>
4366 | 716718      </xs:complexContent>
4367 | 717719      </xs:complexType>
4368 | 718720
4369 | 719721      <xs:complexType name="tPolicyTemplate">
4370 | 720722      <xs:complexContent>
4371 | 721723      <xs:extension base="tEntityTemplate">
4372 | 722724      <xs:attribute name="name" type="xs:string" use="optional"/>
4373 | 723725      </xs:extension>

```

```

4374 | 724726      </xs:complexContent>
4375 | 725727      </xs:complexType>
4376 | 726728
4377 | 727729      <xs:complexType name="tAppliesTo">
4378 | 728730      <xs:sequence>
4379 | 729731      <xs:element name="NodeTypeReference" maxOccurs="unbounded">
4380 | 730732      <xs:complexType>
4381 | 731733      <xs:attribute name="typeRef" type="xs:QName"
4382 |         use="required"/>
4383 | 732734      </xs:complexType>
4384 | 733735      </xs:element>
4385 | 734736      </xs:sequence>
4386 | 735737      </xs:complexType>
4387 | 736738
4388 | 737739      <xs:complexType name="tPolicy">
4389 | 738740      <xs:complexContent>
4390 | 739741      <xs:extension base="tExtensibleElements">
4391 | 740742      <xs:attribute name="name" type="xs:string" use="optional"/>
4392 | 741743      <xs:attribute name="policyType" type="xs:QName"
4393 |         use="required"/>
4394 | 742744      <xs:attribute name="policyRef" type="xs:QName"
4395 |         use="optional"/>
4396 | 743745      </xs:extension>
4397 | 744746      </xs:complexContent>
4398 | 745747      </xs:complexType>
4399 | 746748
4400 | 747749      <xs:complexType name="tConstraint">
4401 | 748750      <xs:sequence>
4402 | 749751      <xs:any namespace="##other" processContents="lax"/>
4403 | 750752      </xs:sequence>
4404 | 751753      <xs:attribute name="constraintType" type="xs:anyURI"
4405 |         use="required"/>
4406 | 752754      </xs:complexType>
4407 | 753755
4408 | 754756      <xs:complexType name="tPropertyConstraint">
4409 | 755757      <xs:complexContent>
4410 | 756758      <xs:extension base="tConstraint">
4411 | 757759      <xs:attribute name="property" type="xs:string"
4412 |         use="required"/>
4413 | 758760      </xs:extension>
4414 | 759761      </xs:complexContent>
4415 | 760762      </xs:complexType>
4416 | 761763
4417 | 762764      <xs:complexType name="tExtensions">
4418 | 763765      <xs:complexContent>
4419 | 764766      <xs:extension base="tExtensibleElements">
4420 | 765767      <xs:sequence>
4421 | 766768      <xs:element name="Extension" type="tExtension"
4422 |             maxOccurs="unbounded"/>
4423 | 768770      </xs:sequence>
4424 | 769771      </xs:extension>
4425 | 770772      </xs:complexContent>
4426 | 771773      </xs:complexType>
4427 | 772774
4428 | 773775      <xs:complexType name="tExtension">
4429 | 774776      <xs:complexContent>
4430 | 775777      <xs:extension base="tExtensibleElements">

```

```

4431 | 776778      <xs:attribute name="namespace" type="xs:anyURI"
4432 |           use="required"/>
4433 | 777779      <xs:attribute name="mustUnderstand" type="tBoolean"
4434 |           use="optional"
4435 | 778780      default="yes"/>
4436 | 779781      </xs:extension>
4437 | 780782      </xs:complexContent>
4438 | 781783      </xs:complexType>
4439 | 782784
4440 | 783785      <xs:complexType name="tParameter">
4441 | 784786      <xs:attribute name="name" type="xs:string" use="required"/>
4442 | 785787      <xs:attribute name="type" type="xs:string" use="required"/>
4443 | 786788      <xs:attribute name="required" type="tBoolean" use="optional"
4444 | 787789      default="yes"/>
4445 | 788790      </xs:complexType>
4446 | 789791
4447 | 790792      <xs:complexType name="tInterface">
4448 | 791793      <xs:sequence>
4449 | 792794      <xs:element name="Operation" type="tOperation"
4450 | 793795      maxOccurs="unbounded"/>
4451 | 794796      </xs:sequence>
4452 | 795797      <xs:attribute name="name" type="xs:anyURI" use="required"/>
4453 | 796798      </xs:complexType>
4454 | 797799
4455 | 798800      <xs:complexType name="tExportedInterface">
4456 | 799801      <xs:sequence>
4457 | 800802      <xs:element name="Operation" type="tExportedOperation"
4458 | 801803      maxOccurs="unbounded"/>
4459 | 802804      </xs:sequence>
4460 | 803805      <xs:attribute name="name" type="xs:anyURI" use="required"/>
4461 | 804806      </xs:complexType>
4462 | 805807
4463 | 806808      <xs:complexType name="tOperation">
4464 | 807809      <xs:complexContent>
4465 | 808810      <xs:extension base="tExtensibleElements">
4466 | 809811      <xs:sequence>
4467 | 810812      <xs:element name="InputParameters" minOccurs="0">
4468 | 811813      <xs:complexType>
4469 | 812814      <xs:sequence>
4470 | 813815      <xs:element name="InputParameter" type="tParameter"
4471 | 814816      maxOccurs="unbounded"/>
4472 | 815817      </xs:sequence>
4473 | 816818      </xs:complexType>
4474 | 817819      </xs:element>
4475 | 818820      <xs:element name="OutputParameters" minOccurs="0">
4476 | 819821      <xs:complexType>
4477 | 820822      <xs:sequence>
4478 | 821823      <xs:element name="OutputParameter" type="tParameter"
4479 | 822824      maxOccurs="unbounded"/>
4480 | 823825      </xs:sequence>
4481 | 824826      </xs:complexType>
4482 | 825827      </xs:element>
4483 | 826828      </xs:sequence>
4484 | 827829      <xs:attribute name="name" type="xs:NCName" use="required"/>
4485 | 828830      </xs:extension>
4486 | 829831      </xs:complexContent>
4487 | 830832      </xs:complexType>
4488 | 831833

```

