



Topology and Orchestration Specification for Cloud Applications Version 1.0

Committee Specification ~~0102~~

~~18 March~~ 09 May 2013

Specification URIs

This version:

<http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs02/TOSCA-v1.0-cs02.pdf> (Authoritative)
<http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs02/TOSCA-v1.0-cs02.html>
<http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs02/TOSCA-v1.0-cs02.doc>

Previous version:

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

Previous version:

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

Latest version:

<http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf> (Authoritative)
<http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html>
<http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.doc>
<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/>
- XML schemas: <http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs02/schemas/>

Declared XML namespace:

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

Abstract:

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

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

Status:

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

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

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

Citation format:

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

[TOSCA-v1.0]

Topology and Orchestration Specification for Cloud Applications Version 1.0. 18 March 09 May 2013. OASIS Committee Specification 042. <http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html> <http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs02/TOSCA-v1.0-cs02.html>.

Notices

Copyright © OASIS Open 2013. All Rights Reserved.

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

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

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

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

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

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

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

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

Table of Contents

1	Introduction	9
2	Language Design	10
2.1	Dependencies on Other Specifications	10
2.2	Notational Conventions	10
2.3	Normative References	10
2.4	Non-Normative References	10
2.5	Typographical Conventions	11
2.6	Namespaces	11
2.7	Language Extensibility	12
3	Core Concepts and Usage Pattern	13
3.1	Core Concepts	13
3.2	Use Cases	14
3.2.1	Services as Marketable Entities	14
3.2.2	Portability of Service Templates	15
3.2.3	Service Composition	15
3.2.4	Relation to Virtual Images	15
3.3	Service Templates and Artifacts	15
3.4	Requirements and Capabilities	16
3.5	Composition of Service Templates	17
3.6	Policies in TOSCA	17
3.7	Archive Format for Cloud Applications	18
4	The TOSCA Definitions Document	20
4.1	XML Syntax	20
4.2	Properties	21
4.3	Example	24
5	Service Templates	25
5.1	XML Syntax	25
5.2	Properties	28
5.3	Example	39
6	Node Types	41
6.1	XML Syntax	41
6.2	Properties	42
6.3	Derivation Rules	45
6.4	Example	45
7	Node Type Implementations	47
7.1	XML Syntax	47
7.2	Properties	48
7.3	Derivation Rules	50
7.4	Example	51
8	Relationship Types	52
8.1	XML Syntax	52
8.2	Properties	53
8.3	Derivation Rules	54

8.4 Example.....	55
9— Relationship Type Implementations.....	56
9.1 XML Syntax.....	56
9.2 Properties.....	56
9.3 Derivation Rules.....	58
9.4 Example.....	59
10— Requirement Types.....	60
10.1 XML Syntax.....	60
10.2 Properties.....	60
10.3 Derivation Rules.....	61
10.4 Example.....	62
11— Capability Types.....	63
11.1 XML Syntax.....	63
11.2 Properties.....	63
11.3 Derivation Rules.....	64
11.4 Example.....	64
12— Artifact Types.....	66
12.1 XML Syntax.....	66
12.2 Properties.....	66
12.3 Derivation Rules.....	67
12.4 Example.....	67
13— Artifact Templates.....	69
13.1 XML Syntax.....	69
13.2 Properties.....	69
13.3 Example.....	71
14— Policy Types.....	72
14.1 XML Syntax.....	72
14.2 Properties.....	72
14.3 Derivation Rules.....	73
14.4 Example.....	74
15— Policy Templates.....	75
15.1 XML Syntax.....	75
15.2 Properties.....	75
15.3 Example.....	76
16— Cloud Service Archive (CSAR).....	77
16.1 Overall Structure of a CSAR.....	77
16.2 TOSCA Meta File.....	77
16.3 Example.....	78
17— Security Considerations.....	82
18— Conformance.....	83
Appendix A.— Portability and Interoperability Considerations.....	84
Appendix B.— Acknowledgements.....	85
Appendix C.— Complete TOSCA Grammar.....	87
Appendix D.— TOSCA Schema.....	95
Appendix E.— Sample.....	111

E.1 Sample Service Topology Definition	111
Appendix F. Revision History	114
1 Introduction	9
2 Language Design	10
2.1 Dependencies on Other Specifications	10
2.2 Notational Conventions	10
2.3 Normative References	10
2.4 Non-Normative References	10
2.5 Typographical Conventions	11
2.6 Namespaces	11
2.7 Language Extensibility	12
3 Core Concepts and Usage Pattern	13
3.1 Core Concepts	13
3.2 Use Cases	14
3.2.1 Services as Marketable Entities	14
3.2.2 Portability of Service Templates	15
3.2.3 Service Composition	15
3.2.4 Relation to Virtual Images	15
3.3 Service Templates and Artifacts	15
3.4 Requirements and Capabilities	16
3.5 Composition of Service Templates	17
3.6 Policies in TOSCA	17
3.7 Archive Format for Cloud Applications	18
4 The TOSCA Definitions Document	20
4.1 XML Syntax	20
4.2 Properties	21
4.3 Example	24
5 Service Templates	25
5.1 XML Syntax	25
5.2 Properties	28
5.3 Example	39
6 Node Types	41
6.1 XML Syntax	41
6.2 Properties	42
6.3 Derivation Rules	45
6.4 Example	45
7 Node Type Implementations	47
7.1 XML Syntax	47
7.2 Properties	48
7.3 Derivation Rules	50
7.4 Example	51
8 Relationship Types	52
8.1 XML Syntax	52
8.2 Properties	53
8.3 Derivation Rules	54

8.4 Example	55
9 Relationship Type Implementations	56
9.1 XML Syntax.....	56
9.2 Properties.....	56
9.3 Derivation Rules	58
9.4 Example	59
10 Requirement Types	60
10.1 XML Syntax	60
10.2 Properties.....	60
10.3 Derivation Rules	61
10.4 Example	62
11 Capability Types	63
11.1 XML Syntax	63
11.2 Properties.....	63
11.3 Derivation Rules	64
11.4 Example	64
12 Artifact Types.....	66
12.1 XML Syntax	66
12.2 Properties.....	66
12.3 Derivation Rules	67
12.4 Example	67
13 Artifact Templates.....	69
13.1 XML Syntax	69
13.2 Properties.....	69
13.3 Example	71
14 Policy Types	72
14.1 XML Syntax	72
14.2 Properties.....	72
14.3 Derivation Rules	73
14.4 Example	74
15 Policy Templates	75
15.1 XML Syntax	75
15.2 Properties.....	75
15.3 Example	76
16 Cloud Service Archive (CSAR).....	77
16.1 Overall Structure of a CSAR.....	77
16.2 TOSCA Meta File.....	77
16.3 Example	78
17 Security Considerations	82
18 Conformance	83
Appendix A. Portability and Interoperability Considerations	84
Appendix B. Acknowledgements	85
Appendix C. Complete TOSCA Grammar	87
Appendix D. TOSCA Schema.....	95
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.

2 Language Design

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

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

2.1 Dependencies on Other Specifications

TOSCA utilizes the following specifications:

- XML Schema 1.0

2.2 Notational Conventions

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

This specification follows XML naming and design rules as described in [Error! Reference source not found., i.e. \[UNCEFACT XMLNDR\], i.e.](#) uses upper camel-case notation for XML element names and lower camel-case notation for XML attribute names.

2.3 Normative References

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

2.4 Non-Normative References

- | | |
|------------|--|
| [BPEL 2.0] | <i>Web Services Business Process Execution Language Version 2.0</i> . OASIS Standard. 11 April 2007. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html . |
| [BPMN 2.0] | OMG Business Process Model and Notation (BPMN) Version 2.0, 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[http://w](http://www.dmtf.org/standards/published_documents/DSP0243_1.1.0.pdf)
- 53 www.dmtf.org/standards/published_documents/DSP0243_1.1.0.pdf
- 54 **[XPath 1.0]** XML Path Language (XPath) Version 1.0, W3C Recommendation, November
55 1999, <http://www.w3.org/TR/1999/REC-xpath-19991116>
- 56 **[UNCEFACT XMLNDR]** UN/CEFACT XML Naming and Design Rules Technical Specification,
57 Version 3.0,
58 [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)
59 [df](http://www.unece.org/fileadmin/DAM/cefact/xml/UNCEFACT+XML+NDR+V3p0.pdf)
- 60

61 2.5 Typographical Conventions

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

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

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

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

76 2.6 Namespaces

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

82

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

83 Table 1: Prefixes and namespaces used in this specification

84

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

88 **2.7 Language Extensibility**

89 The TOSCA extensibility mechanism allows:

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

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

98 **3 Core Concepts and Usage Pattern**

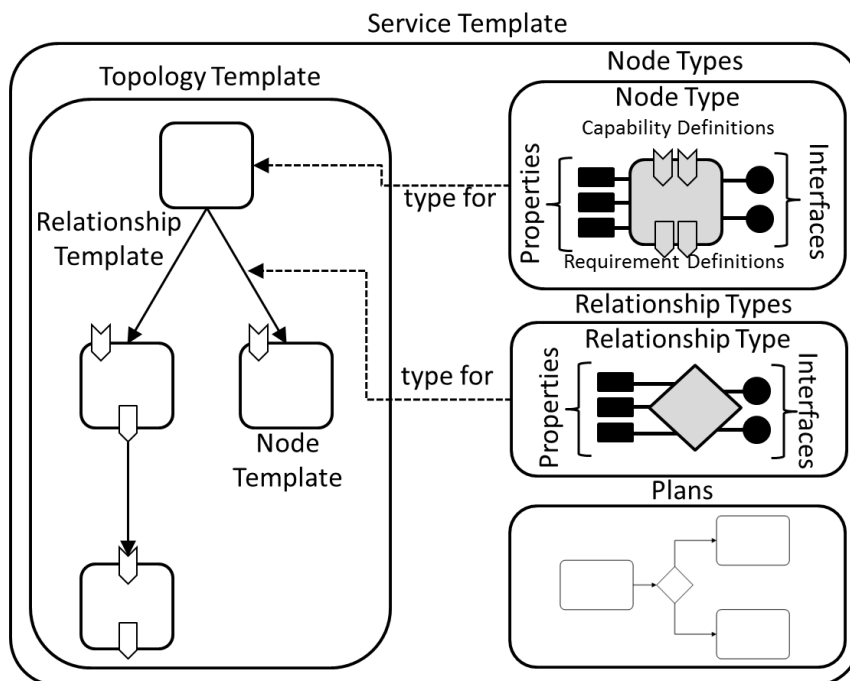
99 The main concepts behind TOSCA are described and some usage patterns of Service Templates are
100 sketched.

101 **3.1 Core Concepts**

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

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

115



116

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

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

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

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

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

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

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

163 3.2 Use Cases

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

165 3.2.1 Services as Marketable Entities

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

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

177 environment of a particular service provider. Other management plans useful in various states of the
178 whole lifecycle of a service could be specified as part of a Service Template. Similar to build plans such
179 management plans can be adapted to the concrete environment of a particular service provider.

180 Thus, not only the structure of a service can be defined in an interoperable manner, but also its
181 management plans. These Plans describe how instances of the specified service are created and
182 managed. Defining a set of management plans for a service will significantly reduce the cost of hosting a
183 service by providing reusable knowledge about best practices for managing each service. While the
184 modeler of a service can include deep domain knowledge into a plan, the user of such a service can use
185 a plan by simply “invoking” it. This hides the complexity of the underlying service behavior. This is very
186 similar to the situation resulting in the specification of ITIL.

187 **3.2.2 Portability of Service Templates**

188 Standardizing Service Templates supports the portability of definitions of IT Services. Here, portability
189 denotes the ability of one cloud provider to understand the structure and behavior of a Service Template
190 created by another party, e.g. another cloud provider, enterprise IT department, or service developer.

191 Note that portability of a service does not imply portability of its encompassed components. Portability of
192 a service means that its definition can be understood in an interoperable manner, i.e. the topology model
193 and corresponding plans are understood by standard compliant vendors. Portability of the individual
194 components themselves making up a particular service has to be ensured by other means – if it is
195 important for the service.

196 **3.2.3 Service Composition**

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

203 **3.2.4 Relation to Virtual Images**

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

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

216 **3.3 Service Templates and Artifacts**

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

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

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

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

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

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

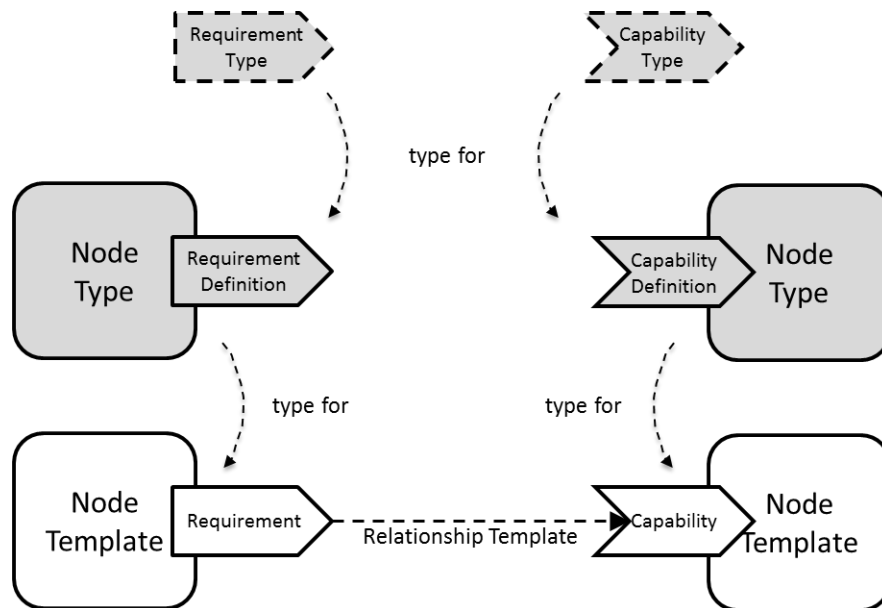
238 The instantiation of a node type can require providing deployment artifacts in the target managed
239 environment. For this purpose, a TOSCA container supports a set of types of deployment artifacts that it
240 can process. A service template that contains (implementation or deployment) artifacts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

242 3.4 Requirements and Capabilities

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

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

253



254

255

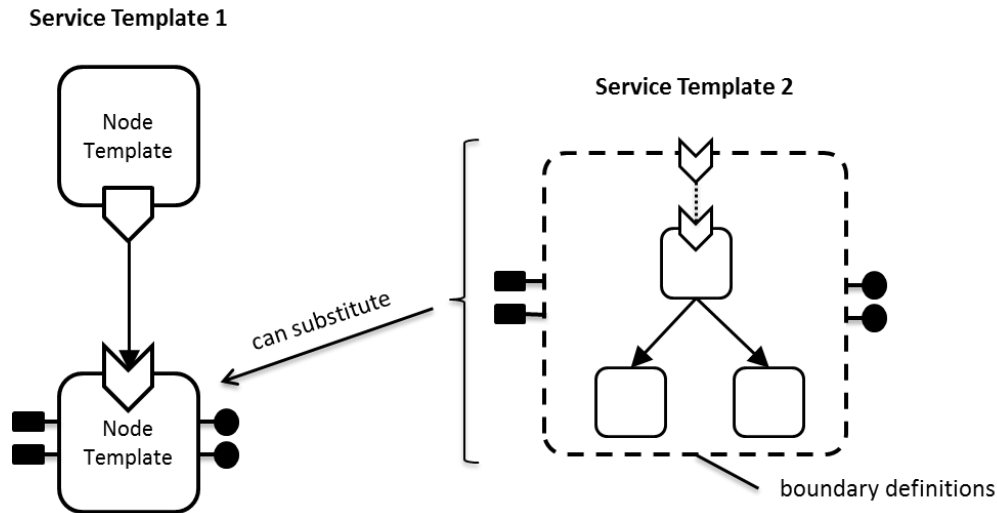
256

Figure 2: Requirements and Capabilities

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

269 **3.5 Composition of Service Templates**

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



277
 278 | Figure 3: Service Template Composition

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

289 **3.6 Policies in TOSCA**

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

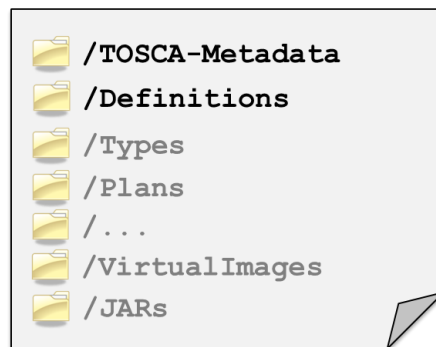
293 A Node Template can be associated with a set of Policies collectively expressing the non-functional
294 behavior or quality-of-services that each instance of the Node Template will expose. Each Policy specifies
295 the actual properties of the non-functional behavior, like the concrete payment information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

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

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

307 3.7 Archive Format for Cloud Applications

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

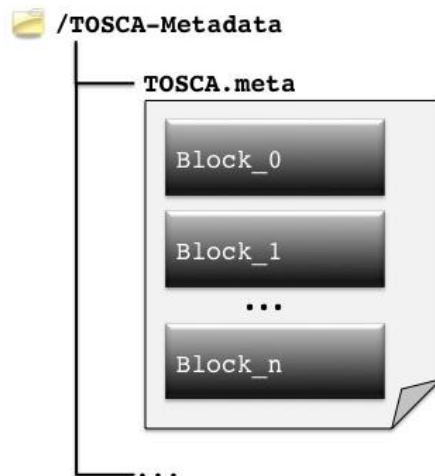


313

314 | Figure 4: Structure of the CSAR

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

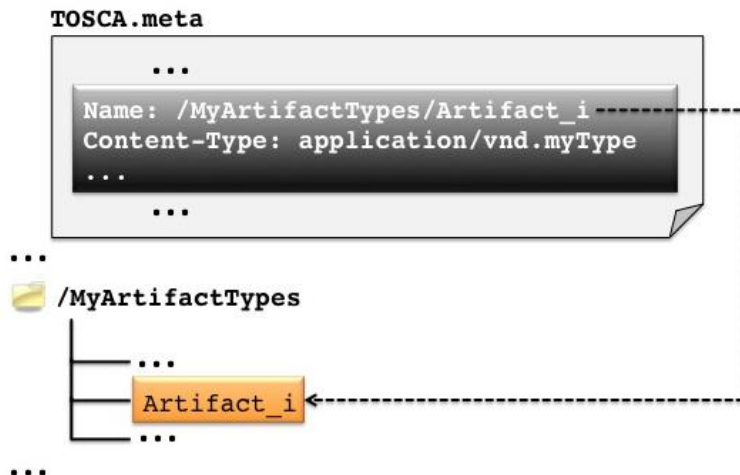
319 Each CSAR MUST contain a subdirectory called *TOSCA-Metadata*. This subdirectory MUST contain a
320 so-called *TOSCA meta file*. This file is named *TOSCA* and has the file extension *.meta*. It represents
321 metadata of the other files in the CSAR. This metadata is given in the format of name/value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



324
 325 |
 326
 327
 328
 329
 330

Figure 5: Structure of the TOSCA Meta File

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



331
 332 |
 333

Figure 6: Providing Metadata for Artifacts

334 4 The TOSCA Definitions Document

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

341 4.1 XML Syntax

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

```
343 01 <Definitions id="xs:ID"  
344 02     name="xs:string"?  
345 03     targetNamespace="xs:anyURI">  
346 04  
347 05     <Extensions>  
348 06         <Extension namespace="xs:anyURI"  
349 07             mustUnderstand="yes|no"?/> +  
350 08     </Extensions> ?  
351 09  
352 10     <Import namespace="xs:anyURI"?  
353 11         location="xs:anyURI"?  
354 12         importType="xs:anyURI"/> *  
355 13  
356 14     <Types>  
357 15         <xs:schema .../> *  
358 16     </Types> ?  
359 17  
360 18     (  
361 19         <ServiceTemplate> ... </ServiceTemplate>  
362 20     |  
363 21         <NodeType> ... </NodeType>  
364 22     |  
365 23         <NodeTypeImplementation> ... </NodeTypeImplementation>  
366 24     |  
367 25         <RelationshipType> ... </RelationshipType>  
368 26     |  
369 27         <RelationshipTypeImplementation> ... </RelationshipTypeImplementation>  
370 28     |  
371 29         <RequirementType> ... </RequirementType>  
372 30     |  
373 31         <CapabilityType> ... </CapabilityType>  
374 32     |  
375 33         <ArtifactType> ... </ArtifactType>  
376 34     |  
377 35         <ArtifactTemplate> ... </ArtifactTemplate>  
378 36     |  
379 37         <PolicyType> ... </PolicyType>  
380 38     |  
381 39         <PolicyTemplate> ... </PolicyTemplate>  
382 40     ) +  
383 41  
384 42 </Definitions>
```

385 4.2 Properties

386 The `Definitions` element has the following properties:

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

397 The `Extension` element has the following properties:

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

415 The `Import` element has the following properties:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
513 01 <documentation source="xs:anyURI"? xml:lang="xs:language"?>
514 02   ...
515 03 </documentation>
```

516 Example of use of a `documentation` element:

```
517 01 <Definitions id="MyDefinitions" name="My Definitions" ...>
518 02
519 03   <documentation xml:lang="EN">
520 04     This is a simple example of the usage of the documentation
521 05     element nested under a Definitions element. It could be used,
522 06     for example, to describe the purpose of the Definitions document
523 07     or to give an overview of elements contained within the Definitions
524 08     document.
525 09   </documentation>
526 10
527 11 </Definitions>
```

528 4.3 Example

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

```
533 01 <Definitions id="MyDefinitions" name="My Definitions"  
534 02   targetNamespace="http://www.example.com/MyDefinitions"  
535 03   xmlns:my="http://www.example.com/MyDefinitions">  
536 04  
537 05   <Import importType="http://www.w3.org/2001/XMLSchema"  
538 06     namespace="http://www.example.com/MyDefinitions">  
539 07  
540 08   <NodeType name="Application">  
541 09     <PropertiesDefinition element="my:ApplicationProperties"/>  
542 10   </NodeType>  
543 11  
544 12   <NodeType name="ApplicationServer">  
545 13     <PropertiesDefinition element="my:ApplicationServerProperties"/>  
546 14   </NodeType>  
547 15  
548 16   <RelationshipType name="ApplicationHostedOnApplicationServer">  
549 17     <ValidSource typeRef="my:Application"/>  
550 18     <ValidTarget typeRef="my:ApplicationServer"/>  
551 19   </RelationshipTemplate>  
552 20  
553 21 </Definitions>
```


554

5 Service Templates

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

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

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

563 5.1 XML Syntax

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

```
565 01 <ServiceTemplate id="xs:ID"  
566 02     name="xs:string"?  
567 03     targetNamespace="xs:anyURI"  
568 04     substitutableNodeType="xs:QName"?>  
569 05  
570 06 <Tags>  
571 07   <Tag name="xs:string" value="xs:string"/> +  
572 08 </Tags> ?  
573 09  
574 10 <BoundaryDefinitions>  
575 11   <Properties>  
576 12     XML fragment  
577 13     <PropertyMappings>  
578 14       <PropertyMapping serviceTemplatePropertyRef="xs:string"  
579 15         targetObjectRef="xs:IDREF"  
580 16         targetPropertyRef="xs:string"/> +  
581 17     </PropertyMappings/> ?  
582 18   </Properties> ?  
583 19  
584 20   <PropertyConstraints>  
585 21     <PropertyConstraint property="xs:string"  
586 22       constraintType="xs:anyURI"> +  
587 23     constraint ?  
588 24   </PropertyConstraint>  
589 25 </PropertyConstraints> ?  
590 26  
591 27 <Requirements>  
592 28   <Requirement name="xs:string"? ref="xs:IDREF"/> +  
593 29 </Requirements> ?  
594 30  
595 31 <Capabilities>  
596 32   <Capability name="xs:string"? ref="xs:IDREF"/> +  
597 33 </Capabilities> ?  
598 34  
599 35 <Policies>  
600 36   <Policy name="xs:string"? policyType="xs:QName"  
601 37     policyRef="xs:QName"?>  
602 38     policy specific content ?  
603 39   </Policy> +  
604 40 </Policies> ?
```

```

605 41
606 42     <Interfaces>
607 43         <Interface name="xs:NCName">
608 44             <Operation name="xs:NCName">
609 45                 (
610 46                     <NodeOperation nodeRef="xs:IDREF"
611 47                         interfaceName="xs:anyURI"
612 48                         operationName="xs:NCName"/>
613 49                 |
614 50                     <RelationshipOperation relationshipRef="xs:IDREF"
615 51                         interfaceName="xs:anyURI"
616 52                         operationName="xs:NCName"/>
617 53                 |
618 54                     <Plan planRef="xs:IDREF"/>
619 55                 )
620 56             </Operation> +
621 57         </Interface> +
622 58     </Interfaces> ?
623 59
624 60 </BoundaryDefinitions> ?
625 61
626 62 <TopologyTemplate>
627 63     (
628 64         <NodeTemplate id="xs:ID" name="xs:string"? type="xs:QName"
629 65             minInstances="xs:integer"?
630 66             maxInstances="xs:integer | xs:string"?>
631 67             <Properties>
632 68                 XML fragment
633 69             </Properties> ?
634 70
635 71             <PropertyConstraints>
636 72                 <PropertyConstraint property="xs:string"
637 73                     constraintType="xs:anyURI">
638 74                     constraint ?
639 75                 </PropertyConstraint> +
640 76             </PropertyConstraints> ?
641 77
642 78             <Requirements>
643 79                 <Requirement id="xs:ID" name="xs:string" type="xs:QName"> +
644 80                     <Properties>
645 81                         XML fragment
646 82                     <Properties> ?
647 83                     <PropertyConstraints>
648 84                         <PropertyConstraint property="xs:string"
649 85                             constraintType="xs:anyURI"> +
650 86                             constraint ?
651 87                         </PropertyConstraint>
652 88                     </PropertyConstraints> ?
653 89                 </Requirement>
654 90             </Requirements> ?
655 91
656 92             <Capabilities>
657 93                 <Capability id="xs:ID" name="xs:string" type="xs:QName"> +
658 94                     <Properties>
659 95                         XML fragment
660 96                     <Properties> ?
661 97                     <PropertyConstraints>
662 98                         <PropertyConstraint property="xs:string"

```

```

663 99             constraintType="xs:anyURI">
664 100             constraint ?
665 101             </PropertyConstraint> +
666 102             </PropertyConstraints> ?
667 103             </Capability>
668 104             </Capabilities> ?
669 105
670 106             <Policies>
671 107             <Policy name="xs:string"? policyType="xs:QName"
672 108                 policyRef="xs:QName"?>
673 109                 policy specific content ?
674 110             </Policy> +
675 111             </Policies> ?
676 112
677 113             <DeploymentArtifacts>
678 114             <DeploymentArtifact name="xs:string" artifactType="xs:QName"
679 115                 artifactRef="xs:QName"?>
680 116                 artifact specific content ?
681 117             </DeploymentArtifact> +
682 118             </DeploymentArtifacts> ?
683 119             </NodeTemplate>
684 120             |
685 121             <RelationshipTemplate id="xs:ID" name="xs:string"?
686 122                 type="xs:QName">
687 123             <Properties>
688 124                 XML fragment
689 125             </Properties> ?
690 126
691 127             <PropertyConstraints>
692 128             <PropertyConstraint property="xs:string"
693 129                 constraintType="xs:anyURI">
694 130                 constraint ?
695 131             </PropertyConstraint> +
696 132             </PropertyConstraints> ?
697 133
698 134             <SourceElement ref="xs:IDREF"/>
699 135             <TargetElement ref="xs:IDREF"/>
700 136
701 137             <RelationshipConstraints>
702 138             <RelationshipConstraint constraintType="xs:anyURI">
703 139                 constraint ?
704 140             </RelationshipConstraint> +
705 141             </RelationshipConstraints> ?
706 142
707 143             </RelationshipTemplate>
708 144             ) +
709 145             </TopologyTemplate>
710 146
711 147             <Plans>
712 148             <Plan id="xs:ID"
713 149                 name="xs:string"?
714 150                 planType="xs:anyURI"
715 151                 planLanguage="xs:anyURI">
716 152
717 153             <Precondition expressionLanguage="xs:anyURI">
718 154                 condition
719 155             </Precondition> ?
720 156

```

```

721 157     <InputParameters>
722 158         <InputParameter name="xs:string" type="xs:string"
723 159             required="yes|no"?/> +
724 160     </InputParameters> ?
725 161
726 162     <OutputParameters>
727 163         <OutputParameter name="xs:string" type="xs:string"
728 164             required="yes|no"?/> +
729 165     </OutputParameters> ?
730 166
731 167     (
732 168         <PlanModel>
733 169             actual plan
734 170         </PlanModel>
735 171         |
736 172         <PlanModelReference reference="xs:anyURI"/>
737 173     )
738 174
739 175     </Plan> +
740 176 </Plans> ?
741 177
742 178 </ServiceTemplate>

```

743 5.2 Properties

744 The `ServiceTemplate` element has the following properties:

- 745 • `id`: This attribute specifies the identifier of the Service Template which **MUST** be unique within
746 the target namespace.
- 747 • `name`: This **OPTIONAL** attribute specifies a descriptive name of the Service Template.
- 748 • `targetNamespace`: The value of this **OPTIONAL** attribute specifies the target namespace for
749 the Service Template. If not specified, the Service Template will be added to the namespace
750 declared by the `targetNamespace` attribute of the enclosing `Definitions` element.
- 751 • `substitutableNodeType`: This **OPTIONAL** attribute specifies a Node Type that can be
752 substituted by this Service Template. If another Service Template contains a Node Template of
753 the specified Node Type (or any Node Type this Node Type is derived from), this Node Template
754 can be substituted by an instance of this Service Template that then provides the functionality of
755 the substituted node. See section 3.5 for more details.
- 756 • `Tags`: This **OPTIONAL** element allows the definition of any number of tags which can be used by
757 the author to describe the Service Template. Each tag is defined by a separate, nested `Tag`
758 element.

759 The `Tag` element has the following properties:

- 760 ○ `name`: This attribute specifies the name of the tag.
- 761 ○ `value`: This attribute specifies the value of the tag.

762
763 **Note:** The name/value pairs defined in tags have no normative interpretation.

- 764 • `BoundaryDefinitions`: This **OPTIONAL** element specifies the properties the Service
765 Template exposes beyond its boundaries, i.e. properties that can be observed from outside the
766 Service Template. The `BoundaryDefinitions` element has the following properties.
 - 767 ○ `Properties`: This **OPTIONAL** element specifies global properties of the Service
768 Template in the form of an XML fragment contained in the body of the `Properties`
769 element. Those properties **MAY** be mapped to properties of components within the

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

- 772 ▪ `PropertyMappings`: This OPTIONAL element specifies mappings of one or
773 more of the Service Template's properties to properties of components within the
774 Service Template (e.g. Node Templates, Relationship Templates, etc.). Each
775 property mapping is defined by a separate, nested `PropertyMapping`
776 element. The `PropertyMapping` element has the following properties:
 - 777 • `serviceTemplatePropertyRef`: This attribute identifies a property
778 of the Service Template by means of an XPath expression to be
779 evaluated on the XML fragment defining the Service Template's
780 properties.
 - 781 • `targetObjectRef`: This attribute specifies the object that provides
782 the property to which the respective Service Template property is
783 mapped. The referenced target object MUST be one of Node Template,
784 Requirement of a Node Template, Capability of a Node Template, or
785 Relationship Template.
 - 786 • `targetPropertyRef`: This attribute identifies a property of the target
787 object by means of an XPath expression to be evaluated on the XML
788 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
794 Note: If a Service Template property is mapped to a property of a
795 component within the Service Template, reading the Service Template
796 property corresponds to reading the mapped property, and writing the
797 Service Template property corresponds to writing the mapped property.
798
- 799 ○ `PropertyConstraints`: This OPTIONAL element specifies constraints on one or
800 more of the Service Template's properties. Each constraint is specified by means of a
801 separate, nested `PropertyConstraint` element.
802 The `PropertyConstraint` element has the following properties:
 - 803 ▪ `property`: This attribute identifies a property by means of an XPath expression
804 to be evaluated on the XML fragment defining the Service Template's properties.
805
806 Note: If the property affected by the property constraint is mapped to a property
807 of a component within the Service Template, the property constraint SHOULD be
808 compatible with any property constraint defined for the mapped property.
 - 809 ▪ `constraintType`: This attribute specifies the type of constraint by means of a
810 URI, which defines both the semantic meaning of the constraint as well as the
811 format of the content.
 - 812 ▪ The body of the `PropertyConstraint` element provides the actual
813 constraint.
814 Note: The body MAY be empty in case the `constraintType` URI already
815 specifies the constraint appropriately. For example, a "read-only" constraint could
816 be expressed solely by the `constraintType` URI.
- 817 ○ `Requirements`: This OPTIONAL element specifies Requirements exposed by the
818 Service Template. Those Requirements correspond to Requirements of Node Templates
819 within the Service Template that are propagated beyond the boundaries of the Service
820 Template. Each Requirement is defined by a separate, nested `Requirement` element.
821 The `Requirement` element has the following properties:

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

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
924

- `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.

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

929 The `TopologyTemplate` element has the following properties:

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

932 The `NodeTemplate` element has the following properties:

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

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

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

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

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

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

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

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

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

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

970 The `PropertyConstraint` element has the following properties:

- 971
- 972
- 973
- 974
- `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.

975

976

977

978

979

980

981

982

983

984

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.

- 985
- 986
- 987
- 988
- 989
- `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:

- 990
- 991
- 992
- `id`: This attribute specifies the identifier of the Requirement. The identifier of the Requirement MUST be unique within the target namespace.
 - `name`: This attribute specifies the name of the Requirement. The `name` and `type` of the Requirement MUST match the `name` and `type` of a Requirement Definition in the Node Type specified in the `type` attribute of the Node Template.
 - `type`: The QName value of this attribute refers to the Requirement Type definition of the Requirement. This Requirement Type denotes the semantics and well as potential properties of the Requirement.
 - `Properties`: This element specifies initial values for one or more of the Requirement Properties according to the Requirement Type providing the property definitions. Properties are provided in the form of an XML fragment. The same rules as outlined for the `Properties` element of the Node Template apply.
 - `PropertyConstraints`: This element specifies constraints on the use of one or more of the Properties of the Requirement Type providing the property definitions for the Requirement. Each constraint is specified by means of a separate nested `PropertyConstraint` element. The same rules as outlined for the `PropertyConstraints` element of the Node Template apply.

- 1000
- 1001
- 1002
- 1003
- 1004
- 1005
- 1006
- 1007
- 1008
- 1009
- 1010
- `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:

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
- 1062
- `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.

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
1110

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

1111 source element and target element MUST be specified in the Topology Template.
1112 The `RelationshipTemplate` element has the following properties:

- 1113 ▪ `id`: This attribute specifies the identifier of the Relationship Template. The
1114 identifier of the Relationship Template MUST be unique within the target
1115 namespace.
- 1116 ▪ `name`: This OPTIONAL attribute specifies the name of the Relationship
1117 Template.
- 1118 ▪ `type`: The QName value of this property refers to the Relationship Type
1119 providing the type of the Relationship Template.

1120
1121 Note: If the Relationship Type referenced by the `type` attribute of a Relationship
1122 Template is declared as abstract, no instances of the specific Relationship
1123 Template can be created. Instead, a substitution of the Relationship Template
1124 with one having a specialized, derived Relationship Type has to be done at the
1125 latest during the instantiation time of the Relationship Template.

- 1126 ▪ `Properties`: Specifies initial values for one or more of the Relationship Type
1127 Properties of the Relationship Type providing the property definitions in the
1128 concrete context of the Relationship Template.
1129 The initial values are specified by providing an instance document of the XML
1130 schema of the corresponding Relationship Type Properties. This instance
1131 document considers the inheritance structure deduced by the `DerivedFrom`
1132 property of the Relationship Type referenced by the `type` attribute of the
1133 Relationship Template.
1134 The instance document of the XML schema might not validate against the
1135 existence constraints of the corresponding schema: not all Relationship Type
1136 properties might have an initial value assigned, i.e. mandatory elements or
1137 attributes might be missing in the instance provided by the `Properties`
1138 element. Once the defined Relationship Template has been instantiated, any
1139 XML representation of the Relationship Type properties MUST validate according
1140 to the associated XML schema definition.

- 1141 ▪ `PropertyConstraints`: Specifies constraints on the use of one or more of
1142 the Relationship Type Properties of the Relationship Type providing the property
1143 definitions for the Relationship Template. Each constraint is specified by means
1144 of a separate nested `PropertyConstraint` element.

1145 The `PropertyConstraint` element has the following properties:

- 1146 • `property`: The string value of this property is an XPath expression
1147 pointing to the property within the Relationship Type Properties
1148 document that is constrained within the context of the Relationship
1149 Template. More than one constraint MUST NOT be defined for each
1150 property.
- 1151 • `constraintType`: The constraint type is specified by means of a URI,
1152 which defines both the semantic meaning of the constraint as well as the
1153 format of the content.

1154 For example, a constraint type of
1155 <http://www.example.com/PropertyConstraints/unique> could denote that
1156 the reference property of the node template under definition has to be
1157

1158 unique within a certain scope. The constraint type specific content of the
1159 respective `PropertyConstraint` element could then define the
1160 actual scope in which uniqueness has to be ensured in more detail.

1161 ▪ `SourceElement`: This element specifies the origin of the relationship
1162 represented by the current `Relationship Template`.

1163 The `SourceElement` element has the following property:

- 1164 • `ref`: This attribute references by ID a Node Template or a Requirement
1165 of a Node Template within the same Service Template document that is
1166 the source of the `Relationship Template`.

1167
1168 If the `Relationship Type` referenced by the `type` attribute defines a
1169 constraint on the valid source of the relationship by means of its
1170 `ValidSource` element, the `ref` attribute of `SourceElement` MUST
1171 reference an object the type of which complies with the valid source
1172 constraint of the respective `Relationship Type`.

1173
1174 In the case where a Node Type is defined as valid source in the
1175 `Relationship Type` definition, the `ref` attribute MUST reference a Node
1176 Template of the corresponding Node Type (or of a sub-type).

1177
1178 In the case where a Requirement Type is defined a valid source in the
1179 `Relationship Type` definition, the `ref` attribute MUST reference a
1180 Requirement of the corresponding Requirement Type within a Node
1181 Template.

1182 ▪ `TargetElement`: This element specifies the target of the relationship
1183 represented by the current `Relationship Template`.

1184 The `TargetElement` element has the following property:

- 1185 • `ref`: This attribute references by ID a Node Template or a Capability of
1186 a Node Template within the same Service Template document that is the
1187 target of the `Relationship Template`.

1188
1189 If the `Relationship Type` referenced by the `type` attribute defines a
1190 constraint on the valid source of the relationship by means of its
1191 `ValidTarget` element, the `ref` attribute of `TargetElement` MUST
1192 reference an object the type of which complies with the valid source
1193 constraint of the respective `Relationship Type`.

1194
1195 In case a Node Type is defined as valid target in the `Relationship Type`
1196 definition, the `ref` attribute MUST reference a Node Template of the
1197 corresponding Node Type (or of a sub-type).

1198
1199 In case a Capability Type is defined a valid target in the `Relationship`
1200 `Type` definition, the `ref` attribute MUST reference a Capability of the
1201 corresponding Capability Type within a Node Template.

1202 ▪ `RelationshipConstraints`: This element specifies a list of constraints on
1203 the use of the relationship in separate nested `RelationshipConstraint`
1204 elements.

1205 The `RelationshipConstraint` element has the following properties:

- 1206
- 1207
- 1208
- 1209
- `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.
- 1210
- `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.
- 1211
- 1212
- 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.
- 1213
- 1214
- 1215
- 1216
- 1217
- 1218
- 1219
- 1220
- 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.
- 1221
- 1222
- 1223
- 1224
- 1225
- 1226
- 1227
- 1228
- 1229
- 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.
- 1230
- 1231
- 1232
- 1233
- 1234
- 1235
- 1236
- 1237
- 1238
- TOSCA does not specify a separate metamodel for defining plans. Instead, it is assumed that a process modelling language (a.k.a. metamodel) like BPEL [BPEL 2.0] or BPMN [BPMN 2.0] is used to define plans. The specification favours the use of BPMN for modeling plans.
- `Precondition`: This OPTIONAL element specifies a condition that needs to be satisfied in order for the plan to be executed. The `expressionLanguage` attribute of this element specifies the expression language the nested condition is provided in.
- 1239
- 1240
- 1241
- 1242
- 1243
- 1244
- 1245
- 1246
- 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.
- 1247
- 1248
- 1249
- 1250
- The `InputParameter` element has the following properties:

- 1251 ▪ name: This attribute specifies the name of the input parameter, which MUST be
- 1252 unique within the set of input parameters defined for the operation.
- 1253 ▪ type: This attribute specifies the type of the input parameter.
- 1254 ▪ required: This OPTIONAL attribute specifies whether or not the input
- 1255 parameter is REQUIRED (required attribute with a value of “yes” – default) or
- 1256 OPTIONAL (required attribute with a value of “no”).
- 1257 ○ OutputParameters: This OPTIONAL property contains a list of one or more output
- 1258 parameter definitions for the Plan, each defined in a nested, separate
- 1259 OutputParameter element.
- 1260 The OutputParameter element has the following properties:
- 1261 ▪ name: This attribute specifies the name of the output parameter, which MUST be
- 1262 unique within the set of output parameters defined for the operation.
- 1263 ▪ type: This attribute specifies the type of the output parameter.
- 1264 ▪ required: This OPTIONAL attribute specifies whether or not the output
- 1265 parameter is REQUIRED (required attribute with a value of “yes” – default) or
- 1266 OPTIONAL (required attribute with a value of “no”).
- 1267 ○ PlanModel: This property contains the actual model content.
- 1268 ○ PlanModelReference: This property points to the model content. Its reference
- 1269 attribute contains a URI of the model of the plan.
- 1270
- 1271 An instance of the Plan element MUST either contain the actual plan as instance of the
- 1272 PlanModel element, or point to the model via the PlanModelReference element.

1273 5.3 Example

1274 The following Service Template defines a Topology Template containing two Node Templates called
 1275 “MyApplication” and “MyAppServer”. These Node Templates have the node types “Application” and
 1276 “ApplicationServer”. The Node Template “MyApplication” is instantiated exactly once. Two of its Node
 1277 Type Properties are initialized by a corresponding Properties element. The Node Template
 1278 “MyAppServer” can be instantiated as many times as needed. The “MyApplication” Node Template is
 1279 connected with the “MyAppServer” Node Template via the Relationship Template named
 1280 “MyHostedRelationship”; the behavior and semantics of the Relationship Template is defined in the
 1281 Relationship Type “HostedOn”, saying that “MyApplication” is hosted on “MyAppServer”. The Service
 1282 Template further defines a Plan “UpdateApplication” for performing an update of the “MyApplication”
 1283 application hosted on the application server. This Plan refers to a BPMN 2.0 process definition contained
 1284 in a separate file.

```

1285 01 <ServiceTemplate id="MyService"
1286 02     name="My Service">
1287 03
1288 04   <TopologyTemplate>
1289 05
1290 06     <NodeTemplate id="MyApplication"
1291 07         name="My Application"
1292 08         type="my:Application">
1293 09       <Properties>
1294 10         <ApplicationProperties>
1295 11           <Owner>Frank</Owner>
1296 12           <InstanceName>Thomas' favorite application</InstanceName>
1297 13         </ApplicationProperties>
1298 14       </Properties>

```

```
1299 15     </NodeTemplate>
1300 16
1301 17     <NodeTemplate id="MyAppServer"
1302 18         name="My Application Server"
1303 19         type="my:ApplicationServer"
1304 20         minInstances="0"
1305 21         maxInstances="unbounded"/>
1306 22
1307 23     <RelationshipTemplate id="MyDeploymentRelationship"
1308 24         type="my:deployedOn">
1309 25         <SourceElement ref="MyApplication"/>
1310 26         <TargetElement ref="MyAppServer"/>
1311 27     </RelationshipTemplate>
1312 28
1313 29 </TopologyTemplate>
1314 30
1315 31 <Plans>
1316 32     <Plan id="UpdateApplication"
1317 33         planType="http://www.example.com/UpdatePlan"
1318 34         planLanguage="http://www.omg.org/spec/BPMN/20100524/MODEL">
1319 35         <PlanModelReference reference="plans:UpdateApp"/>
1320 36     </Plan>
1321 37 </Plans>
1322 38
1323 39 </ServiceTemplate>
```


1324

6 Node Types

1325 This chapter specifies how *Node Types* are defined. A Node Type is a reusable entity that defines the
1326 type of one or more Node Templates. As such, a Node Type defines the structure of observable
1327 properties via a *Properties Definition*, i.e. the names, data types and allowed values the properties
1328 defined in Node Templates using a Node Type or instances of such Node Templates can have.

1329 A Node Type can inherit properties from another Node Type by means of the `DerivedFrom` element.
1330 Node Types might be declared as abstract, meaning that they cannot be instantiated. The purpose of
1331 such abstract Node Types is to provide common properties and behavior for re-use in specialized,
1332 derived Node Types. Node Types might also be declared as final, meaning that they cannot be derived by
1333 other Node Types.

1334 A Node Type can declare to expose certain requirements and capabilities (see section 3.4) by means of
1335 `RequirementDefinition` elements or `CapabilityDefinition` elements, respectively.

1336 The functions that can be performed on (an instance of) a corresponding Node Template are defined by
1337 the *Interfaces* of the Node Type. Finally, management Policies are defined for a Node Type.

1338 6.1 XML Syntax

1339 The following pseudo schema defines the XML syntax of Node Types:

```
1340 01 <NodeType name="xs:NCName" targetNamespace="xs:anyURI"?  
1341 02     abstract="yes|no"? final="yes|no"?>  
1342 03  
1343 04 <Tags>  
1344 05     <Tag name="xs:string" value="xs:string"/> +  
1345 06 </Tags> ?  
1346 07  
1347 08 <DerivedFrom typeRef="xs:QName"/> ?  
1348 09  
1349 10 <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?  
1350 11  
1351 12 <RequirementDefinitions>  
1352 13     <RequirementDefinition name="xs:string"  
1353 14         requirementType="xs:QName"  
1354 15         lowerBound="xs:integer"?  
1355 16         upperBound="xs:integer | xs:string"?>  
1356 17         <Constraints>  
1357 18             <Constraint constraintType="xs:anyURI">  
1358 19                 constraint type specific content  
1359 20             </Constraint> +  
1360 21         </Constraints> ?  
1361 22     </RequirementDefinition> +  
1362 23 </RequirementDefinitions> ?  
1363 24  
1364 25 <CapabilityDefinitions>  
1365 26     <CapabilityDefinition name="xs:string"  
1366 27         capabilityType="xs:QName"  
1367 28         lowerBound="xs:integer"?  
1368 29         upperBound="xs:integer | xs:string"?>  
1369 30         <Constraints>  
1370 31             <Constraint constraintType="xs:anyURI">  
1371 32                 constraint type specific content  
1372 33             </Constraint> +  
1373 34         </Constraints> ?
```