```

4489 | 832834 <xs:complexType name="tExportedOperation">
4490 | 833835 <xs:choice>
4491 | 834836 <xs:element name="NodeOperation">
4492 | 835837 <xs:complexType>
4493 | 836838 <xs:attribute name="nodeRef" type="xs:IDREF"
4494 | use="required"/>
4495 | 837839 <xs:attribute name="interfaceName" type="xs:anyURI"
4496 | 838840 use="required"/>
4497 | 839841 <xs:attribute name="operationName" type="xs:NCName"
4498 | 840842 use="required"/>
4499 | 841843 </xs:complexType>
4500 | 842844 </xs:element>
4501 | 843845 <xs:element name="RelationshipOperation">
4502 | 844846 <xs:complexType>
4503 | 845847 <xs:attribute name="relationshipRef" type="xs:IDREF"
4504 | 846848 use="required"/>
4505 | 847849 <xs:attribute name="interfaceName" type="xs:anyURI"
4506 | 848850 use="required"/>
4507 | 849851 <xs:attribute name="operationName" type="xs:NCName"
4508 | 850852 use="required"/>
4509 | 851853 </xs:complexType>
4510 | 852854 </xs:element>
4511 | 853855 <xs:element name="Plan">
4512 | 854856 <xs:complexType>
4513 | 855857 <xs:attribute name="planRef" type="xs:IDREF"
4514 | use="required"/>
4515 | 856858 </xs:complexType>
4516 | 857859 </xs:element>
4517 | 858860 </xs:choice>
4518 | 859861 <xs:attribute name="name" type="xs:NCName" use="required"/>
4519 | 860862 </xs:complexType>
4520 | 861863 </xs:element>
4521 | 862864 <xs:complexType name="tCondition">
4522 | 863865 <xs:sequence>
4523 | 864866 <xs:any processContents="lax" minOccurs="0"
4524 | maxOccurs="unbounded"/>
4525 | 865867 </xs:sequence>
4526 | 866868 <xs:attribute name="expressionLanguage" type="xs:anyURI"
4527 | 867869 use="required"/>
4528 | 868870 </xs:complexType>
4529 | 869871 </xs:element>
4530 | 870872 <xs:complexType name="tTopologyElementInstanceStates">
4531 | 871873 <xs:sequence>
4532 | 872874 <xs:element name="InstanceState" maxOccurs="unbounded">
4533 | 873875 <xs:complexType>
4534 | 874876 <xs:attribute name="state" type="xs:anyURI" use="required"/>
4535 | 875877 </xs:complexType>
4536 | 876878 </xs:element>
4537 | 877879 </xs:sequence>
4538 | 878880 </xs:complexType>
4539 | 879881 </xs:element>
4540 | 880882 <xs:complexType name="tArtifactReference">
4541 | 881883 <xs:choice minOccurs="0" maxOccurs="unbounded">
4542 | 882884 <xs:element name="Include">
4543 | 883885 <xs:complexType>
4544 | 884886 <xs:attribute name="pattern" type="xs:string"
4545 | use="required"/>
4546 | 885887 </xs:complexType>

```