```

1374 35     </CapabilityDefinition> +
1375 36 </CapabilityDefinitions>
1376 37
1377 38 <InstanceStates>
1378 39     <InstanceState state="xs:anyURI"> +
1379 40 </InstanceStates> ?
1380 41
1381 42 <Interfaces>
1382 43     <Interface name="xs:NCName | xs:anyURI">
1383 44         <Operation name="xs:NCName">
1384 45             <InputParameters>
1385 46                 <InputParameter name="xs:string" type="xs:string"
1386 47                     required="yes|no"?/> +
1387 48             </InputParameters> ?
1388 49             <OutputParameters>
1389 50                 <OutputParameter name="xs:string" type="xs:string"
1390 51                     required="yes|no"?/> +
1391 52             </OutputParameters> ?
1392 53         </Operation> +
1393 54     </Interface> +
1394 55 </Interfaces> ?
1395 56
1396 57 </NodeType>

```

6.2 Properties

1397

1398 The `NodeType` element has the following properties:

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

1403

1404 As a consequence, the corresponding abstract Node Type referenced by any Node Template has to be substituted by a Node Type derived from the abstract Node Type at the latest during the instantiation time of a Node Template.

1405

1406 Note: an abstract Node Type MUST NOT be declared as final.

- 1407 • `final`: This OPTIONAL attribute specifies that other Node Types MUST NOT be derived from this Node Type.

1408

1409 Note: a final Node Type MUST NOT be declared as abstract.

- 1410 • `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by the author to describe the Node Type. Each tag is defined by a separate, nested `Tag` element. The `Tag` element has the following properties:

- 1411
 - 1412 ○ `name`: This attribute specifies the name of the tag.

- 1413
 - 1414 ○ `value`: This attribute specifies the value of the tag.

1415

1416 Note: The name/value pairs defined in tags have no normative interpretation.

- 1425
- 1426
- 1427
- 1428
- **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.

1474 Note that one Node Type might define multiple capabilities of the same Capability Type,
1475 in which case each occurrence of a capability definition is uniquely identified by its name.

- 1476 ○ `capabilityType`: This attribute identifies by QName the Capability Type of capability
1477 that is being defined by the current `CapabilityDefinition`.
- 1478 ○ `lowerBound`: This OPTIONAL attribute specifies the lower boundary of requiring nodes
1479 that the defined capability can serve. The default value for this attribute is one. A value of
1480 zero is invalid, since this would mean that the capability cannot actually satisfy any
1481 requiring nodes.
- 1482 ○ `upperBound`: This OPTIONAL attribute specifies the upper boundary of client
1483 requirements the defined capability can serve. The default value for this attribute is one.
1484 A value of "unbounded" indicates that there is no upper boundary.
- 1485 ○ `Constraints`: This OPTIONAL element contains a list of `Constraint` elements that
1486 specify additional constraints on the capability definition.
1487 The nested `Constraint` element has the following properties:
 - 1488 ▪ `constraintType`: This attribute specifies the type of constraint. According to
1489 this type, the body of the `Constraint` element will contain type specific
1490 content.
- 1491 • `InstanceStates`: This OPTIONAL element lists the set of states an instance of this Node
1492 Type can occupy. Those states are defined in nested `InstanceState` elements.
1493 The `InstanceState` element has the following nested properties:
 - 1494 ○ `state`: This attribute specifies a URI that identifies a potential state.
- 1495 • `Interfaces`: This element contains the definitions of the operations that can be performed on
1496 (instances of) this Node Type. Such operation definitions are given in the form of nested
1497 `Interface` elements.
1498 The `Interface` element has the following properties:
 - 1499 ○ `name`: The name of the interface. This name is either a URI or it is an NCName that
1500 MUST be unique in the scope of the Node Type being defined.
 - 1501 ○ `Operation`: This element defines an operation available to manage particular aspects
1502 of the Node Type.
1503
1504 The `Operation` element has the following properties:
 - 1505 ▪ `name`: This attribute defines the name of the operation and MUST be unique
1506 within the containing `Interface` of the Node Type.
 - 1507 ▪ `InputParameters`: This OPTIONAL property contains a list of one or more
1508 input parameter definitions, each defined in a nested, separate
1509 `InputParameter` element.
1510 The `InputParameter` element has the following properties:
 - 1511 • `name`: This attribute specifies the name of the input parameter, which
1512 MUST be unique within the set of input parameters defined for the
1513 operation.
 - 1514 • `type`: This attribute specifies the type of the input parameter.
 - 1515 • `required`: This OPTIONAL attribute specifies whether or not the input
1516 parameter is REQUIRED (`required` attribute with a value of "yes" –
1517 default) or OPTIONAL (`required` attribute with a value of "no").
 - 1518 ▪ `OutputParameters`: This OPTIONAL property contains a list of one or more
1519 output parameter definitions, each defined in a nested, separate
1520 `OutputParameter` element.
1521 The `OutputParameter` element has the following properties:

- 1522 • `name`: This attribute specifies the name of the output parameter, which
1523 MUST be unique within the set of output parameters defined for the
1524 operation.
- 1525 • `type`: This attribute specifies the type of the output parameter.
- 1526 • `required`: This OPTIONAL attribute specifies whether or not the
1527 output parameter is REQUIRED (`required` attribute with a value of
1528 “yes” – default) or OPTIONAL (`required` attribute with a value of “no”).

1529 6.3 Derivation Rules

1530 The following rules on combining definitions based on `DerivedFrom` apply:

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

1537
1538 In cases where the Node Type under definition defines a requirement or capability with a certain
1539 name where the Node Type derived from already contains a respective definition with the same
1540 name, the definition in the Node Type under definition overrides the definition of the Node Type
1541 derived from. In such a case, the requirement definition or capability definition, respectively,
1542 MUST reference a Requirement Type or Capability Type that is derived from the one in the
1543 corresponding requirement definition or capability definition of the Node Type derived from.

- 1544 • **Instance States:** The set of instance states of the Node Type under definition consists of the set
1545 union of the instances states defined by the Nodes Type derived from and the instance states
1546 defined by the Node Type under definition. A set of instance states of the same name will be
1547 combined into a single instance state of the same name.
- 1548 • **Interfaces:** The set of interfaces of the Node Type under definition consists of the set union of
1549 interfaces defined by the Node Type derived from and the interfaces defined by the Node Type
1550 under definition.
1551 Two interfaces of the same name will be combined into a single, derived interface with the same
1552 name. The set of operations of the derived interface consists of the set union of operations
1553 defined by both interfaces. An operation defined by the Node Type under definition substitutes an
1554 operation with the same name of the Node Type derived from.

1555 6.4 Example

1556 The following example defines the Node Type “Project”. It is defined in a Definitions document
1557 “MyDefinitions” within the target namespace “http://www.example.com/sample”. Thus, by importing the
1558 corresponding namespace in another Definitions document, the Project Node Type is available for use in
1559 the other document.

```
1560 01 <Definitions id="MyDefinitions" name="My Definitions"
1561 02     targetNamespace="http://www.example.com/sample">
1562 03
1563 04   <NodeType name="Project">
1564 05
1565 06     <documentation xml:lang="EN">
1566 07       A reusable definition of a node type supporting
1567 08       the creation of new projects.
```

```

1568 09     </documentation>
1569 10
1570 11     <PropertiesDefinition element="ProjectProperties"/>
1571 12
1572 13     <InstanceStates>
1573 14         <InstanceState state="www.example.com/active"/>
1574 15         <InstanceState state="www.example.com/onHold"/>
1575 16     </InstanceStates>
1576 17
1577 18     <Interfaces>
1578 19         <Interface name="ProjectInterface">
1579 20             <Operation name="CreateProject">
1580 21                 <InputParameters>
1581 22                     <InputParamter name="ProjectName"
1582 23                         type="xs:string"/>
1583 24                     <InputParamter name="Owner"
1584 25                         type="xs:string"/>
1585 26                     <InputParamter name="AccountID"
1586 27                         type="xs:string"/>
1587 28                 </InputParameters>
1588 29             </Operation>
1589 30         </Interface>
1590 31     </Interfaces>
1591 32 </NodeType>
1592 33
1593 34 </Definitions>

```

1594 The Node Type "Project" has three Node Type Properties defined as an XML element in the `Types`
1595 element definition of the Service Template document: `Owner`, `ProjectName` and `AccountID` which are all
1596 of type `xs:string`. An instance of the Node Type "Project" could be "active" (more precise in state
1597 `www.example.com/active`) or "on hold" (more precise in state `www.example.com/onHold`). A single
1598 Interface is defined for this Node Type, and this Interface is defined by an Operation, i.e. its actual
1599 implementation is defined by the definition of the Operation. The Operation has the name `CreateProject`
1600 and three Input Parameters (exploiting the default value "yes" of the attribute `required` of the
1601 `InputParameter` element). The names of these Input Parameters are `ProjectName`, `Owner` and
1602 `AccountID`, all of type `xs:string`.

1603 7 Node Type Implementations

1604 This chapter specifies how *Node Type Implementations* are defined. A Node Type Implementation
1605 represents the executable code that implements a specific Node Type. It provides a collection of
1606 executables implementing the interface operations of a Node Type (aka implementation artifacts) and the
1607 executables needed to materialize instances of Node Templates referring to a particular Node Type (aka
1608 deployment artifacts). The respective executables are defined as separate Artifact Templates and are
1609 referenced from the implementation artifacts and deployment artifacts of a Node Type Implementation.

1610 While Artifact Templates provide invariant information about an artifact – i.e. information that is context
1611 independent like the file name of the artifact – implementation or deployment artifacts can provide variant
1612 (or context specific) information, such as authentication data or deployment paths for a specific
1613 environment.

1614 Node Type Implementations can specify hints for a TOSCA container that enable proper selection of an
1615 implementation that fits into a particular environment by means of Required Container Features
1616 definitions.

1617 7.1 XML Syntax

1618 The following pseudo schema defines the XML syntax of Node Type Implementations:

```
1619 01 <NodeTypeImplementation name="xs:NCName" targetNamespace="xs:anyURI"?  
1620 02     nodeType="xs:QName"  
1621 03     abstract="yes|no"?  
1622 04     final="yes|no"?>  
1623 05  
1624 06 <Tags>  
1625 07     <Tag name="xs:string" value="xs:string"/> +  
1626 08 </Tags> ?  
1627 09  
1628 10 <DerivedFrom nodeTypeImplementationRef="xs:QName"/> ?  
1629 11  
1630 12 <RequiredContainerFeatures>  
1631 13     <RequiredContainerFeature feature="xs:anyURI"/> +  
1632 14 </RequiredContainerFeatures> ?  
1633 15  
1634 16 <ImplementationArtifacts>  
1635 17     <ImplementationArtifact interfaceName="xs:NCName | xs:anyURI"?  
1636 18         operationName="xs:NCName"?  
1637 19         artifactType="xs:QName"  
1638 20         artifactRef="xs:QName"?>  
1639 21         artifact specific content ?  
1640 22     <ImplementationArtifact> +  
1641 23 </ImplementationArtifacts> ?  
1642 24  
1643 25 <DeploymentArtifacts>  
1644 26     <DeploymentArtifact name="xs:string" artifactType="xs:QName"  
1645 27         artifactRef="xs:QName"?>  
1646 28         artifact specific content ?  
1647 29     <DeploymentArtifact> +  
1648 30 </DeploymentArtifacts> ?  
1649 31  
1650 32 </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
- `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
- 1800
- 1801
- 1802
- 1803
- 1804
- 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.

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 interfaceName="MgmtInterface"  
1828 21         artifactType="ba:WARFile"  
1829 22         artifactRef="sa:MyMgmtWebApp">  
1830 23     </ImplementationArtifact>  
1831 24   </ImplementationArtifacts>  
1832 25  
1833 26   <DeploymentArtifacts>  
1834 27     <DeploymentArtifact name="MyDBMS"  
1835 28       artifactType="ba:ZipFile"  
1836 29       artifactRef="sa:MyInstallable">  
1837 30     </DeploymentArtifact>  
1838 31   </DeploymentArtifacts>  
1839 32  
1840 33   </NodeTypeImplementation>  
1841 34  
1842 35 </Definitions>
```

1843 The Node Type Implementation contains the “MyDBMSManagement” implementation artifact, which is an
1844 artifact for the “MgmtInterface” Interface that has been defined for the “DBMS” base Node Type. The type
1845 of this artifact is a “WARFile” that has been defined as base Artifact Type. The implementation artifact
1846 refers to the “MyMgmtWebApp” Artifact Template that has been defined before.

1847 The Node Type Implementation further contains the “MyDBMS” deployment artifact, which is a software
1848 installable used for instantiating the “DBMS” Node Type. This software installable is a “ZipFile” that has
1849 been separately defined as the “MyInstallable” Artifact Template before.

1850 8 Relationship Types

1851 This chapter specifies how *Relationship Types* are defined. A Relationship Type is a reusable entity that
1852 defines the type of one or more Relationship Templates between Node Templates. As such, a
1853 Relationship Type can define the structure of observable properties via a *Properties Definition*, i.e. the
1854 names, data types and allowed values the properties defined in Relationship Templates using a
1855 Relationship Type or instances of such Relationship Templates can have.

1856 The operations that can be performed on (an instance of) a corresponding Relationship Template are
1857 defined by the *Interfaces* of the Relationship Type. Furthermore, a Relationship Type defines the potential
1858 states an instance of it might reveal at runtime.

1859 A Relationship Type can inherit the definitions defined in another Relationship Type by means of the
1860 *DerivedFrom* element. Relationship Types might be declared as abstract, meaning that they cannot be
1861 instantiated. The purpose of such abstract Relationship Types is to provide common properties and
1862 behavior for re-use in specialized, derived Relationship Types. Relationship Types might also be declared
1863 as final, meaning that they cannot be derived by other Relationship Types.

1864 8.1 XML Syntax

1865 The following pseudo schema defines the XML syntax of Relationship Types:

```
1866 01 <RelationshipType name="xs:NCName"  
1867 02         targetNamespace="xs:anyURI"?  
1868 03         abstract="yes|no"?  
1869 04         final="yes|no"?> +  
1870 05  
1871 06 <Tags>  
1872 07   <Tag name="xs:string" value="xs:string"/> +  
1873 08 </Tags> ?  
1874 09  
1875 10 <DerivedFrom typeRef="xs:QName"/> ?  
1876 11  
1877 12 <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?  
1878 13  
1879 14 <InstanceStates>  
1880 15   <InstanceState state="xs:anyURI"> +  
1881 16 </InstanceStates> ?  
1882 17  
1883 18 <SourceInterfaces>  
1884 19   <Interface name="xs:NCName | xs:anyURI">  
1885 20     ...  
1886 21   </Interface> +  
1887 22 </SourceInterfaces> ?  
1888 23  
1889 24 <TargetInterfaces>  
1890 25   <Interface name="xs:NCName | xs:anyURI">  
1891 26     ...  
1892 27   </Interface> +  
1893 28 </TargetInterfaces> ?  
1894 29  
1895 30 <ValidSource typeRef="xs:QName"/> ?  
1896 31  
1897 32 <ValidTarget typeRef="xs:QName"/> ?  
1898 33  
1899 34 </RelationshipType>
```

1900 8.2 Properties

1901 The `RelationshipType` element has the following properties:

- 1902 • `name`: This attribute specifies the name or identifier of the Relationship Type, which MUST be
1903 unique within the target namespace.
- 1904 • `targetNamespace`: This OPTIONAL attribute specifies the target namespace to which the
1905 definition of the Relationship Type will be added. If not specified, the Relationship Type definition
1906 will be added to the target namespace of the enclosing Definitions document.
- 1907 • `abstract`: This OPTIONAL attribute specifies that no instances can be created from
1908 Relationship Templates that use this Relationship Type as their type.
1909

1910 As a consequence, the corresponding abstract Relationship Type referenced by any Relationship
1911 Template has to be substituted by a Relationship Type derived from the abstract Relationship
1912 Type at the latest during the instantiation time of a Relationship Template.
1913

1914 Note: an abstract Relationship Type MUST NOT be declared as final.

- 1915 • `final`: This OPTIONAL attribute specifies that other Relationship Types MUST NOT be derived
1916 from this Relationship Type.
1917

1918 Note: a final Relationship Type MUST NOT be declared as abstract.

- 1919 • `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by
1920 the author to describe the Relationship Type. Each tag is defined by a separate, nested `Tag`
1921 element.

1922 The `Tag` element has the following properties:

- 1923 ○ `name`: This attribute specifies the name of the tag.
- 1924 ○ `value`: This attribute specifies the value of the tag.
1925

1926 Note: The name/value pairs defined in tags have no normative interpretation.

- 1927 • `DerivedFrom`: This is an OPTIONAL reference to another Relationship Type from which this
1928 Relationship Type is derived. Conflicting definitions are resolved by the rule that local new
1929 definitions always override derived definitions. See section 8.3 Derivation Rules for details.

1930 The `DerivedFrom` element has the following properties:

- 1931 ○ `typeRef`: The QName specifies the Relationship Type from which this Relationship
1932 Type derives its definitions.

- 1933 • `PropertiesDefinition`: This element specifies the structure of the observable properties
1934 of the Relationship Type, such as its configuration and state, by means of XML schema.

1935 The `PropertiesDefinition` element has one but not both of the following properties:

- 1936 ○ `element`: This attribute provides the QName of an XML element defining the structure
1937 of the Relationship Type Properties.
- 1938 ○ `type`: This attribute provides the QName of an XML (complex) type defining the
1939 structure of the Relationship Type Properties.

- 1940 • `InstanceStates`: This OPTIONAL element lists the set of states an instance of this
1941 Relationship Type can occupy at runtime. Those states are defined in nested `InstanceState`
1942 elements.

1943 The `InstanceState` element has the following nested properties:

- 1944 ○ `state`: This attribute specifies a URI that identifies a potential state.

- 1945 • `SourceInterfaces`: This OPTIONAL element contains definitions of manageability interfaces
1946 that can be performed on the source of a relationship of this Relationship Type to actually
1947 establish the relationship between the source and the target in the deployed service.

1948 Those interface definitions are contained in nested `Interface` elements, the content of which
1949 is that described for Node Type interfaces (see section 6.2).

1950 • `TargetInterfaces`: This OPTIONAL element contains definitions of manageability interfaces
1951 that can be performed on the target of a relationship of this Relationship Type to actually
1952 establish the relationship between the source and the target in the deployed service.

1953 Those interface definitions are contained in nested `Interface` elements, the content of which
1954 is that described for Node Type interfaces (see section 6.2).

1955 • `ValidSource`: This OPTIONAL element specifies the type of object that is allowed as a valid
1956 origin for relationships defined using the Relationship Type under definition. If not specified, any
1957 Node Type is allowed to be the origin of the relationship.

1958 The `ValidSource` element has the following properties:

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

1963
1964 Note: If `ValidSource` specifies a Node Type, the `ValidTarget` element (if present)
1965 of the Relationship Type under definition MUST also specify a Node Type.

1966 If `ValidSource` specifies a Requirement Type, the `ValidTarget` element (if
1967 present) of the Relationship Type under definition MUST specify a Capability Type. This
1968 Capability Type MUST match the requirement defined in `ValidSource`, i.e. it MUST
1969 be of the type (or a sub-type of) the capability specified in the
1970 `requiredCapabilityType` attribute of the respective `RequirementType`
1971 definition.

1972 • `ValidTarget`: This OPTIONAL element specifies the type of object that is allowed as a valid
1973 target for relationships defined using the Relationship Type under definition. If not specified, any
1974 Node Type is allowed to be the origin of the relationship.

1975 The `ValidTarget` element has the following properties:

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

1980
1981 Note: If `ValidTarget` specifies a Node Type, the `ValidSource` element (if present)
1982 of the Relationship Type under definition MUST also specify a Node Type.

1983 If `ValidTarget` specifies a Capability Type, the `ValidSource` element (if present)
1984 of the Relationship Type under definition MUST specify a Requirement Type. This
1985 Requirement Type MUST declare it requires the capability defined in `ValidTarget`,
1986 i.e. it MUST declare the type (or a super-type of) the capability in the
1987 `requiredCapabilityType` attribute of the respective `RequirementType`
1988 definition.

1989 8.3 Derivation Rules

1990 The following rules on combining definitions based on `DerivedFrom` apply:

1991 • `Relationship Type Properties`: It is assumed that the XML element (or type) representing the
1992 Relationship Type properties of the Relationship Type under definition extends the XML element
1993 (or type) of the Relationship Type properties of the Relationship Type referenced in the
1994 `DerivedFrom` element.

1995 • `Instance States`: The resulting set of instance states of the Relationship Type under definition
1996 consists of the set union of the instances states defined by the Relationship Type derived from

- 1997 and the instance states explicitly defined by the Relationship Type under definition. Instance
 1998 states with the same state attribute will be combined into a single instance state of the same
 1999 state.
- 2000 • Valid source and target: An object specified as a valid source or target, respectively, of the
 2001 Relationship Type under definition MUST be of a subtype defined as valid source or target,
 2002 respectively, of the Relationship Type derived from.
 2003
- 2004 If the Relationship Type derived from has no valid source or target defined, the types of object
 2005 being defined in the `ValidSource` or `ValidTarget` elements of the Relationship Type
 2006 under definition are not restricted.
 2007
- 2008 If the Relationship Type under definition has no source or target defined, only the types of objects
 2009 defined as source or target of the Relationship Type derived from are valid origins or destinations
 2010 of the Relationship Type under definition.
- 2011 • Interfaces: The set of interfaces (both source and target interfaces) of the Relationship Type
 2012 under definition consists of the set union of interfaces defined by the Relationship Type derived
 2013 from and the interfaces defined by the Relationship Type under definition.
 2014 Two interfaces of the same name will be combined into a single, derived interface with the same
 2015 name. The set of operations of the derived interface consists of the set union of operations
 2016 defined by both interfaces. An operation defined by the Relationship Type under definition
 2017 substitutes an operation with the same name of the Relationship Type derived from.

2018 8.4 Example

2019 The following example defines the Relationship Type “processDeployedOn”. The meaning of this
 2020 Relationship Type is that “a process is deployed on a hosting environment”. When the source of an
 2021 instance of a Relationship Template referring to this Relationship Type is deleted, its target is
 2022 automatically deleted as well. The Relationship Type has Relationship Type Properties defined in the
 2023 `Types` section of the same Definitions document as the “ProcessDeployedOnProperties” element. The
 2024 states an instance of this Relationship Type can be in are also listed.

```

2025 01 <RelationshipType name="processDeployedOn">
2026 02
2027 03   <RelationshipTypeProperties element="ProcessDeployedOnProperties"/>
2028 04
2029 05   <InstanceStates>
2030 06     <InstanceState state="www.example.com/successfullyDeployed"/>
2031 07     <InstanceState state="www.example.com/failed"/>
2032 08   </InstanceStates>
2033 09
2034 10 </RelationshipType>
  
```


2035

9 Relationship Type Implementations

2036 This chapter specifies how *Relationship Type Implementations* are defined. A Relationship Type
2037 Implementation represents the runnable code that implements a specific Relationship Type. It provides a
2038 collection of executables implementing the interface operations of a Relationship Type (aka
2039 implementation artifacts). The particular executables are defined as separate Artifact Templates and are
2040 referenced from the implementation artifacts of a Relationship Type Implementation.

2041 While Artifact Templates provide invariant information about an artifact – i.e. information that is context
2042 independent like the file name of the artifact – implementation artifacts can provide variant (or context
2043 specific) information, e.g. authentication data for a specific environment.

2044 Relationship Type Implementations can specify hints for a TOSCA container that enable proper selection
2045 of an implementation that fits into a particular environment by means of Required Container Features
2046 definitions.

2047 Note that there MAY be Relationship Types that do not define any interface operations, i.e. that also do
2048 not require any implementation artifacts. In such cases, no Relationship Type Implementation is needed
2049 but the respective Relationship Types can be used by a TOSCA implementation as is.

9.1 XML Syntax

2051 The following pseudo schema defines the XML syntax of Relationship Type Implementations:

```
2052 01 <RelationshipTypeImplementation name="xs:NCName"
2053 02         targetNamespace="xs:anyURI"?
2054 03         relationshipType="xs:QName"
2055 04         abstract="yes|no"?
2056 05         final="yes|no"?>
2057 06
2058 07 <Tags>
2059 08   <Tag name="xs:string" value="xs:string" /> +
2060 09 </Tags> ?
2061 10
2062 11 <DerivedFrom relationshipTypeImplementationRef="xs:QName" /> ?
2063 12
2064 13 <RequiredContainerFeatures>
2065 14   <RequiredContainerFeature feature="xs:anyURI" /> +
2066 15 </RequiredContainerFeatures> ?
2067 16
2068 17 <ImplementationArtifacts>
2069 18   <ImplementationArtifact interfaceName="xs:NCName | xs:anyURI"?
2070 19         operationName="xs:NCName"?
2071 20         artifactType="xs:QName"
2072 21         artifactRef="xs:QName"?>
2073 22     artifact specific content ?
2074 23   <ImplementationArtifact> +
2075 24 </ImplementationArtifacts> ?
2076 25
2077 26 </RelationshipTypeImplementation>
```

9.2 Properties

2079 The RelationshipTypeImplementation element has the following properties:

- 2080 • name: This attribute specifies the name or identifier of the Relationship Type Implementation,
2081 which MUST be unique within the target namespace.

2082 • `targetNamespace`: This OPTIONAL attribute specifies the target namespace to which the
2083 definition of the Relationship Type Implementation will be added. If not specified, the Relationship
2084 Type Implementation will be added to the target namespace of the enclosing Definitions
2085 document.

2086 • `relationshipType`: The QName value of this attribute specifies the Relationship Type
2087 implemented by this Relationship Type Implementation.

2088 • `abstract`: This OPTIONAL attribute specifies that this Relationship Type Implementation
2089 cannot be used directly as an implementation for the Relationship Type specified in the
2090 `relationshipType` attribute.

2091
2092 For example, a Relationship Type implementer might decide to deliver only part of the
2093 implementation of a specific Relationship Type (i.e. for only some operations) for re-use purposes
2094 and require the implementation for specific operations to be delivered in a more concrete, derived
2095 Relationship Type Implementation.
2096

2097 Note: an abstract Relationship Type Implementation MUST NOT be declared as final.

2098 • `final`: This OPTIONAL attribute specifies that other Relationship Type Implementations MUST
2099 NOT be derived from this Relationship Type Implementation.
2100

2101 Note: a final Relationship Type Implementation MUST NOT be declared as abstract.

2102 • `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by
2103 the author to describe the Relationship Type Implementation. Each tag is defined by a separate,
2104 nested `Tag` element.

2105 The `Tag` element has the following properties:

2106 ○ `name`: This attribute specifies the name of the tag.

2107 ○ `value`: This attribute specifies the value of the tag.

2108
2109 Note: The name/value pairs defined in tags have no normative interpretation.

2110 • `DerivedFrom`: This is an OPTIONAL reference to another Relationship Type Implementation
2111 from which this Relationship Type Implementation derives. See section 9.3 Derivation Rules or
2112 details.

2113 The `DerivedFrom` element has the following properties:

2114 ○ `relationshipTypeImplementationRef`: The QName specifies the Relationship
2115 Type Implementation from which this Relationship Type Implementation derives.

2116 • `RequiredContainerFeatures`: An implementation of a Relationship Type might depend
2117 on certain features of the environment it is executed in, such as specific (potentially proprietary)
2118 APIs of the TOSCA container.

2119 Thus, the contents of the `RequiredContainerFeatures` element provide “hints” to the
2120 TOSCA container allowing it to select the appropriate Relationship Type Implementation if
2121 multiple alternatives are provided.

2122 Each such dependency is defined by a separate `RequiredContainerFeature` element.

2123 The `RequiredContainerFeature` element has the following properties:

2124 ○ `feature`: The value of this attribute is a URI that denotes the corresponding needed
2125 feature of the environment.

2126 • `ImplementationArtifacts`: This element specifies a set of implementation artifacts for
2127 interfaces or operations of a Relationship Type.

2128 The `ImplementationArtifacts` element has the following properties:

2129 ○ `ImplementationArtifact`: This element specifies one implementation artifact of
2130 an interface or an operation.
2131

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

2137
2138 The `ImplementationArtifact` element has the following properties:

- 2139 ▪ `name`: This attribute specifies the name of the artifact, which SHOULD be unique
2140 within the scope of the encompassing Node Type Implementation.
- 2141 ▪ `interfaceName`: This OPTIONAL attribute specifies the name of the interface
2142 that is implemented by the actual implementation artifact. If not specified, the
2143 implementation artifact is assumed to provide the implementation for all
2144 interfaces defined by the Relationship Type referred to by the
2145 `relationshipType` attribute of the containing
2146 `RelationshipTypeImplementation`.

2147
2148 Note that the referenced interface can be defined in either the
2149 `SourceInterfaces` element or the `TargetInterfaces` element of the
2150 Relationship Type implemented by this Relationship Type Implementation.

- 2151 ▪ `operationName`: This OPTIONAL attribute specifies the name of the
2152 operation that is implemented by the actual implementation artifact. If specified,
2153 the `interfaceName` MUST be specified and the specified `operationName`
2154 MUST refer to an operation of the specified interface. If not specified, the
2155 implementation artifact is assumed to provide the implementation for all
2156 operations defined within the specified interface.
- 2157 ▪ `artifactType`: This attribute specifies the type of this artifact. The QName
2158 value of this attribute SHOULD correspond to the QName of an
2159 `ArtifactType` defined in the same Definitions document or in an imported
2160 document.

2161
2162 The `artifactType` attribute specifies the artifact type specific content of the
2163 `ImplementationArtifact` element body and indicates the type of Artifact
2164 Template referenced by the Implementation Artifact via the `artifactRef`
2165 attribute.

- 2166 ▪ `artifactRef`: This OPTIONAL attribute contains a QName that identifies an
2167 Artifact Template to be used as implementation artifact. This Artifact Template
2168 can be defined in the same Definitions document or in a separate, imported
2169 document.
2170 The type of Artifact Template referenced by the `artifactRef` attribute MUST
2171 be the same type or a sub-type of the type specified in the `artifactType`
2172 attribute.

2173
2174 Note: if no Artifact Template is referenced, the artifact type specific content of the
2175 `ImplementationArtifact` element alone is assumed to represent the
2176 actual artifact. For example, a simple script could be defined in place within the
2177 `ImplementationArtifact` element.

2178 9.3 Derivation Rules

2179 The following rules on combining definitions based on `DerivedFrom` apply:

- 2180 • Implementation Artifacts: The set of implementation artifacts of a Relationship Type
2181 Implementation consists of the set union of implementation artifacts defined by the Relationship

2182 Type Implementation itself and the implementation artifacts defined by any Relationship Type
 2183 Implementation the Relationship Type Implementation is derived from.
 2184 An implementation artifact defined by a Node Type Implementation overrides an implementation
 2185 artifact having the same interface name and operation name of a Relationship Type
 2186 Implementation the Relationship Type Implementation is derived from.
 2187 If an implementation artifact defined in a Relationship Type Implementation specifies only an
 2188 interface name, it substitutes implementation artifacts having the same interface name (with or
 2189 without an operation name defined) of any Relationship Type Implementation the Relationship
 2190 Type Implementation is derived from. In this case, the implementation of a complete interface of a
 2191 Relationship Type is overridden.
 2192 If an implementation artifact defined in a Relationship Type Implementation neither defines an
 2193 interface name nor an operation name, it overrides all implementation artifacts of any
 2194 Relationship Type Implementation the Relationship Type Implementation is derived from. In this
 2195 case, the complete implementation of a Relationship Type is overridden.

2196 9.4 Example

2197 The following example defines the Node Type Implementation “MyDBMSImplementation”. This is an
 2198 implementation of a Node Type “DBMS”.

```

2199 01 <Definitions id="MyImpls" name="My Implementations"
2200 02   targetNamespace="http://www.example.com/SampleImplementations"
2201 03   xmlns:bn="http://www.example.com/BaseRelationshipTypes"
2202 04   xmlns:ba="http://www.example.com/BaseArtifactTypes"
2203 05   xmlns:sa="http://www.example.com/SampleArtifacts">
2204 06
2205 07   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
2206 08     namespace="http://www.example.com/BaseArtifactTypes"/>
2207 09
2208 10   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
2209 11     namespace="http://www.example.com/BaseRelationshipTypes"/>
2210 12
2211 13   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
2212 14     namespace="http://www.example.com/SampleArtifacts"/>
2213 15
2214 16   <RelationshipTypeImplementation name="MyDBConnectImplementation"
2215 17     relationshipType="bn:DBConnection">
2216 18
2217 19     <ImplementationArtifacts>
2218 20       <ImplementationArtifact interfaceName="ConnectionInterface"
2219 21         operationName="connectTo"
2220 22         artifactType="ba:ScriptArtifact"
2221 23         artifactRef="sa:MyConnectScript">
2222 24       <ImplementationArtifact>
2223 25     </ImplementationArtifacts>
2224 26
2225 27   </RelationshipTypeImplementation>
2226 28
2227 29 </Definitions>
  
```

2228 The Relationship Type Implementation contains the “MyDBConnectionImpl” implementation artifact,
 2229 which is an artifact for the “ConnectionInterface” interface that has been defined for the “DBConnection”
 2230 base Relationship Type. The type of this artifact is a “ScriptArtifact” that has been defined as base Artifact
 2231 Type. The implementation artifact refers to the “MyConnectScript” Artifact Template that has been defined
 2232 before.

2233

10 Requirement Types

2234 This chapter specifies how *Requirement Types* are defined. A Requirement Type is a reusable entity that
2235 describes a kind of requirement that a Node Type can declare to expose. For example, a Requirement
2236 Type for a database connection can be defined and various Node Types (e.g. a Node Type for an
2237 application) can declare to expose (or “to have”) a requirement for a database connection.

2238 A Requirement Type defines the structure of observable properties via a *Properties Definition*, i.e. the
2239 names, data types and allowed values the properties defined in *Requirements* of Node Templates of a
2240 Node Type can have in cases where the Node Type defines a requirement of the respective Requirement
2241 Type.

2242 A Requirement Type can inherit properties and semantics from another Requirement Type by means of
2243 the *DerivedFrom* element. Requirement Types might be declared as abstract, meaning that they
2244 cannot be instantiated. The purpose of such abstract Requirement Types is to provide common
2245 properties for re-use in specialized, derived Requirement Types. Requirement Types might also be
2246 declared as final, meaning that they cannot be derived by other Requirement Types.

10.1 XML Syntax

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

```
2249 01 <RequirementType name="xs:NCName"  
2250 02     targetNamespace="xs:anyURI"?  
2251 03     abstract="yes|no"?  
2252 04     final="yes|no"?  
2253 05     requiredCapabilityType="xs:QName"?>  
2254 06  
2255 07   <Tags>  
2256 08     <Tag name="xs:string" value="xs:string"/> +  
2257 09   </Tags> ?  
2258 10  
2259 11   <DerivedFrom typeRef="xs:QName"/> ?  
2260 12  
2261 13   <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?  
2262 14  
2263 15 </RequirementType>
```

10.2 Properties

2264 The RequirementType element has the following properties:

- 2266 • **name**: This attribute specifies the name or identifier of the Requirement Type, which **MUST** be
2267 unique within the target namespace.
- 2268 • **targetNamespace**: This **OPTIONAL** attribute specifies the target namespace to which the
2269 definition of the Requirement Type will be added. If not specified, the Requirement Type definition
2270 will be added to the target namespace of the enclosing Definitions document.
- 2271 • **abstract**: This **OPTIONAL** attribute specifies that no instances can be created from Node
2272 Templates of a Node Type that defines a requirement of this Requirement Type.

2273
2274 As a consequence, a Node Type with a Requirement Definition of an abstract Requirement Type
2275 **MUST** be declared as abstract as well and a derived Node Type that defines a requirement of a
2276 type derived from the abstract Requirement Type has to be defined. For example, an abstract
2277 Node Type “Application” might be defined having a requirement of the abstract type “Container”.
2278 A derived Node Type “Web Application” can then be defined with a more concrete requirement of
2279 type “Web Application Container” which can then be used for defining Node Templates that can

- 2280 be instantiated during the creation of a service according to a Service Template.
2281
2282 Note: an abstract Requirement Type MUST NOT be declared as final.
- 2283 • `final`: This OPTIONAL attribute specifies that other Requirement Types MUST NOT be derived
2284 from this Requirement Type.
2285
2286 Note: a final Requirement Type MUST NOT be declared as abstract.
 - 2287 • `requiredCapabilityType`; This OPTIONAL attribute specifies the type of capability
2288 needed to match the defined Requirement Type. The QName value of this attribute refers to the
2289 QName of a `CapabilityType` element defined in the same Definitions document or in a
2290 separate, imported document.
2291
2292 Note: The following basic match-making for Requirements and Capabilities MUST be supported
2293 by each TOSCA implementation. Each Requirement is defined by a Requirement Definition,
2294 which in turn refers to a Requirement Type that specifies the needed Capability Type by means of
2295 its `requiredCapabilityType` attribute. The value of this attribute is used for basic type-
2296 based match-making: a Capability matches a Requirement if the Requirement's Requirement
2297 Type has a `requiredCapabilityType` value that corresponds to the Capability Type of the
2298 Capability or one of its super-types.
2299 Any domain-specific match-making semantics (e.g. based on constraints or properties) has to be
2300 defined in the cause of specifying the corresponding Requirement Types and Capability Types.
 - 2301 • `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by
2302 the author to describe the Requirement Type. Each tag is defined by a separate, nested `Tag`
2303 element.
2304 The `Tag` element has the following properties:
 - 2305 ○ `name`: This attribute specifies the name of the tag.
 - 2306 ○ `value`: This attribute specifies the value of the tag.
- 2307 Note: The name/value pairs defined in tags have no normative interpretation.
- 2309 • `DerivedFrom`: This is an OPTIONAL reference to another Requirement Type from which this
2310 Requirement Type derives. See section 10.3 Derivation Rules for details.
2311 The `DerivedFrom` element has the following properties:
 - 2312 ○ `typeRef`: The QName specifies the Requirement Type from which this Requirement
2313 Type derives its definitions and semantics.
 - 2314 • `PropertiesDefinition`: This element specifies the structure of the observable properties
2315 of the Requirement Type, such as its configuration and state, by means of XML schema.
2316 The `PropertiesDefinition` element has one but not both of the following properties:
 - 2317 ○ `element`: This attribute provides the QName of an XML element defining the structure
2318 of the Requirement Type Properties.
 - 2319 ○ `type`: This attribute provides the QName of an XML (complex) type defining the
2320 structure of the Requirement Type Properties.

2321 10.3 Derivation Rules

2322 The following rules on combining definitions based on `DerivedFrom` apply:

- 2323 • Requirement Type Properties: It is assumed that the XML element (or type) representing the
2324 Requirement Type Properties extends the XML element (or type) of the Requirement Type
2325 Properties of the Requirement Type referenced in the `DerivedFrom` element.

2326 10.4 Example

2327 The following example defines the Requirement Type “DatabaseClientEndpoint” that expresses the
2328 requirement of a client for a database connection. It is defined in a Definitions document
2329 “MyRequirements” within the target namespace “http://www.example.com/SampleRequirements”. Thus,
2330 by importing the corresponding namespace into another Definitions document, the
2331 “DatabaseClientEndpoint” Requirement Type is available for use in the other document.

```
2332 01 <Definitions id="MyRequirements" name="My Requirements"  
2333 02   targetNamespace="http://www.example.com/SampleRequirements"  
2334 03   xmlns:br="http://www.example.com/BaseRequirementTypes"  
2335 04   xmlns:mrp="http://www.example.com/SampleRequirementProperties">  
2336 05  
2337 06   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"  
2338 07     namespace="http://www.example.com/BaseRequirementTypes"/>  
2339 08  
2340 09   <Import importType="http://www.w3.org/2001/XMLSchema"  
2341 10     namespace="http://www.example.com/SampleRequirementProperties"/>  
2342 11  
2343 12   <RequirementType name="DatabaseClientEndpoint">  
2344 13     <DerivedFrom typeRef="br:ClientEndpoint"/>  
2345 14     <PropertiesDefinition  
2346 15       element="mrp:DatabaseClientEndpointProperties"/>  
2347 16   </RequirementType>  
2348 17  
2349 18 </Definitions>
```

2350 The Requirement Type “DatabaseClientEndpoint” defined in the example above is derived from another
2351 generic “ClientEndpoint” Requirement Type defined in a separate file by means of the `DerivedFrom`
2352 element. The definitions in that separate Definitions file are imported by means of the first `Import`
2353 element and the namespace of those imported definitions is assigned the prefix “br” in the current file.

2354 The “DatabaseClientEndpoint” Requirement Type defines a set of properties through an XML schema
2355 element definition “DatabaseClientEndpointProperties”. For example, those properties might include the
2356 definition of a port number to be used for client connections. The XML schema definition is stored in a
2357 separate XSD file that is imported by means of the second `Import` element. The namespace of the XML
2358 schema definitions is assigned the prefix “mrp” in the current file.

2359

11 Capability Types

2360 This chapter specifies how *Capability Types* are defined. A Capability Type is a reusable entity that
2361 describes a kind of capability that a Node Type can declare to expose. For example, a Capability Type for
2362 a database server endpoint can be defined and various Node Types (e.g. a Node Type for a database)
2363 can declare to expose (or to “provide”) the capability of serving as a database server endpoint.

2364 A Capability Type defines the structure of observable properties via a *Properties Definition*, i.e. the
2365 names, data types and allowed values the properties defined in *Capabilities* of Node Templates of a Node
2366 Type can have in cases where the Node Type defines a capability of the respective Capability Type.

2367 A Capability Type can inherit properties and semantics from another Capability Type by means of the
2368 *DerivedFrom* element. Capability Types might be declared as abstract, meaning that they cannot be
2369 instantiated. The purpose of such abstract Capability Types is to provide common properties for re-use in
2370 specialized, derived Capability Types. Capability Types might also be declared as final, meaning that they
2371 cannot be derived by other Capability Types.

11.1 XML Syntax

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

```
2374 01 <CapabilityType name="xs:NCName"  
2375 02           targetNamespace="xs:anyURI"?  
2376 03           abstract="yes|no"?  
2377 04           final="yes|no"?>  
2378 05  
2379 06   <Tags>  
2380 07     <Tag name="xs:string" value="xs:string"/> +  
2381 08   </Tags> ?  
2382 09  
2383 10   <DerivedFrom typeRef="xs:QName"/> ?  
2384 11  
2385 12   <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?  
2386 13  
2387 14 </CapabilityType>
```

11.2 Properties

2388 The *CapabilityType* element has the following properties:

- 2390
- 2391 • **name**: This attribute specifies the name or identifier of the Capability Type, which **MUST** be
unique within the target namespace.
 - 2392 • **targetNamespace**: This **OPTIONAL** attribute specifies the target namespace to which the
2393 definition of the Capability Type will be added. If not specified, the Capability Type definition will
2394 be added to the target namespace of the enclosing Definitions document.
 - 2395 • **abstract**: This **OPTIONAL** attribute specifies that no instances can be created from Node
2396 Templates of a Node Type that defines a capability of this Capability Type.

2397
2398 As a consequence, a Node Type with a Capability Definition of an abstract Capability Type **MUST**
2399 be declared as abstract as well and a derived Node Type that defines a capability of a type
2400 derived from the abstract Capability Type has to be defined. For example, an abstract Node Type
2401 “Server” might be defined having a capability of the abstract type “Container”. A derived Node
2402 Type “Web Server” can then be defined with a more concrete capability of type “Web Application
2403 Container” which can then be used for defining Node Templates that can be instantiated during
2404 the creation of a service according to a Service Template.

- 2405
- 2406 Note: an abstract Capability Type MUST NOT be declared as final.
- 2407
- 2408
- 2409
- 2410 Note: a final Capability Type MUST NOT be declared as abstract.
- 2411
- 2412
- 2413
- 2414
- **final**: This OPTIONAL attribute specifies that other Capability Types MUST NOT be derived from this Capability Type.
- 2415
- 2416
- 2417
- 2418 Note: The name/value pairs defined in tags have no normative interpretation.
- 2419
- 2420
- 2421
- **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.
- 2422
- 2423
- 2424
- 2425
- 2426
- 2427
- 2428
- 2429
- 2430
- 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.
- 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.

2431 11.3 Derivation Rules

2432 The following rules on combining definitions based on `DerivedFrom` apply:

- 2433
- 2434
- 2435
- **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.

2436 11.4 Example

2437 The following example defines the Capability Type “DatabaseServerEndpoint” that expresses the

2438 capability of a component to serve database connections. It is defined in a Definitions document

2439 “MyCapabilities” within the target namespace “http://www.example.com/SampleCapabilities”. Thus, by

2440 importing the corresponding namespace into another Definitions document, the

2441 “DatabaseServerEndpoint” Capability Type is available for use in the other document.