```

4547 | 886888      </xs:element>
4548 | 887889      <xs:element name="Exclude">
4549 | 888890      <xs:complexType>
4550 | 889891      <xs:attribute name="pattern" type="xs:string"
4551 |           use="required"/>
4552 | 890892      </xs:complexType>
4553 | 891893      </xs:element>
4554 | 892894      </xs:choice>
4555 | 893895      <xs:attribute name="reference" type="xs:anyURI"
4556 |           use="required"/>
4557 | 894896      </xs:complexType>
4558 | 895897
4559 | 896898      <xs:complexType name="tRequiredContainerFeatures">
4560 | 897899      <xs:sequence>
4561 | 898900      <xs:element name="RequiredContainerFeature"
4562 |           type="tRequiredContainerFeature" maxOccurs="unbounded"/>
4563 | 900902      </xs:sequence>
4564 | 901903      </xs:complexType>
4565 | 902904
4566 | 903905      <xs:complexType name="tRequiredContainerFeature">
4567 | 904906      <xs:attribute name="feature" type="xs:anyURI" use="required"/>
4568 | 905907      </xs:complexType>
4569 | 906908
4570 | 907909      <xs:simpleType name="tBoolean">
4571 | 908910      <xs:restriction base="xs:string">
4572 | 909911      <xs:enumeration value="yes"/>
4573 | 910912      <xs:enumeration value="no"/>
4574 | 911913      </xs:restriction>
4575 | 912914      </xs:simpleType>
4576 | 913915
4577 | 914916      <xs:simpleType name="importedURI">
4578 | 915917      <xs:restriction base="xs:anyURI"/>
4579 | 916918      </xs:simpleType>
4580 | 917919
4581 | 918920      </xs:schema>

```

4582

Appendix E. Sample

4583

This appendix contains the full sample used in this specification.

4584

E.1 Sample Service Topology Definition

4585

```
01 <Definitions name="MyServiceTemplateDefinition"  
02     targetNamespace="http://www.example.com/sample">
```

4586

4587