```

2442 01 <Definitions id="MyCapabilities" name="My Capabilities"
2443 02   targetNamespace="http://www.example.com/SampleCapabilities"
2444 03   xmlns:bc="http://www.example.com/BaseCapabilityTypes"
2445 04   xmlns:mcp="http://www.example.com/SampleCapabilityProperties">
2446 05
2447 06   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
2448 07     namespace="http://www.example.com/BaseCapabilityTypes"/>
2449 08
2450 09   <Import importType="http://www.w3.org/2001/XMLSchema"
2451 10     namespace="http://www.example.com/SampleCapabilityProperties"/>

```



```
2452 11
2453 12 <CapabilityType name="DatabaseServerEndpoint">
2454 13   <DerivedFrom typeRef="bc:ServerEndpoint"/>
2455 14   <PropertiesDefinition
2456 15     element="mcp:DatabaseServerEndpointProperties"/>
2457 16 </CapabilityType>
2458 17
2459 18 </Definitions>
```

2460 The Capability Type “DatabaseServerEndpoint” defined in the example above is derived from another
2461 generic “ServerEndpoint” Capability Type defined in a separate file by means of the `DerivedFrom`
2462 element. The definitions in that separate Definitions file are imported by means of the first `Import`
2463 element and the namespace of those imported definitions is assigned the prefix “bc” in the current file.

2464 The “DatabaseServerEndpoint” Capability Type defines a set of properties through an XML schema
2465 element definition “DatabaseServerEndpointProperties”. For example, those properties might include the
2466 definition of a port number where the server listens for client connections, or credentials to be used by
2467 clients. The XML schema definition is stored in a separate XSD file that is imported by means of the
2468 second `Import` element. The namespace of the XML schema definitions is assigned the prefix “mcp”
2469 in the current file.

2470

12 Artifact Types

2471 This chapter specifies how *Artifact Types* are defined. An Artifact Type is a reusable entity that defines
2472 the type of one or more Artifact Templates which in turn serve as deployment artifacts for Node
2473 Templates or implementation artifacts for Node Type and Relationship Type interface operations. For
2474 example, an Artifact Type “WAR File” might be defined for describing web application archive files. Based
2475 on this Artifact Type, one or more Artifact Templates representing concrete WAR files can be defined and
2476 referenced as deployment or implementation artifacts.

2477 An Artifact Type can define the structure of observable properties via a *Properties Definition*, i.e. the
2478 names, data types and allowed values the properties defined in Artifact Templates using an Artifact Type
2479 or instances of such Artifact Templates can have. Note that properties defined by an Artifact Type are
2480 assumed to be invariant across the contexts in which corresponding artifacts are used – as opposed to
2481 properties that can vary depending on the context. As an example of such an invariant property, an
2482 Artifact Type for a WAR file could define a “signature” property that can hold a hash for validating the
2483 actual artifact proper. In contrast, the path where the web application contained in the WAR file gets
2484 deployed can vary for each place where the WAR file is used.

2485 An Artifact Type can inherit definitions and semantics from another Artifact Type by means of the
2486 *DerivedFrom* element. Artifact Types can be declared as abstract, meaning that they cannot be
2487 instantiated. The purpose of such abstract Artifact Types is to provide common properties for re-use in
2488 specialized, derived Artifact Types. Artifact Types can also be declared as final, meaning that they cannot
2489 be derived by other Artifact Types.

2490

12.1 XML Syntax

2491 The following pseudo schema defines the XML syntax of Artifact Types:

```
2492 01 <ArtifactType name="xs:NCName"  
2493 02     targetNamespace="xs:anyURI"  
2494 03     abstract="yes|no"?  
2495 04     final="yes|no"?>  
2496 05  
2497 06   <Tags>  
2498 07     <Tag name="xs:string" value="xs:string"/> +  
2499 08   </Tags> ?  
2500 09  
2501 10   <DerivedFrom typeRef="xs:QName"/> ?  
2502 11  
2503 12   <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?  
2504 13  
2505 14 </ArtifactType>
```

2506

12.2 Properties

2507 The *ArtifactType* element has the following properties:

- 2508 • **name**: This attribute specifies the name or identifier of the Artifact Type, which **MUST** be unique
2509 within the target namespace.
- 2510 • **targetNamespace**: This **OPTIONAL** attribute specifies the target namespace to which the
2511 definition of the Artifact Type will be added. If not specified, the Artifact Type definition will be
2512 added to the target namespace of the enclosing Definitions document.
- 2513 • **abstract**: This **OPTIONAL** attribute specifies that no instances can be created from Artifact
2514 Templates of that abstract Artifact Type, i.e. the respective artifacts cannot be used directly as
2515 deployment or implementation artifact in any context.
2516

2517 As a consequence, an Artifact Template of an abstract Artifact Type MUST be replaced by an
2518 artifact of a derived Artifact Type at the latest during deployment of the element that uses the
2519 artifact (i.e. a Node Template or Relationship Template).

2520
2521 Note: an abstract Artifact Type MUST NOT be declared as final.

- 2522 • `final`: This OPTIONAL attribute specifies that other Artifact Types MUST NOT be derived from
2523 this Artifact Type.

2524
2525 Note: a final Artifact Type MUST NOT be declared as abstract.

- 2526 • `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by
2527 the author to describe the Artifact Type. Each tag is defined by a separate, nested `Tag` element.
2528 The `Tag` element has the following properties:

- 2529 ○ `name`: This attribute specifies the name of the tag.
- 2530 ○ `value`: This attribute specifies the value of the tag.

2531
2532 Note: The name/value pairs defined in tags have no normative interpretation.

- 2533 • `DerivedFrom`: This is an OPTIONAL reference to another Artifact Type from which this Artifact
2534 Type derives. See section 12.3 Derivation Rules for details.

2535 The `DerivedFrom` element has the following properties:

- 2536 ○ `typeRef`: The QName specifies the Artifact Type from which this Artifact Type derives
2537 its definitions and semantics.

- 2538 • `PropertiesDefinition`: This element specifies the structure of the observable properties
2539 of the Artifact Type, such as its configuration and state, by means of XML schema.

2540 The `PropertiesDefinition` element has one but not both of the following properties:

- 2541 ○ `element`: This attribute provides the QName of an XML element defining the structure
2542 of the Artifact Type Properties.
- 2543 ○ `type`: This attribute provides the QName of an XML (complex) type defining the
2544 structure of the Artifact Type Properties.

2545 12.3 Derivation Rules

2546 The following rules on combining definitions based on `DerivedFrom` apply:

- 2547 • `Artifact Type Properties`: It is assumed that the XML element (or type) representing the Artifact
2548 Type Properties extends the XML element (or type) of the Artifact Type Properties of the Artifact
2549 Type referenced in the `DerivedFrom` element.

2550 12.4 Example

2551 The following example defines the Artifact Type “RPM Package” that can be used for describing RPM
2552 packages as deployable artifacts on various Linux distributions. It is defined in a Definitions document
2553 “MyArtifacts” within the target namespace “<http://www.example.com/SampleArtifacts>”. Thus, by importing
2554 the corresponding namespace into another Definitions document, the “RPM Package” Artifact Type is
2555 available for use in the other document.

```
2556 01 <Definitions id="MyArtifacts" name="My Artifacts"  
2557 02   targetNamespace="http://www.example.com/SampleArtifacts"  
2558 03   xmlns:ba="http://www.example.com/BaseArtifactTypes"  
2559 04   xmlns:map="http://www.example.com/SampleArtifactProperties">  
2560 05  
2561 06   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"  
2562 07     namespace="http://www.example.com/BaseArtifactTypes"/>  
2563 08
```

```
2564 09 <Import importType="http://www.w3.org/2001/XMLSchema"
2565 10     namespace="http://www.example.com/SampleArtifactProperties"/>
2566 11
2567 12 <ArtifactType name="RPMPackage">
2568 13     <DerivedFrom typeRef="ba:OSPackage"/>
2569 14     <PropertiesDefinition element="map:RPMPackageProperties"/>
2570 15 </ArtifactType>
2571 16
2572 17 </Definitions>
```

2573 The Artifact Type “RPMPackage” defined in the example above is derived from another generic
2574 “OSPackage” Artifact Type defined in a separate file by means of the `DerivedFrom` element. The
2575 definitions in that separate Definitions file are imported by means of the first `Import` element and the
2576 namespace of those imported definitions is assigned the prefix “ba” in the current file.

2577 The “RPMPackage” Artifact Type defines a set of properties through an XML schema element definition
2578 “RPMPackageProperties”. For example, those properties might include the definition of the name or
2579 names of one or more RPM packages. The XML schema definition is stored in a separate XSD file that is
2580 imported by means of the second `Import` element. The namespace of the XML schema definitions is
2581 assigned the prefix “map” in the current file.

2582

13 Artifact Templates

2583 This chapter specifies how *Artifact Templates* are defined. An Artifact Template represents an artifact that
2584 can be referenced from other objects in a Service Template as a deployment artifact or implementation
2585 artifact. For example, from Node Types or Node Templates, an Artifact Template for some software
2586 installable could be referenced as a deployment artifact for materializing a specific software component.
2587 As another example, from within interface definitions of Node Types or Relationship Types, an Artifact
2588 Template for a WAR file could be referenced as implementation artifact for a REST operation.

2589 An Artifact Template refers to a specific Artifact Type that defines the structure of observable properties
2590 (metadata) or the artifact. The Artifact Template then typically defines values for those properties inside
2591 the `Properties` element. Note that properties defined by an Artifact Type are assumed to be invariant
2592 across the contexts in which corresponding artifacts are used – as opposed to properties that can vary
2593 depending on the context.

2594 Furthermore, an Artifact Template typically provides one or more references to the actual artifact itself
2595 that can be contained as a file in the CSAR (see section 3.7 and section 14) containing the overall
2596 Service Template or that can be available at a remote location such as an FTP server.

2597

13.1 XML Syntax

2598 The following pseudo schema defines the XML syntax of Artifact Templates:

```
2599 01 <ArtifactTemplate id="xs:ID" name="xs:string"? type="xs:QName">  
2600 02  
2601 03   <Properties>  
2602 04     XML fragment  
2603 05   </Properties> ?  
2604 06  
2605 07   <PropertyConstraints>  
2606 08     <PropertyConstraint property="xs:string"  
2607 09       constraintType="xs:anyURI"> +  
2608 10       constraint ?  
2609 11     </PropertyConstraint>  
2610 12   </PropertyConstraints> ?  
2611 13  
2612 14   <ArtifactReferences>  
2613 15     <ArtifactReference reference="xs:anyURI">  
2614 16       (  
2615 17         <Include pattern="xs:string"/>  
2616 18         |  
2617 19         <Exclude pattern="xs:string"/>  
2618 20       )*  
2619 21     </ArtifactReference> +  
2620 22   </ArtifactReferences> ?  
2621 23  
2622 24 </ArtifactTemplate>
```

2623

13.2 Properties

2624 The `ArtifactTemplate` element has the following properties:

- 2625 • `id`: This attribute specifies the identifier of the Artifact Template. The identifier of the Artifact
2626 Template MUST be unique within the target namespace.
- 2627 • `name`: This OPTIONAL attribute specifies the name of the Artifact Template.

- 2628
- 2629
- 2630
- 2631
- 2632
- 2633
- 2634
- 2635
- `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.
- 2636
- 2637
- 2638
- 2639
- `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.
- 2640
- 2641
- 2642
- 2643
- `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.
- 2644
- 2645
- 2646
- 2647
- 2648
- `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:

- 2677
- 2678
- 2679
- `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.

2680 13.3 Example

2681 The following example defines the Artifact Template "MyInstallable" that points to a zip file containing
2682 some software installable. It is defined in a Definitions document "MyArtifacts" within the target
2683 namespace "http://www.example.com/SampleArtifacts". The Artifact Template can be used in the same
2684 document, for example as a deployment artifact for some Node Template representing a software
2685 component, or it can be used in other Definitions documents by importing the corresponding namespace
2686 into another document.

```
2687 01 <Definitions id="MyArtifacts" name="My Artifacts"  
2688 02   targetNamespace="http://www.example.com/SampleArtifacts"  
2689 03   xmlns:ba="http://www.example.com/BaseArtifactTypes">  
2690 04  
2691 05   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"  
2692 06     namespace="http://www.example.com/BaseArtifactTypes"/>  
2693 07  
2694 08   <ArtifactTemplate id="MyInstallable"  
2695 09     name="My installable"  
2696 10     type="ba:ZipFile">  
2697 11     <ArtifactReferences>  
2698 12       <ArtifactReference reference="files/MyInstallable.zip"/>  
2699 13     </ArtifactReferences>  
2700 14   </ArtifactTemplate>  
2701 15  
2702 16 </Definitions>
```

2703 The Artifact Template "MyInstallable" defined in the example above is of type "ZipFile" that is specified in
2704 the `type` attribute of the `ArtifactTemplate` element. This Artifact Type is defined in a separate file,
2705 the definitions of which are imported by means of the `Import` element and the namespace of those
2706 imported definitions is assigned the prefix "ba" in the current file.

2707 The "MyInstallable" Artifact Template provides a reference to a file "MyInstallable.zip" by means of the
2708 `ArtifactReference` element. Since the URI provided in the `reference` attribute is a relative URI,
2709 it is interpreted relative to the root directory of the CSAR containing the Service Template.

2710 14 Policy Types

2711 This chapter specifies how *Policy Types* are defined. A Policy Type is a reusable entity that describes a
2712 kind of non-functional behavior or a kind of quality-of-service (QoS) that a Node Type can declare to
2713 expose. For example, a Policy Type can be defined to express high availability for specific Node Types
2714 (e.g. a Node Type for an application server).

2715 A Policy Type defines the structure of observable properties via a Properties Definition, i.e. the names,
2716 data types and allowed values the properties defined in a corresponding Policy Template can have.

2717 A Policy Type can inherit properties from another Policy Type by means of the `DerivedFrom` element.

2718 A Policy Type declares the set of Node Types it specifies non-functional behavior for via the `AppliesTo`
2719 element. Note that being “applicable to” does not enforce implementation: i.e. in case a Policy Type
2720 expressing high availability is associated with a “Webserver” Node Type, an instance of the Webserver is
2721 not necessarily highly available. Whether or not an instance of a Node Type to which a Policy Type is
2722 applicable will show the specified non-functional behavior, is determined by a Node Template of the
2723 corresponding Node Type.

2724 14.1 XML Syntax

2725 The following pseudo schema defines the XML syntax of Policy Types:

```
2726 01 <PolicyType name="xs:NCName"  
2727 02     policyLanguage="xs:anyURI"?  
2728 03     abstract="yes|no"?  
2729 04     final="yes|no"?  
2730 05     targetNamespace="xs:anyURI"?>  
2731 06   <Tags>  
2732 07     <Tag name="xs:string" value="xs:string"/> +  
2733 08   </Tags> ?  
2734 09  
2735 10   <DerivedFrom typeRef="xs:QName"/> ?  
2736 11  
2737 12   <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?  
2738 13  
2739 14   <AppliesTo>  
2740 15     <NodeTypeReference typeRef="xs:QName"/> +  
2741 16   </AppliesTo> ?  
2742 17  
2743 18   policy type specific content ?  
2744 19  
2745 20 </PolicyType>
```

2746 14.2 Properties

2747 The `PolicyType` element has the following properties:

- 2748 • `name`: This attribute specifies the name or identifier of the Policy Type, which **MUST** be unique
2749 within the target namespace.
- 2750 • `targetNamespace`: This **OPTIONAL** attribute specifies the target namespace to which the
2751 definition of the Policy Type will be added. If not specified, the Policy Type definition will be added
2752 to the target namespace of the enclosing Definitions document.
- 2753 • `policyLanguage`: This **OPTIONAL** attribute specifies the language used to specify the details
2754 of the Policy Type. These details can be defined as policy type specific content of the `PolicyType`
2755 element.

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

2759
2760 As a consequence, a Policy Template of an abstract Policy Type MUST be replaced by a policy
2761 of a derived Policy Type at the latest during deployment of the element that policy is attached to.

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

2764
2765 Note: a final Policy Type MUST NOT be declared as abstract.

2766 • **Tags**: This OPTIONAL element allows the definition of any number of tags which can be used by
2767 the author to describe the Policy Type. Each tag is defined by a separate, nested `Tag` element.
2768 The `Tag` element has the following properties:

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

2771
2772 Note: The name/value pairs defined in tags have no normative interpretation.

2773 • **DerivedFrom**: This is an OPTIONAL reference to another Policy Type from which this Policy
2774 Type derives. See section 14.3 Derivation Rules for details.
2775 The `DerivedFrom` element has the following properties:

- 2776 ○ **typeRef**: The QName specifies the Policy Type from which this Policy Type derives its
2777 definitions from.

2778 • **PropertiesDefinition**: This element specifies the structure of the observable properties
2779 of the Policy Type by means of XML schema.
2780 The `PropertiesDefinition` element has one but not both of the following properties:

- 2781 ○ **element**: This attribute provides the QName of an XML element defining the structure
2782 of the Policy Type Properties.
- 2783 ○ **type**: This attribute provides the QName of an XML (complex) type defining the
2784 structure of the Policy Type Properties.

2785 • **AppliesTo**: This OPTIONAL element specifies the set of Node Types the Policy Type is
2786 applicable to, each defined as a separate, nested `NodeTypeReference` element.
2787 The `NodeTypeReference` element has the following property:

- 2788 ○ **typeRef**: The attribute provides the QName of a Node Type to which the Policy Type
2789 applies.

2790 14.3 Derivation Rules

2791 The following rules on combining definitions based on `DerivedFrom` apply:

2792 • **Properties Definitions**: It is assumed that the XML element (or type) representing the Policy Type
2793 Properties Definitions extends the XML element (or type) of the Policy Type Properties Definitions
2794 of the Policy Type referenced in the `DerivedFrom` element.

2795 • **Applies To**: The set of Node Types the Policy Type is applicable to consist of the set union of
2796 Node Types derived from and Node Types explicitly referenced by the Policy Type by means of
2797 its `AppliesTo` element.

2798 • **Policy Language**: A Policy Type MUST define the same policy language as the Policy Type it
2799 derives from. In case the Policy Type used as basis for derivation has no `policyLanguage`
2800 attribute defined, the deriving Policy Type can define any appropriate policy language.

2801 14.4 Example

2802 The following example defines two Policy Types, the “HighAvailability” Policy Type and the
2803 “ContinuousAvailability” Policy Type. They are defined in a Definitions document “MyPolicyTypes” within
2804 the target namespace “http://www.example.com/SamplePolicyTypes”. Thus, by importing the
2805 corresponding namespace into another Definitions document, both Policy Types are available for use in
2806 the other document.

```
2807 01 <Definitions id="MyPolicyTypes" name="My Policy Types"  
2808 02   targetNamespace="http://www.example.com/SamplePolicyTypes"  
2809 03   xmlns:bnt="http://www.example.com/BaseNodeTypes">  
2810 04   xmlns:spp="http://www.example.com/SamplePolicyProperties">  
2811 05  
2812 06   <Import importType="http://www.w3.org/2001/XMLSchema"  
2813 07     namespace="http://www.example.com/SamplePolicyProperties"/>  
2814 08  
2815 09   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"  
2816 10     namespace="http://www.example.com/BaseNodeTypes"/>  
2817 11  
2818 12  
2819 13   <PolicyType name="HighAvailability">  
2820 14     <PropertiesDefinition element="spp:HAProperties"/>  
2821 15   </PolicyType>  
2822 16  
2823 17   <PolicyType name="ContinuousAvailability">  
2824 18     <DerivedFrom typeRef="HighAvailability"/>  
2825 19     <PropertiesDefinition element="spp:CAProperties"/>  
2826 20     <AppliesTo>  
2827 21       <NodeTypeReference typeRef="bnt:DBMS"/>  
2828 22     </AppliesTo>  
2829 23   </PolicyType>  
2830 24  
2831 25 </Definitions>
```

2832 The Policy Type “HighAvailability” defined in the example above has the “HAProperties” properties that
2833 are defined in a separate namespace as an XML element. The same namespace contains the
2834 “CAProperties” element that defines the properties of the “ContinuousAvailability” Policy Type. This
2835 namespace is imported by means of the first `Import` element and the namespace of those imported
2836 definitions is assigned the prefix “spp” in the current file.

2837 The “ContinuousAvailability” Policy Type is derived from the “HighAvailability” Policy Type. Furthermore, it
2838 is applicable to the “DBMS” Node Type. This Node Type is defined in a separate namespace, which is
2839 imported by means of the second `Import` element and the namespace of those imported definitions is
2840 assigned the prefix “bnt” in the current file.

2841

15 Policy Templates

2842 This chapter specifies how *Policy Templates* are defined. A Policy Template represents a particular non-
2843 functional behavior or quality-of-service that can be referenced by a Node Template. A Policy Template
2844 refers to a specific Policy Type that defines the structure of observable properties (metadata) of the non-
2845 functional behavior. The Policy Template then typically defines values for those properties inside the
2846 *Properties* element. Note that properties defined by a Policy Template are assumed to be invariant
2847 across the contexts in which corresponding behavior is exposed – as opposed to properties defined in
2848 Policies of Node Templates that may vary depending on the context.

2849

15.1 XML Syntax

2850 The following pseudo schema defines the XML syntax of Policy Templates:

```
2851 01 <PolicyTemplate id="xs:ID" name="xs:string"? type="xs:QName">  
2852 02  
2853 03   <Properties>  
2854 04     XML fragment  
2855 05   </Properties> ?  
2856 06  
2857 07   <PropertyConstraints>  
2858 08     <PropertyConstraint property="xs:string"  
2859 09       constraintType="xs:anyURI"> +  
2860 10       constraint ?  
2861 11     </PropertyConstraint>  
2862 12   </PropertyConstraints> ?  
2863 13  
2864 14   policy type specific content ?  
2865 15  
2866 16 </PolicyTemplate>
```

2867

15.2 Properties

2868 The `PolicyTemplate` element has the following properties:

- 2869 • `id`: This attribute specifies the identifier of the Policy Template which **MUST** be unique within the
2870 target namespace.
- 2871 • `name`: This **OPTIONAL** attribute specifies the name of the Policy Template.
- 2872 • `type`: The QName value of this attribute refers to the Policy Type providing the type of the Policy
2873 Template.
- 2874 • `Properties`: This **OPTIONAL** element specifies the invariant properties of the Policy
2875 Template, i.e. those properties that will be commonly used across different contexts in which the
2876 Policy Template is used.

2877

2878 The initial values are specified by providing an instance document of the XML schema of the
2879 corresponding Policy Type Properties. This instance document considers the inheritance
2880 structure deduced by the `DerivedFrom` property of the Policy Type referenced by the `type`
2881 attribute of the Policy Template.

- 2882 • `PropertyConstraints`: This **OPTIONAL** element specifies constraints on the use of one or
2883 more of the Policy Type Properties of the Policy Type providing the property definitions for the
2884 Policy Template. Each constraint is specified by means of a separate nested
2885 `PropertyConstraint` element.

2886 The `PropertyConstraint` element has the following properties:

- 2887 o `property`: The string value of this property is an XPath expression pointing to the
2888 property within the Policy Type Properties document that is constrained within the context
2889 of the Policy Template. More than one constraint MUST NOT be defined for each
2890 property.
- 2891 o `constraintType`: The constraint type is specified by means of a URI, which defines
2892 both the semantic meaning of the constraint as well as the format of the content.

2893 15.3 Example

2894 The following example defines a Policy Template “MyHAPolicy”. It is defined in a Definitions document
2895 “MyPolicies” within the target namespace “http://www.example.com/SamplePolicies”. The Policy
2896 Template can be used in the same Definitions document, for example, as a Policy of some Node
2897 Template, or it can be used in other document by importing the corresponding namespace into the other
2898 document.