```
42 <Tags>
```

4588

```
43 <Tag name="author" value="someone@example.com"/>
```

4589

```
44 </Tags>
```

4590

```
45
```

4591

```
03 <Types>
```

4592

```
04 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

4593

```
05     elementFormDefault="qualified"
```

4594

```
06     attributeFormDefault="unqualified">
```

4595

```
07 <xs:element name="ApplicationProperties">
```

4596

```
08 <xs:complexType>
```

4597

```
09 <xs:sequence>
```

4598

```
10 <xs:element name="Owner" type="xs:string"/>
```

4599

```
11 <xs:element name="InstanceName" type="xs:string"/>
```

4600

```
12 <xs:element name="AccountID" type="xs:string"/>
```

4601

```
13 </xs:sequence>
```

4602

```
14 </xs:complexType>
```

4603

```
15 </xs:element>
```

4604

```
16 <xs:element name="AppServerProperties">
```

4605

```
17 <xs:complexType>
```

4606

```
18 <xs:sequence>
```

4607

```
19 <element name="HostName" type="xs:string"/>
```

4608

```
20 <element name="IPAddress" type="xs:string"/>
```

4609

```
21 <element name="HeapSize" type="xs:positiveInteger"/>
```

4610

```
22 <element name="SoapPort" type="xs:positiveInteger"/>
```

4611

```
23 </xs:sequence>
```

4612

```
24 </xs:complexType>
```

4613

```
25 </xs:element>
```

4614

```
26 </xs:schema>
```

4615

```
27 </Types>
```

4616

4617

```
29 <ServiceTemplate id="MyServiceTemplate">
```

4618

```
30
```

4619

```
31 <Tags>
```

4620

```
32 <Tag name="author" value="someone@example.com"/>
```

4621

```
33 </Tags>
```

4622

```
34
```

4623

```
35 <TopologyTemplate id="SampleApplication">
```

4624

```
36
```

4625

```
37 <NodeTemplate id="MyApplication"
```

4626

```
38     name="My Application"
```

4627

```
39     nodeType="abc:Application">
```

4628

```
40 <Properties>
```

4629

```
41 <ApplicationProperties>
```

4630

```
42 <Owner>Frank</Owner>
```

4631

```
43 <InstanceName>Thomas' favorite application</InstanceName>
```

4632

```
44 </ApplicationProperties>
```

4633

```
45 </Properties>
```

```

4634 4146 <</NodeTemplate/>>
4635 4247
4636 4348 <NodeTemplate id="MyAppServer"
4637 4449     name="My Application Server"
4638 4550     nodeType="abc:ApplicationServer"
4639 4651     minInstances="0"
4640 4752     maxInstances="unbounded"/>
4641 4853
4642 4954 <RelationshipTemplate id="MyDeploymentRelationship"
4643 5055     relationshipType="abc:deployedOn">
4644 5156     <SourceElement id="MyApplication"/>
4645 5257     <TargetElement id="MyAppServer"/>
4646 5358 </RelationshipTemplate>
4647 5459
4648 5560 </TopologyTemplate>
4649 5661
4650 5762 <Plans>
4651 5863 <Plan id="DeployApplication"
4652 5964     name="Sample Application Build Plan"
4653 6065     planType="http://docs.oasis-
4654 6166     open.org/tosca/ns/2011/12/PlanTypes/BuildPlan"
4655 6267     planLanguage="http://www.omg.org/spec/BPMN/20100524/MODEL">
4656 6368
4657 6469     <PreEcondition expressionLanguage="www.example.com/text"> ?
4658 6570     Run only if funding is available
4659 6671 </PreEcondition>
4660 6772
4661 6873 <PlanModel>
4662 6974 <process name="DeployNewApplication" id="p1">
4663 7075 <documentation>This process deploys a new instance of the
4664 7176 sample application.
4665 7277 </documentation>
4666 7378
4667 7479 <task id="t1" name="CreateAccount"/>
4668 7580
4669 7681 <task id="t2" name="AcquireNetworkAddresses"
4670 7782     isSequential="false"
4671 7883     loopDataInput="t2Input.LoopCounter"/>
4672 7984 <documentation>Assumption: t2 gets data of type "input"
4673 8085     as input and this data has a field names "LoopCounter"
4674 8186     that contains the actual multiplicity of the task.
4675 8287 </documentation>
4676 8388
4677 8489 <task id="t3" name="DeployApplicationServer"
4678 8590     isSequential="false"
4679 8691     loopDataInput="t3Input.LoopCounter"/>
4680 8792
4681 8893 <task id="t4" name="DeployApplication"
4682 8994     isSequential="false"
4683 9095     loopDataInput="t4Input.LoopCounter"/>
4684 9196
4685 9297 <sequenceFlow id="s1" targetRef="t2" sourceRef="t1"/>
4686 9398 <sequenceFlow id="s2" targetRef="t3" sourceRef="t2"/>
4687 9499 <sequenceFlow id="s3" targetRef="t4" sourceRef="t3"/>
4688 95100 </process>
4689 96101 </PlanModel>
4690 97102 </Plan>
4691 98103

```



```

4692 99104      <Plan id="RemoveApplication"
4693 100105          planType="http://docs.oasis-
4694 101106          open.org/tosca/ns/2011/12/PlanTypes/TerminationPlan"
4695 102107          planLanguage="http://docs.oasis-
4696 103108          open.org/wsbpel/2.0/process/executable">
4697 104109          <PlanModelReference reference="prj:RemoveApp"/>
4698 105110          </Plan>
4699 106111      </Plans>
4700 107112
4701 108113  </ServiceTemplate>
4702 109114
4703 110115  <NodeType name="Application">
4704 111116      <documentation xml:lang="EN">
4705 112117          A reusable definition of a node type representing an
4706 113118          application that can be deployed on application servers.
4707 114119      </documentation>
4708 115120      <NodeTypeProperties element="ApplicationProperties"/>
4709 116121      <InstanceStates>
4710 117122          <InstanceState state="http://www.example.com/started"/>
4711 118123          <InstanceState state="http://www.example.com/stopped"/>
4712 119124      </InstanceStates>
4713 120125      <Interfaces>
4714 121126          <Interface name="DeploymentInterface">
4715 122127              <Operation name="DeployApplication">
4716 123128                  <InputParameters>
4717 124129                      <InputParamter name="InstanceName"
4718 125130                          type="xs:string"/>
4719 126131                      <InputParamter name="AppServerHostname"
4720 127132                          type="xs:string"/>
4721 128133                      <InputParamter name="ContextRoot"
4722 129134                          type="xs:string"/>
4723 130135                  </InputParameters>
4724 131136              </Operation>
4725 132137          </Interface>
4726 133138      </Interfaces>
4727 134139  </NodeType>
4728 135140
4729 136141  <NodeType name="ApplicationServer"
4730 137142          targetNamespace="http://www.example.com/sample">
4731 138143      <NodeTypeProperties element="AppServerProperties"/>
4732 139144      <Interfaces>
4733 140145          <Interface name="MyAppServerInterface">
4734 141146              <Operation name="AcquireNetworkAddress"/>
4735 142147              <Operation name="DeployApplicationServer"/>
4736 143148          </Interface>
4737 144149      </Interfaces>
4738 145150  </NodeType>
4739 146151
4740 147152  <RelationshipType name="deployedOn">
4741 148153      <documentation xml:lang="EN">
4742 149154          A reusable definition of relation that expresses deployment
4743 150155          of
4744 151156          an artifact on a hosting environment.
4745 152157      </documentation>
4746 153158  </RelationshipType>
4747 154159
4748 154159  </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 Entry-Definitions property for TOSCA.meta 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>

4751