```
2899 01 <Definitions id="MyPolicies" name="My Policies"  
2900 02   targetNamespace="http://www.example.com/SamplePolicies"  
2901 03   xmlns:spt="http://www.example.com/SamplePolicyTypes">  
2902 04  
2903 05   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"  
2904 06     namespace="http://www.example.com/SamplePolicyTypes"/>  
2905 07  
2906 08   <PolicyTemplate id="MyHAPolicy"  
2907 09     name="My High Availability Policy"  
2908 10     type="bpt:HighAvailability">  
2909 11     <Properties>  
2910 12       <HAProperties>  
2911 13         <AvailabilityClass>4</AvailabilityClass>  
2912 14         <HeartbeatFrequency measuredIn="msec">  
2913 15           250  
2914 16         </HeartbeatFrequency>  
2915 17       </HAProperties>  
2916 18     </Properties>  
2917 19   </PolicyTemplate>  
2918 20  
2919 21 </Definitions>
```

2920 The Policy Template “MyHAPolicy” defined in the example above is of type “HighAvailability” that is
2921 specified in the `type` attribute of the `PolicyTemplate` element. This Policy Type is defined in a
2922 separate file, the definitions of which are imported by means of the `Import` element and the namespace
2923 of those imported definitions is assigned the prefix “spt” in the current file.

2924 The “MyHAPolicy” Policy Template provides values for the properties defined by the Properties Definition
2925 of the “HighAvailability” Policy Type. The `AvailabilityClass` property is set to “4”. The value of the
2926 `HeartbeatFrequency` is “250”, measured in “msec”.

2927

2928 16 Cloud Service Archive (CSAR)

2929 This section defines the metadata of a cloud service archive as well as its overall structure.

2930 16.1 Overall Structure of a CSAR

2931 A CSAR is a zip file containing at least two directories, the *TOSCA-Metadata* directory and the *Definitions*
2932 directory. Beyond that, other directories MAY be contained in a CSAR, i.e. the creator of a CSAR has all
2933 freedom to define the content of a CSAR and the structuring of this content as appropriate for the cloud
2934 application.

2935 The TOSCA-Metadata directory contains metadata describing the other content of the CSAR. This
2936 metadata is referred to as *TOSCA meta file*. This file is named `TOSCA` and has the file extension `.meta`.

2937 The Definitions directory contains one or more TOSCA Definitions documents (file extension `.tosca`).
2938 These Definitions files typically contain definitions related to the cloud application of the CSAR. In
2939 addition, CSARs can contain just the definition of elements for re-use in other contexts. For example, a
2940 CSAR might be used to package a set of Node Types and Relationship Types with their respective
2941 implementations that can then be used by Service Templates provided in other CSARs. In cases where a
2942 complete cloud application is packaged in a CSAR, one of the Definitions documents in the Definitions
2943 directory MUST contain a Service Template definition that defines the structure and behavior of the cloud
2944 application.

2945 16.2 TOSCA Meta File

2946 The TOSCA meta file includes metadata that allows interpreting the various artifacts within the CSAR
2947 properly. The `TOSCA.meta` file is contained in the `TOSCA-Metadata` directory of the CSAR.

2948 A TOSCA meta file consists of name/value pairs. The name-part of a name/value pair is followed by a
2949 colon, followed by a blank, followed by the value-part of the name/value pair. The name MUST NOT
2950 contain a colon. Values that represent binary data MUST be base64 encoded. Values that extend beyond
2951 one line can be spread over multiple lines if each subsequent line starts with at least one space. Such
2952 spaces are then collapsed when the value string is read.

```
2953 01 <name>: <value>
```

2954 Each name/value pair is in a separate line. A list of related name/value pairs, i.e. a list of consecutive
2955 name/value pairs describing a particular file in a CSAR, is called a *block*. Blocks are separated by an
2956 empty line. The first block, called *block_0*, is metadata about the CSAR itself. All other blocks represent
2957 metadata of files in the CSAR.

2958 The structure of `block_0` in the TOSCA meta file is as follows:

```
2959 01 TOSCA-Meta-File-Version: digit.digit  
2960 02 CSAR-Version: digit.digit  
2961 03 Created-By: string  
2962 04 Entry-Definitions: string ?
```

2963 The name/value pairs are as follows:

- 2964 • `TOSCA-Meta-File-Version`: This is the version number of the TOSCA meta file format.
2965 The value MUST be “1.0” in the current version of the TOSCA specification.
- 2966 • `CSAR-Version`: This is the version number of the CSAR specification. The value MUST be
2967 “1.0” in the current version of the TOSCA specification.
- 2968 • `Created-By`: The person or vendor, respectively, who created the CSAR.

- `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.
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.

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.

```

2983 01 Name: <path-name_1>
2984 02 Content-Type: type_1/subtype_1
2985 03 <name_11>: <value_11>
2986 04 <name_12>: <value_12>
2987 05 ...
2988 06 <name_1n>: <value_1n>
2989 07
2990 08 ...
2991 09
2992 10 Name: <path-name_k>
2993 11 Content-Type: type_k/subtype_k
2994 12 <name_k1>: <value_k1>
2995 13 <name_k2>: <value_k2>
2996 14 ...
2997 15 <name_km>: <value_km>

```

The name/value pairs are as follows:

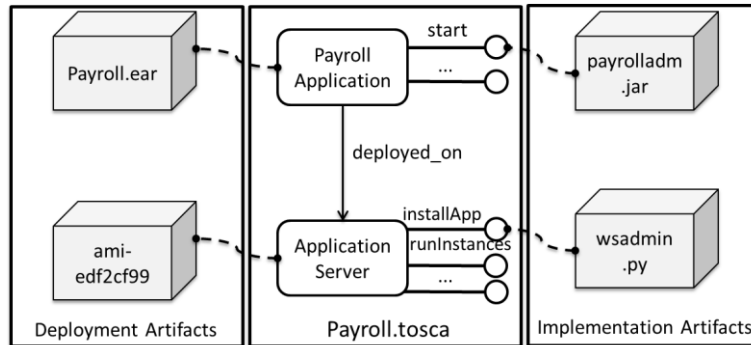
- `Name`: The pathname or pathname pattern of the file(s) or resources described within the actual CSAR.
Note, that the file located at this location MAY basically contain a reference to an external file. Such a reference is given by a URI that is of one of the URL schemes “file”, “http”, or “https”.
- `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.”.

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.

16.3 Example

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

3018 provided too; for example, the start operation of the Payroll Application is implemented by a
 3019 Java API supported by the payrolladm.jar file, the installApp operation of the Application
 3020 Server is realized by the Python script wsadmin.py, while the runInstances operation is a REST
 3021 API available at Amazon for running instances of an AMI. Note, that the runInstances operation is
 3022 not related to a particular implementation artifact because it is available as an Amazon Web Service
 3023 (https://ec2.amazonaws.com/?Action=RunInstances); but the details of this REST API are specified with
 3024 the operation of the Application Server Node Type.



3025 |
 3026 | Figure 7: Sample Service Template

3027 The corresponding Node Types and Relationship Types have been defined in the
 3028 PayrollTypes.tosca document, which is imported by the Definitions document containing the
 3029 Payroll Service Template. The following listing provides some of the details:

```

3030 01 <Definitions id="PayrollDefinitions"
3031 02     targetNamespace="http://www.example.com/stetosca"
3032 03     xmlns:pay="http://www.example.com/stetosca/Types">
3033 04
3034 05     <Import namespace="http://www.example.com/stetosca/Types"
3035 06
3036 07     location="http://www.example.com/stetosca/Types/PayrollTypes.tosca"
3037 08     importType=" http://docs.oasis-open.org/tosca/ns/2011/12"/>
3038 09
3039 10     <Types>
3040 11     ...
3041 12     </Types>
3042 13
3043 14     <ServiceTemplate id="Payroll" name="Payroll Service Template">
3044 15
3045 16     <TopologyTemplate ID="PayrollTemplate">
3046 17
3047 18     <NodeTemplate id="Payroll Application"
3048 19     type="pay:ApplicationNodeType">
3049 20     ...
3050 21
3051 22     <DeploymentArtifacts>
3052 23     <DeploymentArtifact name="PayrollEAR"
3053 24     type="http://www.example.com/
3054 25     ns/tosca/2011/12/
3055 26     DeploymentArtifactTypes/CSARref">
3056 27     EARs/Payroll.ear
3057 28     </DeploymentArtifact>
3058 29     </DeploymentArtifacts>
3059 30
3060 31     </NodeTemplate>
3061 32
3062 33     <NodeTemplate id="Application Server"
  
```

```

3063 33         type="pay:ApplicationServerNodeType">
3064 34     ...
3065 35
3066 36     <DeploymentArtifacts>
3067 37         <DeploymentArtifact name="ApplicationServerImage"
3068 38             type="http://www.example.com/
3069 39                 ns/tosca/2011/12/
3070 40                 DeploymentArtifactTypes/AMIref">
3071 41             ami-edf2cf99
3072 42         </DeploymentArtifact>
3073 43     </DeploymentArtifacts>
3074 44
3075 45 </NodeTemplate>
3076 46
3077 47 <RelationshipTemplate id="deployed_on"
3078 48     type="pay:deployed_on">
3079 49     <SourceElement ref="Payroll Application"/>
3080 50     <TargetElement ref="Application Server"/>
3081 51 </RelationshipTemplate>
3082 52
3083 53 </TopologyTemplate>
3084 54
3085 55 </ServiceTemplate>
3086 56
3087 57 </Definitions>

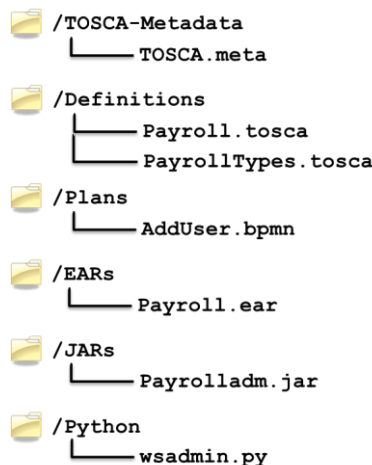
```

3088

3089 The Payroll Application Node Template specifies the deployment artifact PayrollEAR. It is a
3090 reference to the CSAR containing the Payroll. [stetosca](#) file, which is indicated by the
3091 `.../CSARref` type of the `DeploymentArtifact` element. The type specific content is a path
3092 expression in the directory structure of the CSAR: it points to the `Payroll.ear` file in the `EARS`
3093 directory of the CSAR (see Figure 8 for the structure of the corresponding CSAR).

3094 The Application Server Node Template has a `DeploymentArtifact` called
3095 `ApplicationServerImage` that is a reference to an AMI (Amazon Machine Image), indicated by an
3096 `.../AMIref` type.

3097 The corresponding CSAR has the following structure (see Figure 8): The `TOSCA.meta` file is contained
3098 in the `TOSCA-Metadata` directory. The Payroll. [stetosca](#) file itself is contained in the
3099 `Service-Template` directory. Also, the `PayrollTypes. stetosca` file is in this directory. The
3100 content of the other directories has been sketched before.



3101

3102 | Figure 8: Structure of CSAR Sample

3103 The TOSCA.meta file is as follows:

```
3104 01 TOSCA-Meta-Version: 1.0
3105 02 CSAR-Version: 1.0
3106 03 Created-By: Frank
3107 04
3108 05 Name: Service-Template/Payroll.tosca
3109 06 Content-Type: application/vnd.oasis.tosca.definitions
3110 07
3111 | 08 Name: Service-Template/PayrollTypes.stetosca
3112 09 Content-Type: application/vnd.oasis.tosca.definitions
3113 10
3114 11 Name: Plans/AddUser.bpmn
3115 12 Content-Type: application/vnd.oasis.bpmn
3116 13
3117 14 Name: EARs/Payroll.ear
3118 15 Content-Type: application/vnd.oasis.ear
3119 16
3120 17 Name: JARs/Payrolladm.jar
3121 18 Content-Type: application/vnd.oasis.jar
3122 19
3123 20 Name: Python/wsadmin.py
3124 21 Content-Type: application/vnd.oasis.py
3125
```

3126 **17 Security Considerations**

3127 TOSCA does not mandate the use of any specific mechanism or technology for client authentication.
3128 However, a client **MUST** provide a principal or the principal **MUST** be obtainable by the infrastructure.

3129 **18 Conformance**

- 3130 A TOSCA Definitions document conforms to this specification if it conforms to the TOSCA schema and
3131 follows the syntax and semantics defined in the normative portions of this specification. The TOSCA
3132 schema takes precedence over the TOSCA grammar (pseudo schema as defined in section 2.5), which
3133 in turn takes precedence over normative text, which in turn takes precedence over examples.
- 3134 An implementation conforms to this specification if it can process a conformant TOSCA Definitions
3135 document according to the rules described in chapters 4 through 16 of this specification.
- 3136 This specification allows extensions. Each implementation SHALL fully support all required functionality of
3137 the specification exactly as specified. The use of extensions SHALL NOT contradict nor cause the non-
3138 conformance of functionality defined in the specification.

3139 **Appendix A. Portability and Interoperability**
3140 **Considerations**

3141 This section illustrates the portability and interoperability aspects addressed by Service Templates:

3142 Portability - The ability to take Service Templates created in one vendor's environment and use them in
3143 another vendor's environment.

3144 Interoperability - The capability for multiple components (e.g. a task of a plan and the definition of a
3145 topology node) to interact using well-defined messages and protocols. This enables combining
3146 components from different vendors allowing seamless management of services.

3147 Portability demands support of TOSCA elements.

3148

Appendix B. Acknowledgements

3149 The following individuals have participated in the creation of this specification and are gratefully
3150 acknowledged.

3151 **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.

3152

Appendix C. Complete TOSCA Grammar

3154 **Note:** The following is a pseudo EBNF grammar notation meant for documentation purposes only. The
3155 grammar is not intended for machine processing.

```

3156 01 <Definitions id="xs:ID"
3157 02     name="xs:string"?
3158 03     targetNamespace="xs:anyURI">
3159 04
3160 05     <Extensions>
3161 06         <Extension namespace="xs:anyURI"
3162 07             mustUnderstand="yes|no"?/> +
3163 08     </Extensions> ?
3164 09
3165 10     <Import namespace="xs:anyURI"?
3166 11         location="xs:anyURI"?
3167 12         importType="xs:anyURI"/> *
3168 13
3169 14     <Types>
3170 15         <xs:schema .../> *
3171 16     </Types> ?
3172 17
3173 18     (
3174 19         <ServiceTemplate id="xs:ID"
3175 20             name="xs:string"?
3176 21             targetNamespace="xs:anyURI"
3177 22             substitutableNodeType="xs:QName"?>
3178 23
3179 24         <Tags>
3180 25             <Tag name="xs:string" value="xs:string"/> +
3181 26         </Tags> ?
3182 27
3183 28         <BoundaryDefinitions>
3184 29             <Properties>
3185 30                 XML fragment
3186 31                 <PropertyMappings>
3187 32                     <PropertyMapping serviceTemplatePropertyRef="xs:string"
3188 33                         targetObjectRef="xs:IDREF"
3189 34                         targetPropertyRef="xs:IDREF"/> +
3190 35                 </PropertyMappings/> ?
3191 36             </Properties> ?
3192 37
3193 38             <PropertyConstraints>
3194 39                 <PropertyConstraint property="xs:string"
3195 40                     constraintType="xs:anyURI"> +
3196 41                     constraint ?
3197 42                 </PropertyConstraint>
3198 43             </PropertyConstraints> ?
3199 44
3200 45             <Requirements>
3201 46                 <Requirement name="xs:string" ref="xs:IDREF"/> +
3202 47             </Requirements> ?
3203 48
3204 49             <Capabilities>
3205 50                 <Capability name="xs:string" ref="xs:IDREF"/> +
3206 51             </Capabilities> ?

```

```

3207 52
3208 53     <Policies>
3209 54         <Policy name="xs:string"? policyType="xs:QName"
3210 55             policyRef="xs:QName"?>
3211 56             policy specific content ?
3212 57         </Policy> +
3213 58     </Policies> ?
3214 59
3215 60     <Interfaces>
3216 61         <Interface name="xs:NCName">
3217 62             <Operation name="xs:NCName">
3218 63                 (
3219 64                     <NodeOperation nodeRef="xs:IDREF"
3220 65                         interfaceName="xs:anyURI"
3221 66                         operationName="xs:NCName"/>
3222 67                 |
3223 68                     <RelationshipOperation relationshipRef="xs:IDREF"
3224 69                         interfaceName="xs:anyURI"
3225 70                         operationName="xs:NCName"/>
3226 71                 |
3227 72                     <Plan planRef="xs:IDREF"/>
3228 73                 )
3229 74             </Operation> +
3230 75         </Interface> +
3231 76     </Interfaces> ?
3232 77
3233 78 </BoundaryDefinitions> ?
3234 79
3235 80 <TopologyTemplate>
3236 81     (
3237 82         <NodeTemplate id="xs:ID" name="xs:string"? type="xs:QName"
3238 83             minInstances="xs:integer"?
3239 84             maxInstances="xs:integer | xs:string"?>
3240 85         <Properties>
3241 86             XML fragment
3242 87         </Properties> ?
3243 88
3244 89         <PropertyConstraints>
3245 90             <PropertyConstraint property="xs:string"
3246 91                 constraintType="xs:anyURI">
3247 92                 constraint ?
3248 93             </PropertyConstraint> +
3249 94         </PropertyConstraints> ?
3250 95
3251 96         <Requirements>
3252 97             <Requirement id="xs:ID" name="xs:string" type="xs:QName"> +
3253 98                 <Properties>
3254 99                     XML fragment
3255 100                 <Properties> ?
3256 101                 <PropertyConstraints>
3257 102                     <PropertyConstraint property="xs:string"
3258 103                         constraintType="xs:anyURI"> +
3259 104                         constraint ?
3260 105                     </PropertyConstraint>
3261 106                 </PropertyConstraints> ?
3262 107             </Requirement>
3263 108         </Requirements> ?
3264 109

```



```

3265 110      <Capabilities>
3266 111          <Capability id="xs:ID" name="xs:string"
3267 112              type="xs:QName"> +
3268 113              <Properties>
3269 114                  XML fragment
3270 115              <Properties> ?
3271 116              <PropertyConstraints>
3272 117                  <PropertyConstraint property="xs:string"
3273 118                      constraintType="xs:anyURI">
3274 119                      constraint ?
3275 120                  </PropertyConstraint> +
3276 121              </PropertyConstraints> ?
3277 122          </Capability>
3278 123      </Capabilities> ?
3279 124
3280 125      <Policies>
3281 126          <Policy name="xs:string"? policyType="xs:QName"
3282 127              policyRef="xs:QName"?>
3283 128              policy specific content ?
3284 129          </Policy> +
3285 130      </Policies> ?
3286 131
3287 132      <DeploymentArtifacts>
3288 133          <DeploymentArtifact name="xs:string"
3289 134              artifactType="xs:QName"
3290 135              artifactRef="xs:QName"?>
3291 136              artifact specific content ?
3292 137          </DeploymentArtifact> +
3293 138      </DeploymentArtifacts> ?
3294 139  </NodeTemplate>
3295 140  |
3296 141  <RelationshipTemplate id="xs:ID" name="xs:string"?
3297 142              type="xs:QName">
3298 143      <Properties>
3299 144          XML fragment
3300 145      </Properties> ?
3301 146
3302 147      <PropertyConstraints>
3303 148          <PropertyConstraint property="xs:string"
3304 149              constraintType="xs:anyURI">
3305 150              constraint ?
3306 151          </PropertyConstraint> +
3307 152      </PropertyConstraints> ?
3308 153
3309 154      <SourceElement ref="xs:IDREF"/>
3310 155      <TargetElement ref="xs:IDREF"/>
3311 156
3312 157      <RelationshipConstraints>
3313 158          <RelationshipConstraint constraintType="xs:anyURI">
3314 159              constraint ?
3315 160          </RelationshipConstraint> +
3316 161      </RelationshipConstraints> ?
3317 162
3318 163      </RelationshipTemplate>
3319 164  ) +
3320 165  </TopologyTemplate>
3321 166
3322 167  <Plans>

```

```

3323 168     <Plan id="xs:ID"
3324 169         name="xs:string"?
3325 170         planType="xs:anyURI"
3326 171         planLanguage="xs:anyURI">
3327 172
3328 173         <Precondition expressionLanguage="xs:anyURI">
3329 174             condition
3330 175         </Precondition> ?
3331 176
3332 177         <InputParameters>
3333 178             <InputParameter name="xs:string" type="xs:string"
3334 179                 required="yes|no"?/> +
3335 180         </InputParameters> ?
3336 181
3337 182         <OutputParameters>
3338 183             <OutputParameter name="xs:string" type="xs:string"
3339 184                 required="yes|no"?/> +
3340 185         </OutputParameters> ?
3341 186
3342 187         (
3343 188             <PlanModel>
3344 189                 actual plan
3345 190             </PlanModel>
3346 191             |
3347 192             <PlanModelReference reference="xs:anyURI"/>
3348 193         )
3349 194
3350 195     </Plan> +
3351 196 </Plans> ?
3352 197
3353 198 </ServiceTemplate>
3354 199 |
3355 200 <NodeType name="xs:NCName" targetNamespace="xs:anyURI"?
3356 201     abstract="yes|no"? final="yes|no"?>
3357 202
3358 203     <DerivedFrom typeRef="xs:QName"/> ?
3359 204
3360 205     <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
3361 206
3362 207     <RequirementDefinitions>
3363 208         <RequirementDefinition name="xs:string"
3364 209             requirementType="xs:QName"
3365 210             lowerBound="xs:integer"?
3366 211             upperBound="xs:integer | xs:string"?>
3367 212             <Constraints>
3368 213                 <Constraint constraintType="xs:anyURI">
3369 214                     constraint type specific content
3370 215                 </Constraint> +
3371 216             </Constraints> ?
3372 217         </RequirementDefinition> +
3373 218     </RequirementDefinitions> ?
3374 219
3375 220     <CapabilityDefinitions>
3376 221         <CapabilityDefinition name="xs:string"
3377 222             capabilityType="xs:QName"
3378 223             lowerBound="xs:integer"?
3379 224             upperBound="xs:integer | xs:string"?>
3380 225         <Constraints>

```

```

3381 226         <Constraint constraintType="xs:anyURI">
3382 227             constraint type specific content
3383 228         </Constraint> +
3384 229     </Constraints> ?
3385 230     </CapabilityDefinition> +
3386 231 </CapabilityDefinitions>
3387 232
3388 233 <InstanceStates>
3389 234     <InstanceState state="xs:anyURI"> +
3390 235 </InstanceStates> ?
3391 236
3392 237 <Interfaces>
3393 238     <Interface name="xs:NCName | xs:anyURI">
3394 239         <Operation name="xs:NCName">
3395 240             <InputParameters>
3396 241                 <InputParameter name="xs:string" type="xs:string"
3397 242                     required="yes|no"?/> +
3398 243             </InputParameters> ?
3399 244             <OutputParameters>
3400 245                 <OutputParameter name="xs:string" type="xs:string"
3401 246                     required="yes|no"?/> +
3402 247             </OutputParameters> ?
3403 248         </Operation> +
3404 249     </Interface> +
3405 250 </Interfaces> ?
3406 251
3407 252 </NodeType>
3408 253 |
3409 254 <NodeTypeImplementation name="xs:NCName"
3410 255     targetNamespace="xs:anyURI"?
3411 256     nodeType="xs:QName"
3412 257     abstract="yes|no"?
3413 258     final="yes|no"?>
3414 259
3415 260     <DerivedFrom nodeTypeImplementationRef="xs:QName"/> ?
3416 261
3417 262     <RequiredContainerFeatures>
3418 263         <RequiredContainerFeature feature="xs:anyURI"/> +
3419 264     </RequiredContainerFeatures> ?
3420 265
3421 266     <ImplementationArtifacts>
3422 267         <ImplementationArtifact interfaceName="xs:NCName | xs:anyURI"?
3423 268             operationName="xs:NCName"?
3424 269             artifactType="xs:QName"
3425 270             artifactRef="xs:QName"?>
3426 271             artifact specific content ?
3427 272         </ImplementationArtifact> +
3428 273     </ImplementationArtifacts> ?
3429 274
3430 275     <DeploymentArtifacts>
3431 276         <DeploymentArtifact name="xs:string" artifactType="xs:QName"
3432 277             artifactRef="xs:QName"?>
3433 278             artifact specific content ?
3434 279         </DeploymentArtifact> +
3435 280     </DeploymentArtifacts> ?
3436 281
3437 282 </NodeTypeImplementation>
3438 283 |

```

```

3439 284     <RelationshipType name="xs:NCName"
3440 285         targetNamespace="xs:anyURI"?
3441 286         abstract="yes|no"?
3442 287         final="yes|no"?> +
3443 288
3444 289     <DerivedFrom typeRef="xs:QName"/> ?
3445 290
3446 291     <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
3447 292
3448 293     <InstanceStates>
3449 294         <InstanceState state="xs:anyURI"> +
3450 295     </InstanceStates> ?
3451 296
3452 297     <SourceInterfaces>
3453 298         <Interface name="xs:NCName | xs:anyURI">
3454 299             <Operation name="xs:NCName">
3455 300                 <InputParameters>
3456 301                     <InputParameter name="xs:string" type="xs:string"
3457 302                         required="yes|no"?/> +
3458 303                 </InputParameters> ?
3459 304                 <OutputParameters>
3460 305                     <OutputParameter name="xs:string" type="xs:string"
3461 306                         required="yes|no"?/> +
3462 307                 </OutputParameters> ?
3463 308             </Operation> +
3464 309         </Interface> +
3465 310     </SourceInterfaces> ?
3466 311
3467 312     <TargetInterfaces>
3468 313         <Interface name="xs:NCName | xs:anyURI">
3469 314             <Operation name="xs:NCName">
3470 315                 <InputParameters>
3471 316                     <InputParameter name="xs:string" type="xs:string"
3472 317                         required="yes|no"?/> +
3473 318                 </InputParameters> ?
3474 319                 <OutputParameters>
3475 320                     <OutputParameter name="xs:string" type="xs:string"
3476 321                         required="yes|no"?/> +
3477 322                 </OutputParameters> ?
3478 323             </Operation> +
3479 324         </Interface> +
3480 325     </TargetInterfaces> ?
3481 326
3482 327     <ValidSource typeRef="xs:QName"/> ?
3483 328
3484 329     <ValidTarget typeRef="xs:QName"/> ?
3485 330
3486 331 </RelationshipType>
3487 332 |
3488 333 <RelationshipTypeImplementation name="xs:NCName"
3489 334     targetNamespace="xs:anyURI"?
3490 335     relationshipType="xs:QName"
3491 336     abstract="yes|no"?
3492 337     final="yes|no"?>
3493 338
3494 339     <DerivedFrom relationshipTypeImplementationRef="xs:QName"/> ?
3495 340
3496 341     <RequiredContainerFeatures>

```

```

3497 342      <RequiredContainerFeature feature="xs:anyURI"/> +
3498 343    </RequiredContainerFeatures> ?
3499 344
3500 345    <ImplementationArtifacts>
3501 346      <ImplementationArtifact interfaceName="xs:NCName | xs:anyURI"?
3502 347        operationName="xs:NCName"?
3503 348        artifactType="xs:QName"
3504 349        artifactRef="xs:QName"?>
3505 350        artifact specific content ?
3506 351      <ImplementationArtifact> +
3507 352    </ImplementationArtifacts> ?
3508 353
3509 354  </RelationshipTypeImplementation>
3510 355  |
3511 356    <RequirementType name="xs:NCName"
3512 357      targetNamespace="xs:anyURI"?
3513 358      abstract="yes|no"?
3514 359      final="yes|no"?
3515 360      requiredCapabilityType="xs:QName"?>
3516 361
3517 362    <DerivedFrom typeRef="xs:QName"/> ?
3518 363
3519 364    <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
3520 365
3521 366  </RequirementType>
3522 367  |
3523 368    <CapabilityType name="xs:NCName"
3524 369      targetNamespace="xs:anyURI"?
3525 370      abstract="yes|no"?
3526 371      final="yes|no"?>
3527 372
3528 373    <DerivedFrom typeRef="xs:QName"/> ?
3529 374
3530 375    <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
3531 376
3532 377  </CapabilityType>
3533 378  |
3534 379    <ArtifactType name="xs:NCName"
3535 380      targetNamespace="xs:anyURI"?
3536 381      abstract="yes|no"?
3537 382      final="yes|no"?>
3538 383
3539 384    <DerivedFrom typeRef="xs:QName"/> ?
3540 385
3541 386    <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
3542 387
3543 388  </ArtifactType>
3544 389  |
3545 390    <ArtifactTemplate id="xs:ID" name="xs:string"? type="xs:QName">
3546 391
3547 392      <Properties>
3548 393        XML fragment
3549 394      </Properties> ?
3550 395
3551 396      <PropertyConstraints>
3552 397        <PropertyConstraint property="xs:string"
3553 398          constraintType="xs:anyURI"> +
3554 399          constraint ?

```

```

3555 400         </PropertyConstraint>
3556 401     </PropertyConstraints> ?
3557 402
3558 403     <ArtifactReferences>
3559 404         <ArtifactReference reference="xs:anyURI">
3560 405             (
3561 406                 <Include pattern="xs:string"/>
3562 407                 |
3563 408                 <Exclude pattern="xs:string"/>
3564 409             ) *
3565 410         </ArtifactReference> +
3566 411     </ArtifactReferences> ?
3567 412
3568 413 </ArtifactTemplate>
3569 414 |
3570 415 <PolicyType name="xs:NCName"
3571 416             policyLanguage="xs:anyURI"?
3572 417             abstract="yes|no"?
3573 418             final="yes|no"?
3574 419             targetNamespace="xs:anyURI"?>
3575 420     <Tags>
3576 421         <Tag name="xs:string" value="xs:string"/> +
3577 422     </Tags> ?
3578 423
3579 424     <DerivedFrom typeRef="xs:QName"/> ?
3580 425
3581 426     <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
3582 427
3583 428     <AppliesTo>
3584 429         <NodeTypeReference typeRef="xs:QName"/> +
3585 430     </AppliesTo> ?
3586 431
3587 432     policy type specific content ?
3588 433
3589 434 </PolicyType>
3590 435 |
3591 436 <PolicyTemplate id="xs:ID" name="xs:string"? type="xs:QName">
3592 437
3593 438     <Properties>
3594 439         XML fragment
3595 440     </Properties> ?
3596 441
3597 442     <PropertyConstraints>
3598 443         <PropertyConstraint property="xs:string"
3599 444                             constraintType="xs:anyURI"> +
3600 445             constraint ?
3601 446         </PropertyConstraint>
3602 447     </PropertyConstraints> ?
3603 448
3604 449     policy type specific content ?
3605 450
3606 451 </PolicyTemplate>
3607 452 ) +
3608 453
3609 454 </Definitions>

```

3610

Appendix D. TOSCA Schema

3611 | [TOSCA-v1.0.xsd](#)~~TOSCA-v1.0.xsd~~:

```
3612 01 <?xml version="1.0" encoding="UTF-8"?>
3613 02 <xs:schema targetNamespace="http://docs.oasis-open.org/tosca/ns/2011/12"
3614 03   elementFormDefault="qualified" attributeFormDefault="unqualified"
3615 04   xmlns="http://docs.oasis-open.org/tosca/ns/2011/12"
3616 05   xmlns:xs="http://www.w3.org/2001/XMLSchema">
3617 06
3618 07   <xs:import namespace="http://www.w3.org/XML/1998/namespace"
3619 08     schemaLocation="http://www.w3.org/2001/xml.xsd"/>
3620 09
3621 10   <xs:element name="documentation" type="tDocumentation"/>
3622 11   <xs:complexType name="tDocumentation" mixed="true">
3623 12     <xs:sequence>
3624 13       <xs:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
3625 14     </xs:sequence>
3626 15     <xs:attribute name="source" type="xs:anyURI"/>
3627 16     <xs:attribute ref="xml:lang"/>
3628 17   </xs:complexType>
3629 18
3630 19   <xs:complexType name="tExtensibleElements">
3631 20     <xs:sequence>
3632 21       <xs:element ref="documentation" minOccurs="0" maxOccurs="unbounded"/>
3633 22       <xs:any namespace="##other" processContents="lax" minOccurs="0"
3634 23         maxOccurs="unbounded"/>
3635 24     </xs:sequence>
3636 25     <xs:anyAttribute namespace="##other" processContents="lax"/>
3637 26   </xs:complexType>
3638 27
3639 28   <xs:complexType name="tImport">
3640 29     <xs:complexContent>
3641 30       <xs:extension base="tExtensibleElements">
3642 31         <xs:attribute name="namespace" type="xs:anyURI"/>
3643 32         <xs:attribute name="location" type="xs:anyURI"/>
3644 33         <xs:attribute name="importType" type="importedURI" use="required"/>
3645 34       </xs:extension>
3646 35     </xs:complexContent>
3647 36   </xs:complexType>
3648 37
3649 38   <xs:element name="Definitions">
3650 39     <xs:complexType>
3651 40       <xs:complexContent>
3652 41         <xs:extension base="tDefinitions"/>
3653 42       </xs:complexContent>
3654 43     </xs:complexType>
3655 44   </xs:element>
3656 45   <xs:complexType name="tDefinitions">
3657 46     <xs:complexContent>
3658 47       <xs:extension base="tExtensibleElements">
3659 48         <xs:sequence>
3660 49           <xs:element name="Extensions" minOccurs="0">
3661 50             <xs:complexType>
3662 51               <xs:sequence>
3663 52                 <xs:element name="Extension" type="tExtension"
```

```

3664 53         maxOccurs="unbounded"/>
3665 54     </xs:sequence>
3666 55 </xs:complexType>
3667 56 </xs:element>
3668 57 <xs:element name="Import" type="tImport" minOccurs="0"
3669 58     maxOccurs="unbounded"/>
3670 59 <xs:element name="Types" minOccurs="0">
3671 60     <xs:complexType>
3672 61         <xs:sequence>
3673 62             <xs:any namespace="##other" processContents="lax" minOccurs="0"
3674 63                 maxOccurs="unbounded"/>
3675 64         </xs:sequence>
3676 65     </xs:complexType>
3677 66 </xs:element>
3678 67 <xs:choice maxOccurs="unbounded">
3679 68     <xs:element name="ServiceTemplate" type="tServiceTemplate"/>
3680 69     <xs:element name="NodeType" type="tNodeType"/>
3681 70     <xs:element name="NodeTypeImplementation"
3682 71         type="tNodeTypeImplementation"/>
3683 72     <xs:element name="RelationshipType" type="tRelationshipType"/>
3684 73     <xs:element name="RelationshipTypeImplementation"
3685 74         type="tRelationshipTypeImplementation"/>
3686 75     <xs:element name="RequirementType" type="tRequirementType"/>
3687 76     <xs:element name="CapabilityType" type="tCapabilityType"/>
3688 77     <xs:element name="ArtifactType" type="tArtifactType"/>
3689 78     <xs:element name="ArtifactTemplate" type="tArtifactTemplate"/>
3690 79     <xs:element name="PolicyType" type="tPolicyType"/>
3691 80     <xs:element name="PolicyTemplate" type="tPolicyTemplate"/>
3692 81 </xs:choice>
3693 82 </xs:sequence>
3694 83 <xs:attribute name="id" type="xs:ID" use="required"/>
3695 84 <xs:attribute name="name" type="xs:string" use="optional"/>
3696 85 <xs:attribute name="targetNamespace" type="xs:anyURI" use="required"/>
3697 86 </xs:extension>
3698 87 </xs:complexContent>
3699 88 </xs:complexType>
3700 89
3701 90 <xs:complexType name="tServiceTemplate">
3702 91     <xs:complexContent>
3703 92         <xs:extension base="tExtensibleElements">
3704 93             <xs:sequence>
3705 94                 <xs:element name="Tags" type="tTags" minOccurs="0"/>
3706 95                 <xs:element name="BoundaryDefinitions" type="tBoundaryDefinitions"
3707 96                     minOccurs="0"/>
3708 97                 <xs:element name="TopologyTemplate" type="tTopologyTemplate"/>
3709 98                 <xs:element name="Plans" type="tPlans" minOccurs="0"/>
3710 99             </xs:sequence>
3711 100             <xs:attribute name="id" type="xs:ID" use="required"/>
3712 101             <xs:attribute name="name" type="xs:string" use="optional"/>
3713 102             <xs:attribute name="targetNamespace" type="xs:anyURI"/>
3714 103             <xs:attribute name="substitutableNodeType" type="xs:QName"
3715 104                 use="optional"/>
3716 105             </xs:extension>
3717 106         </xs:complexContent>
3718 107     </xs:complexType>
3719 108
3720 109 <xs:complexType name="tTags">
3721 110     <xs:sequence>

```



```

3722 111     <xs:element name="Tag" type="tTag" maxOccurs="unbounded"/>
3723 112     </xs:sequence>
3724 113 </xs:complexType>
3725 114
3726 115 <xs:complexType name="tTag">
3727 116     <xs:attribute name="name" type="xs:string" use="required"/>
3728 117     <xs:attribute name="value" type="xs:string" use="required"/>
3729 118 </xs:complexType>
3730 119
3731 120 <xs:complexType name="tBoundaryDefinitions">
3732 121     <xs:sequence>
3733 122         <xs:element name="Properties" minOccurs="0">
3734 123             <xs:complexType>
3735 124                 <xs:sequence>
3736 125                     <xs:any namespace="##other"/>
3737 126                     <xs:element name="PropertyMappings" minOccurs="0">
3738 127                         <xs:complexType>
3739 128                             <xs:sequence>
3740 129                                 <xs:element name="PropertyMapping" type="tPropertyMapping"
3741 130                                     maxOccurs="unbounded"/>
3742 131                             </xs:sequence>
3743 132                         </xs:complexType>
3744 133                     </xs:element>
3745 134                 </xs:sequence>
3746 135             </xs:complexType>
3747 136         </xs:element>
3748 137     <xs:element name="PropertyConstraints" minOccurs="0">
3749 138         <xs:complexType>
3750 139             <xs:sequence>
3751 140                 <xs:element name="PropertyConstraint" type="tPropertyConstraint"
3752 141                     maxOccurs="unbounded"/>
3753 142             </xs:sequence>
3754 143         </xs:complexType>
3755 144     </xs:element>
3756 145 <xs:element name="Requirements" minOccurs="0">
3757 146     <xs:complexType>
3758 147         <xs:sequence>
3759 148             <xs:element name="Requirement" type="tRequirementRef"
3760 149                 maxOccurs="unbounded"/>
3761 150         </xs:sequence>
3762 151     </xs:complexType>
3763 152 </xs:element>
3764 153 <xs:element name="Capabilities" minOccurs="0">
3765 154     <xs:complexType>
3766 155         <xs:sequence>
3767 156             <xs:element name="Capability" type="tCapabilityRef"
3768 157                 maxOccurs="unbounded"/>
3769 158         </xs:sequence>
3770 159     </xs:complexType>
3771 160 </xs:element>
3772 161 <xs:element name="Policies" minOccurs="0">
3773 162     <xs:complexType>
3774 163         <xs:sequence>
3775 164             <xs:element name="Policy" type="tPolicy" maxOccurs="unbounded"/>
3776 165         </xs:sequence>
3777 166     </xs:complexType>
3778 167 </xs:element>
3779 168 <xs:element name="Interfaces" minOccurs="0">

```

```

3780 169     <xs:complexType>
3781 170     <xs:sequence>
3782 171         <xs:element name="Interface" type="tExportedInterface"
3783 172             maxOccurs="unbounded"/>
3784 173     </xs:sequence>
3785 174     </xs:complexType>
3786 175     </xs:element>
3787 176 </xs:sequence>
3788 177 </xs:complexType>
3789 178
3790 179 <xs:complexType name="tPropertyMapping">
3791 180     <xs:attribute name="serviceTemplatePropertyRef" type="xs:string"
3792 181         use="required"/>
3793 182     <xs:attribute name="targetObjectRef" type="xs:IDREF" use="required"/>
3794 183     <xs:attribute name="targetPropertyRef" type="xs:string"
3795 184         use="required"/>
3796 185 </xs:complexType>
3797 186
3798 187 <xs:complexType name="tRequirementRef">
3799 188     <xs:attribute name="name" type="xs:string" use="optional"/>
3800 189     <xs:attribute name="ref" type="xs:IDREF" use="required"/>
3801 190 </xs:complexType>
3802 191
3803 192 <xs:complexType name="tCapabilityRef">
3804 193     <xs:attribute name="name" type="xs:string" use="optional"/>
3805 194     <xs:attribute name="ref" type="xs:IDREF" use="required"/>
3806 195 </xs:complexType>
3807 196
3808 197 <xs:complexType name="tEntityType" abstract="true">
3809 198     <xs:complexContent>
3810 199         <xs:extension base="tExtensibleElements">
3811 200             <xs:sequence>
3812 201                 <xs:element name="Tags" type="tTags" minOccurs="0"/>
3813 202                 <xs:element name="DerivedFrom" minOccurs="0">
3814 203                     <xs:complexType>
3815 204                         <xs:attribute name="typeRef" type="xs:QName" use="required"/>
3816 205                     </xs:complexType>
3817 206                 </xs:element>
3818 207                 <xs:element name="PropertiesDefinition" minOccurs="0">
3819 208                     <xs:complexType>
3820 209                         <xs:attribute name="element" type="xs:QName"/>
3821 210                         <xs:attribute name="type" type="xs:QName"/>
3822 211                     </xs:complexType>
3823 212                 </xs:element>
3824 213             </xs:sequence>
3825 214             <xs:attribute name="name" type="xs:NCName" use="required"/>
3826 215             <xs:attribute name="abstract" type="tBoolean" default="no"/>
3827 216             <xs:attribute name="final" type="tBoolean" default="no"/>
3828 217             <xs:attribute name="targetNamespace" type="xs:anyURI"
3829 218                 use="optional"/>
3830 219         </xs:extension>
3831 220     </xs:complexContent>
3832 221 </xs:complexType>
3833 222
3834 223 <xs:complexType name="tEntityTypeTemplate" abstract="true">
3835 224     <xs:complexContent>
3836 225         <xs:extension base="tExtensibleElements">
3837 226             <xs:sequence>

```

```

3838 227     <xs:element name="Properties" minOccurs="0">
3839 228         <xs:complexType>
3840 229             <xs:sequence>
3841 230                 <xs:any namespace="##other" processContents="lax"/>
3842 231             </xs:sequence>
3843 232         </xs:complexType>
3844 233     </xs:element>
3845 234     <xs:element name="PropertyConstraints" minOccurs="0">
3846 235         <xs:complexType>
3847 236             <xs:sequence>
3848 237                 <xs:element name="PropertyConstraint"
3849 238                     type="tPropertyConstraint" maxOccurs="unbounded"/>
3850 239             </xs:sequence>
3851 240         </xs:complexType>
3852 241     </xs:element>
3853 242 </xs:sequence>
3854 243     <xs:attribute name="id" type="xs:ID" use="required"/>
3855 244     <xs:attribute name="type" type="xs:QName" use="required"/>
3856 245 </xs:extension>
3857 246 </xs:complexContent>
3858 247 </xs:complexType>
3859 248
3860 249 <xs:complexType name="tNodeTemplate">
3861 250     <xs:complexContent>
3862 251         <xs:extension base="tEntityTemplate">
3863 252             <xs:sequence>
3864 253                 <xs:element name="Requirements" minOccurs="0">
3865 254                     <xs:complexType>
3866 255                         <xs:sequence>
3867 256                             <xs:element name="Requirement" type="tRequirement"
3868 257                                 maxOccurs="unbounded"/>
3869 258                         </xs:sequence>
3870 259                     </xs:complexType>
3871 260                 </xs:element>
3872 261                 <xs:element name="Capabilities" minOccurs="0">
3873 262                     <xs:complexType>
3874 263                         <xs:sequence>
3875 264                             <xs:element name="Capability" type="tCapability"
3876 265                                 maxOccurs="unbounded"/>
3877 266                         </xs:sequence>
3878 267                     </xs:complexType>
3879 268                 </xs:element>
3880 269                 <xs:element name="Policies" minOccurs="0">
3881 270                     <xs:complexType>
3882 271                         <xs:sequence>
3883 272                             <xs:element name="Policy" type="tPolicy"
3884 273                                 maxOccurs="unbounded"/>
3885 274                         </xs:sequence>
3886 275                     </xs:complexType>
3887 276                 </xs:element>
3888 277                 <xs:element name="DeploymentArtifacts" type="tDeploymentArtifacts"
3889 278                     minOccurs="0"/>
3890 279             </xs:sequence>
3891 280             <xs:attribute name="name" type="xs:string" use="optional"/>
3892 281             <xs:attribute name="minInstances" type="xs:int" use="optional"
3893 282                 default="1"/>
3894 283             <xs:attribute name="maxInstances" use="optional" default="1">
3895 284                 <xs:simpleType>

```

```

3896 285     <xs:union>
3897 286     <xs:simpleType>
3898 287         <xs:restriction base="xs:nonNegativeInteger">
3899 288             <xs:pattern value="([1-9]+[0-9]*)"/>
3900 289         </xs:restriction>
3901 290     </xs:simpleType>
3902 291     <xs:simpleType>
3903 292         <xs:restriction base="xs:string">
3904 293             <xs:enumeration value="unbounded"/>
3905 294         </xs:restriction>
3906 295     </xs:simpleType>
3907 296 </xs:union>
3908 297 </xs:simpleType>
3909 298 </xs:attribute>
3910 299 </xs:extension>
3911 300 </xs:complexContent>
3912 301 </xs:complexType>
3913 302
3914 303 <xs:complexType name="tTopologyTemplate">
3915 304     <xs:complexContent>
3916 305         <xs:extension base="tExtensibleElements">
3917 306             <xs:choice maxOccurs="unbounded">
3918 307                 <xs:element name="NodeTemplate" type="tNodeTemplate"/>
3919 308                 <xs:element name="RelationshipTemplate"
3920 309                     type="tRelationshipTemplate"/>
3921 310             </xs:choice>
3922 311         </xs:extension>
3923 312     </xs:complexContent>
3924 313 </xs:complexType>
3925 314
3926 315 <xs:complexType name="tRelationshipType">
3927 316     <xs:complexContent>
3928 317         <xs:extension base="tEntityType">
3929 318             <xs:sequence>
3930 319                 <xs:element name="InstanceStates"
3931 320                     type="tTopologyElementInstanceStates" minOccurs="0"/>
3932 321                 <xs:element name="SourceInterfaces" minOccurs="0">
3933 322                     <xs:complexType>
3934 323                         <xs:sequence>
3935 324                             <xs:element name="Interface" type="tInterface"
3936 325                                 maxOccurs="unbounded"/>
3937 326                         </xs:sequence>
3938 327                     </xs:complexType>
3939 328                 </xs:element>
3940 329                 <xs:element name="TargetInterfaces" minOccurs="0">
3941 330                     <xs:complexType>
3942 331                         <xs:sequence>
3943 332                             <xs:element name="Interface" type="tInterface"
3944 333                                 maxOccurs="unbounded"/>
3945 334                         </xs:sequence>
3946 335                     </xs:complexType>
3947 336                 </xs:element>
3948 337                 <xs:element name="ValidSource" minOccurs="0">
3949 338                     <xs:complexType>
3950 339                         <xs:attribute name="typeRef" type="xs:QName" use="required"/>
3951 340                     </xs:complexType>
3952 341                 </xs:element>
3953 342                 <xs:element name="ValidTarget" minOccurs="0">

```

```

3954 343     <xs:complexType>
3955 344     <xs:attribute name="typeRef" type="xs:QName" use="required"/>
3956 345     </xs:complexType>
3957 346     </xs:element>
3958 347     </xs:sequence>
3959 348     </xs:extension>
3960 349     </xs:complexContent>
3961 350 </xs:complexType>
3962 351
3963 352 <xs:complexType name="tRelationshipTypeImplementation">
3964 353   <xs:complexContent>
3965 354     <xs:extension base="tExtensibleElements">
3966 355       <xs:sequence>
3967 356         <xs:element name="Tags" type="tTags" minOccurs="0"/>
3968 357         <xs:element name="DerivedFrom" minOccurs="0">
3969 358           <xs:complexType>
3970 359             <xs:attribute name="relationshipTypeImplementationRef"
3971 360               type="xs:QName" use="required"/>
3972 361           </xs:complexType>
3973 362         </xs:element>
3974 363         <xs:element name="RequiredContainerFeatures"
3975 364           type="tRequiredContainerFeatures" minOccurs="0"/>
3976 365         <xs:element name="ImplementationArtifacts"
3977 366           type="tImplementationArtifacts" minOccurs="0"/>
3978 367       </xs:sequence>
3979 368       <xs:attribute name="name" type="xs:NCName" use="required"/>
3980 369       <xs:attribute name="targetNamespace" type="xs:anyURI"
3981 370         use="optional"/>
3982 371       <xs:attribute name="relationshipType" type="xs:QName"
3983 372         use="required"/>
3984 373       <xs:attribute name="abstract" type="tBoolean" use="optional"
3985 374         default="no"/>
3986 375       <xs:attribute name="final" type="tBoolean" use="optional"
3987 376         default="no"/>
3988 377     </xs:extension>
3989 378   </xs:complexContent>
3990 379 </xs:complexType>
3991 380
3992 381 <xs:complexType name="tRelationshipTemplate">
3993 382   <xs:complexContent>
3994 383     <xs:extension base="tEntityTemplate">
3995 384       <xs:sequence>
3996 385         <xs:element name="SourceElement">
3997 386           <xs:complexType>
3998 387             <xs:attribute name="ref" type="xs:IDREF" use="required"/>
3999 388           </xs:complexType>
4000 389         </xs:element>
4001 390         <xs:element name="TargetElement">
4002 391           <xs:complexType>
4003 392             <xs:attribute name="ref" type="xs:IDREF" use="required"/>
4004 393           </xs:complexType>
4005 394         </xs:element>
4006 395         <xs:element name="RelationshipConstraints" minOccurs="0">
4007 396           <xs:complexType>
4008 397             <xs:sequence>
4009 398               <xs:element name="RelationshipConstraint"
4010 399                 maxOccurs="unbounded">
4011 400                 <xs:complexType>

```

```

4012 401         <xs:sequence>
4013 402         <xs:any namespace="##other" processContents="lax"
4014 403             minOccurs="0"/>
4015 404         </xs:sequence>
4016 405         <xs:attribute name="constraintType" type="xs:anyURI"
4017 406             use="required"/>
4018 407         </xs:complexType>
4019 408     </xs:element>
4020 409 </xs:sequence>
4021 410 </xs:complexType>
4022 411 </xs:element>
4023 412 </xs:sequence>
4024 413 <xs:attribute name="name" type="xs:string" use="optional"/>
4025 414 </xs:extension>
4026 415 </xs:complexContent>
4027 416 </xs:complexType>
4028 417
4029 418 <xs:complexType name="tNodeType">
4030 419 <xs:complexContent>
4031 420 <xs:extension base="tEntityType">
4032 421 <xs:sequence>
4033 422 <xs:element name="RequirementDefinitions" minOccurs="0">
4034 423 <xs:complexType>
4035 424 <xs:sequence>
4036 425 <xs:element name="RequirementDefinition"
4037 426     type="tRequirementDefinition" maxOccurs="unbounded"/>
4038 427 </xs:sequence>
4039 428 </xs:complexType>
4040 429 </xs:element>
4041 430 <xs:element name="CapabilityDefinitions" minOccurs="0">
4042 431 <xs:complexType>
4043 432 <xs:sequence>
4044 433 <xs:element name="CapabilityDefinition"
4045 434     type="tCapabilityDefinition" maxOccurs="unbounded"/>
4046 435 </xs:sequence>
4047 436 </xs:complexType>
4048 437 </xs:element>
4049 438 <xs:element name="InstanceStates"
4050 439     type="tTopologyElementInstanceStates" minOccurs="0"/>
4051 440 <xs:element name="Interfaces" minOccurs="0">
4052 441 <xs:complexType>
4053 442 <xs:sequence>
4054 443 <xs:element name="Interface" type="tInterface"
4055 444     maxOccurs="unbounded"/>
4056 445 </xs:sequence>
4057 446 </xs:complexType>
4058 447 </xs:element>
4059 448 </xs:sequence>
4060 449 </xs:extension>
4061 450 </xs:complexContent>
4062 451 </xs:complexType>
4063 452
4064 453 <xs:complexType name="tNodeTypeImplementation">
4065 454 <xs:complexContent>
4066 455 <xs:extension base="tExtensibleElements">
4067 456 <xs:sequence>
4068 457 <xs:element name="Tags" type="tTags" minOccurs="0"/>
4069 458 <xs:element name="DerivedFrom" minOccurs="0">

```

```

4070 459     <xs:complexType>
4071 460     <xs:attribute name="nodeTypeImplementationRef" type="xs:QName"
4072 461         use="required"/>
4073 462     </xs:complexType>
4074 463 </xs:element>
4075 464 <xs:element name="RequiredContainerFeatures"
4076 465     type="tRequiredContainerFeatures" minOccurs="0"/>
4077 466 <xs:element name="ImplementationArtifacts"
4078 467     type="tImplementationArtifacts" minOccurs="0"/>
4079 468 <xs:element name="DeploymentArtifacts" type="tDeploymentArtifacts"
4080 469     minOccurs="0"/>
4081 470 </xs:sequence>
4082 471 <xs:attribute name="name" type="xs:NCName" use="required"/>
4083 472 <xs:attribute name="targetNamespace" type="xs:anyURI"
4084 473     use="optional"/>
4085 474 <xs:attribute name="nodeType" type="xs:QName" use="required"/>
4086 475 <xs:attribute name="abstract" type="tBoolean" use="optional"
4087 476     default="no"/>
4088 477 <xs:attribute name="final" type="tBoolean" use="optional"
4089 478     default="no"/>
4090 479 </xs:extension>
4091 480 </xs:complexContent>
4092 481 </xs:complexType>
4093 482
4094 483 <xs:complexType name="tRequirementType">
4095 484 <xs:complexContent>
4096 485 <xs:extension base="tEntityType">
4097 486 <xs:attribute name="requiredCapabilityType" type="xs:QName"
4098 487     use="optional"/>
4099 488 </xs:extension>
4100 489 </xs:complexContent>
4101 490 </xs:complexType>
4102 491
4103 492 <xs:complexType name="tRequirementDefinition">
4104 493 <xs:complexContent>
4105 494 <xs:extension base="tExtensibleElements">
4106 495 <xs:sequence>
4107 496 <xs:element name="Constraints" minOccurs="0">
4108 497 <xs:complexType>
4109 498 <xs:sequence>
4110 499 <xs:element name="Constraint" type="tConstraint"
4111 500     maxOccurs="unbounded"/>
4112 501 </xs:sequence>
4113 502 </xs:complexType>
4114 503 </xs:element>
4115 504 </xs:sequence>
4116 505 <xs:attribute name="name" type="xs:string" use="required"/>
4117 506 <xs:attribute name="requirementType" type="xs:QName"
4118 507     use="required"/>
4119 508 <xs:attribute name="lowerBound" type="xs:int" use="optional"
4120 509     default="1"/>
4121 510 <xs:attribute name="upperBound" use="optional" default="1">
4122 511 <xs:simpleType>
4123 512 <xs:union>
4124 513 <xs:simpleType>
4125 514 <xs:restriction base="xs:nonNegativeInteger">
4126 515 <xs:pattern value="([1-9]+[0-9]*)"/>
4127 516 </xs:restriction>

```

```

4128 517         </xs:simpleType>
4129 518         <xs:simpleType>
4130 519             <xs:restriction base="xs:string">
4131 520                 <xs:enumeration value="unbounded"/>
4132 521             </xs:restriction>
4133 522         </xs:simpleType>
4134 523     </xs:union>
4135 524 </xs:simpleType>
4136 525 </xs:attribute>
4137 526 </xs:extension>
4138 527 </xs:complexContent>
4139 528 </xs:complexType>
4140 529
4141 530 <xs:complexType name="tRequirement">
4142 531     <xs:complexContent>
4143 532         <xs:extension base="tEntityType">
4144 533             <xs:attribute name="name" type="xs:string" use="required"/>
4145 534         </xs:extension>
4146 535     </xs:complexContent>
4147 536 </xs:complexType>
4148 537
4149 538 <xs:complexType name="tCapabilityType">
4150 539 <xs:complexContent>
4151 540     <xs:extension base="tEntityType"/>
4152 541 </xs:complexContent>
4153 542 </xs:complexType>
4154 543
4155 544 <xs:complexType name="tCapabilityDefinition">
4156 545     <xs:complexContent>
4157 546         <xs:extension base="tExtensibleElements">
4158 547             <xs:sequence>
4159 548                 <xs:element name="Constraints" minOccurs="0">
4160 549                     <xs:complexType>
4161 550                         <xs:sequence>
4162 551                             <xs:element name="Constraint" type="tConstraint"
4163 552                                 maxOccurs="unbounded"/>
4164 553                         </xs:sequence>
4165 554                     </xs:complexType>
4166 555                 </xs:element>
4167 556             </xs:sequence>
4168 557             <xs:attribute name="name" type="xs:string" use="required"/>
4169 558             <xs:attribute name="capabilityType" type="xs:QName"
4170 559                 use="required"/>
4171 560             <xs:attribute name="lowerBound" type="xs:int" use="optional"
4172 561                 default="1"/>
4173 562             <xs:attribute name="upperBound" use="optional" default="1">
4174 563                 <xs:simpleType>
4175 564                     <xs:union>
4176 565                         <xs:simpleType>
4177 566                             <xs:restriction base="xs:nonNegativeInteger">
4178 567                                 <xs:pattern value="([1-9]+[0-9]*)"/>
4179 568                             </xs:restriction>
4180 569                         </xs:simpleType>
4181 570                     </xs:union>
4182 571                 </xs:simpleType>
4183 572                 <xs:restriction base="xs:string">
4184 573                     <xs:enumeration value="unbounded"/>
4185 574                 </xs:restriction>
4185 574             </xs:simpleType>

```



```

4186 575     </xs:union>
4187 576     </xs:simpleType>
4188 577     </xs:attribute>
4189 578     </xs:extension>
4190 579     </xs:complexContent>
4191 580 </xs:complexType>
4192 581
4193 582 <xs:complexType name="tCapability">
4194 583   <xs:complexContent>
4195 584     <xs:extension base="tEntityType">
4196 585       <xs:attribute name="name" type="xs:string" use="required"/>
4197 586     </xs:extension>
4198 587   </xs:complexContent>
4199 588 </xs:complexType>
4200 589
4201 590 <xs:complexType name="tArtifactType">
4202 591   <xs:complexContent>
4203 592     <xs:extension base="tEntityType"/>
4204 593   </xs:complexContent>
4205 594 </xs:complexType>
4206 595
4207 596 <xs:complexType name="tArtifactTemplate">
4208 597   <xs:complexContent>
4209 598     <xs:extension base="tEntityTypeTemplate">
4210 599       <xs:sequence>
4211 600         <xs:element name="ArtifactReferences" minOccurs="0">
4212 601           <xs:complexType>
4213 602             <xs:sequence>
4214 603               <xs:element name="ArtifactReference" type="tArtifactReference"
4215 604                 maxOccurs="unbounded"/>
4216 605             </xs:sequence>
4217 606           </xs:complexType>
4218 607         </xs:element>
4219 608       </xs:sequence>
4220 609       <xs:attribute name="name" type="xs:string" use="optional"/>
4221 610     </xs:extension>
4222 611   </xs:complexContent>
4223 612 </xs:complexType>
4224 613
4225 614 <xs:complexType name="tDeploymentArtifacts">
4226 615   <xs:sequence>
4227 616     <xs:element name="DeploymentArtifact" type="tDeploymentArtifact"
4228 617       maxOccurs="unbounded"/>
4229 618   </xs:sequence>
4230 619 </xs:complexType>
4231 620
4232 621 <xs:complexType name="tDeploymentArtifact">
4233 622   <xs:complexContent>
4234 623     <xs:extension base="tExtensibleElements">
4235 624       <xs:attribute name="name" type="xs:string" use="required"/>
4236 625       <xs:attribute name="artifactType" type="xs:QName" use="required"/>
4237 626       <xs:attribute name="artifactRef" type="xs:QName" use="optional"/>
4238 627     </xs:extension>
4239 628   </xs:complexContent>
4240 629 </xs:complexType>
4241 630
4242 631 <xs:complexType name="tImplementationArtifacts">
4243 632   <xs:sequence>

```

```

4244 633     <xs:element name="ImplementationArtifact" maxOccurs="unbounded">
4245 634         <xs:complexType>
4246 635             <xs:complexContent>
4247 636                 <xs:extension base="tImplementationArtifact"/>
4248 637             </xs:complexContent>
4249 638         </xs:complexType>
4250 639     </xs:element>
4251 640 </xs:sequence>
4252 641 </xs:complexType>
4253 642
4254 643 <xs:complexType name="tImplementationArtifact">
4255 644     <xs:complexContent>
4256 645         <xs:extension base="tExtensibleElements">
4257 646             <xs:attribute name="interfaceName" type="xs:anyURI"
4258 647                 use="optional"/>
4259 648             <xs:attribute name="operationName" type="xs:NCName"
4260 649                 use="optional"/>
4261 650             <xs:attribute name="artifactType" type="xs:QName" use="required"/>
4262 651             <xs:attribute name="artifactRef" type="xs:QName" use="optional"/>
4263 652         </xs:extension>
4264 653     </xs:complexContent>
4265 654 </xs:complexType>
4266 655
4267 656 <xs:complexType name="tPlans">
4268 657     <xs:sequence>
4269 658         <xs:element name="Plan" type="tPlan" maxOccurs="unbounded"/>
4270 659     </xs:sequence>
4271 660     <xs:attribute name="targetNamespace" type="xs:anyURI"
4272 661         use="optional"/>
4273 662 </xs:complexType>
4274 663
4275 664 <xs:complexType name="tPlan">
4276 665     <xs:complexContent>
4277 666         <xs:extension base="tExtensibleElements">
4278 667             <xs:sequence>
4279 668                 <xs:element name="Precondition" type="tCondition" minOccurs="0"/>
4280 669                 <xs:element name="InputParameters" minOccurs="0">
4281 670                     <xs:complexType>
4282 671                         <xs:sequence>
4283 672                             <xs:element name="InputParameter" type="tParameter"
4284 673                                 maxOccurs="unbounded"/>
4285 674                         </xs:sequence>
4286 675                     </xs:complexType>
4287 676                 </xs:element>
4288 677                 <xs:element name="OutputParameters" minOccurs="0">
4289 678                     <xs:complexType>
4290 679                         <xs:sequence>
4291 680                             <xs:element name="OutputParameter" type="tParameter"
4292 681                                 maxOccurs="unbounded"/>
4293 682                         </xs:sequence>
4294 683                     </xs:complexType>
4295 684                 </xs:element>
4296 685             <xs:choice>
4297 686                 <xs:element name="PlanModel">
4298 687                     <xs:complexType>
4299 688                         <xs:sequence>
4300 689                             <xs:any namespace="##other" processContents="lax"/>
4301 690                         </xs:sequence>

```

```

4302 691     </xs:complexType>
4303 692     </xs:element>
4304 693     <xs:element name="PlanModelReference">
4305 694         <xs:complexType>
4306 695             <xs:attribute name="reference" type="xs:anyURI"
4307 696                 use="required"/>
4308 697         </xs:complexType>
4309 698     </xs:element>
4310 699 </xs:choice>
4311 700 </xs:sequence>
4312 701 <xs:attribute name="id" type="xs:ID" use="required"/>
4313 702 <xs:attribute name="name" type="xs:string" use="optional"/>
4314 703 <xs:attribute name="planType" type="xs:anyURI" use="required"/>
4315 704 <xs:attribute name="planLanguage" type="xs:anyURI" use="required"/>
4316 705 </xs:extension>
4317 706 </xs:complexContent>
4318 707 </xs:complexType>
4319 708
4320 709 <xs:complexType name="tPolicyType">
4321 710 <xs:complexContent>
4322 711 <xs:extension base="tEntityType">
4323 712 <xs:sequence>
4324 713 <xs:element name="AppliesTo" type="tAppliesTo" minOccurs="0"/>
4325 714 </xs:sequence>
4326 715 <xs:attribute name="policyLanguage" type="xs:anyURI"
4327 716     use="optional"/>
4328 717 </xs:extension>
4329 718 </xs:complexContent>
4330 719 </xs:complexType>
4331 720
4332 721 <xs:complexType name="tPolicyTemplate">
4333 722 <xs:complexContent>
4334 723 <xs:extension base="tEntityTemplate">
4335 724 <xs:attribute name="name" type="xs:string" use="optional"/>
4336 725 </xs:extension>
4337 726 </xs:complexContent>
4338 727 </xs:complexType>
4339 728
4340 729 <xs:complexType name="tAppliesTo">
4341 730 <xs:sequence>
4342 731 <xs:element name="NodeTypeReference" maxOccurs="unbounded">
4343 732 <xs:complexType>
4344 733 <xs:attribute name="typeRef" type="xs:QName" use="required"/>
4345 734 </xs:complexType>
4346 735 </xs:element>
4347 736 </xs:sequence>
4348 737 </xs:complexType>
4349 738
4350 739 <xs:complexType name="tPolicy">
4351 740 <xs:complexContent>
4352 741 <xs:extension base="tExtensibleElements">
4353 742 <xs:attribute name="name" type="xs:string" use="optional"/>
4354 743 <xs:attribute name="policyType" type="xs:QName" use="required"/>
4355 744 <xs:attribute name="policyRef" type="xs:QName" use="optional"/>
4356 745 </xs:extension>
4357 746 </xs:complexContent>
4358 747 </xs:complexType>
4359 748

```

```

4360 749 <xs:complexType name="tConstraint">
4361 750 <xs:sequence>
4362 751 <xs:any namespace="##other" processContents="lax"/>
4363 752 </xs:sequence>
4364 753 <xs:attribute name="constraintType" type="xs:anyURI" use="required"/>
4365 754 </xs:complexType>
4366 755
4367 756 <xs:complexType name="tPropertyConstraint">
4368 757 <xs:complexContent>
4369 758 <xs:extension base="tConstraint">
4370 759 <xs:attribute name="property" type="xs:string" use="required"/>
4371 760 </xs:extension>
4372 761 </xs:complexContent>
4373 762 </xs:complexType>
4374 763
4375 764 <xs:complexType name="tExtensions">
4376 765 <xs:complexContent>
4377 766 <xs:extension base="tExtensibleElements">
4378 767 <xs:sequence>
4379 768 <xs:element name="Extension" type="tExtension"
4380 769 maxOccurs="unbounded"/>
4381 770 </xs:sequence>
4382 771 </xs:extension>
4383 772 </xs:complexContent>
4384 773 </xs:complexType>
4385 774
4386 775 <xs:complexType name="tExtension">
4387 776 <xs:complexContent>
4388 777 <xs:extension base="tExtensibleElements">
4389 778 <xs:attribute name="namespace" type="xs:anyURI" use="required"/>
4390 779 <xs:attribute name="mustUnderstand" type="tBoolean" use="optional"
4391 780 default="yes"/>
4392 781 </xs:extension>
4393 782 </xs:complexContent>
4394 783 </xs:complexType>
4395 784
4396 785 <xs:complexType name="tParameter">
4397 786 <xs:attribute name="name" type="xs:string" use="required"/>
4398 787 <xs:attribute name="type" type="xs:string" use="required"/>
4399 788 <xs:attribute name="required" type="tBoolean" use="optional"
4400 789 default="yes"/>
4401 790 </xs:complexType>
4402 791
4403 792 <xs:complexType name="tInterface">
4404 793 <xs:sequence>
4405 794 <xs:element name="Operation" type="tOperation"
4406 795 maxOccurs="unbounded"/>
4407 796 </xs:sequence>
4408 797 <xs:attribute name="name" type="xs:anyURI" use="required"/>
4409 798 </xs:complexType>
4410 799
4411 800 <xs:complexType name="tExportedInterface">
4412 801 <xs:sequence>
4413 802 <xs:element name="Operation" type="tExportedOperation"
4414 803 maxOccurs="unbounded"/>
4415 804 </xs:sequence>
4416 805 <xs:attribute name="name" type="xs:anyURI" use="required"/>
4417 806 </xs:complexType>

```

```

4418 807
4419 808 <xs:complexType name="tOperation">
4420 809   <xs:complexContent>
4421 810     <xs:extension base="tExtensibleElements">
4422 811       <xs:sequence>
4423 812         <xs:element name="InputParameters" minOccurs="0">
4424 813           <xs:complexType>
4425 814             <xs:sequence>
4426 815               <xs:element name="InputParameter" type="tParameter"
4427 816                 maxOccurs="unbounded"/>
4428 817             </xs:sequence>
4429 818           </xs:complexType>
4430 819         </xs:element>
4431 820         <xs:element name="OutputParameters" minOccurs="0">
4432 821           <xs:complexType>
4433 822             <xs:sequence>
4434 823               <xs:element name="OutputParameter" type="tParameter"
4435 824                 maxOccurs="unbounded"/>
4436 825             </xs:sequence>
4437 826           </xs:complexType>
4438 827         </xs:element>
4439 828       </xs:sequence>
4440 829       <xs:attribute name="name" type="xs:NCName" use="required"/>
4441 830     </xs:extension>
4442 831   </xs:complexContent>
4443 832 </xs:complexType>
4444 833
4445 834 <xs:complexType name="tExportedOperation">
4446 835   <xs:choice>
4447 836     <xs:element name="NodeOperation">
4448 837       <xs:complexType>
4449 838         <xs:attribute name="nodeRef" type="xs:IDREF" use="required"/>
4450 839         <xs:attribute name="interfaceName" type="xs:anyURI"
4451 840           use="required"/>
4452 841         <xs:attribute name="operationName" type="xs:NCName"
4453 842           use="required"/>
4454 843       </xs:complexType>
4455 844     </xs:element>
4456 845     <xs:element name="RelationshipOperation">
4457 846       <xs:complexType>
4458 847         <xs:attribute name="relationshipRef" type="xs:IDREF"
4459 848           use="required"/>
4460 849         <xs:attribute name="interfaceName" type="xs:anyURI"
4461 850           use="required"/>
4462 851         <xs:attribute name="operationName" type="xs:NCName"
4463 852           use="required"/>
4464 853       </xs:complexType>
4465 854     </xs:element>
4466 855     <xs:element name="Plan">
4467 856       <xs:complexType>
4468 857         <xs:attribute name="planRef" type="xs:IDREF" use="required"/>
4469 858       </xs:complexType>
4470 859     </xs:element>
4471 860   </xs:choice>
4472 861   <xs:attribute name="name" type="xs:NCName" use="required"/>
4473 862 </xs:complexType>
4474 863
4475 864 <xs:complexType name="tCondition">

```

```

4476 865     <xs:sequence>
4477 866     <xs:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
4478 867     </xs:sequence>
4479 868     <xs:attribute name="expressionLanguage" type="xs:anyURI"
4480 869         use="required"/>
4481 870 </xs:complexType>
4482 871
4483 872 <xs:complexType name="tTopologyElementInstanceStates">
4484 873     <xs:sequence>
4485 874         <xs:element name="InstanceState" maxOccurs="unbounded">
4486 875             <xs:complexType>
4487 876                 <xs:attribute name="state" type="xs:anyURI" use="required"/>
4488 877                 </xs:complexType>
4489 878             </xs:element>
4490 879         </xs:sequence>
4491 880     </xs:complexType>
4492 881
4493 882 <xs:complexType name="tArtifactReference">
4494 883     <xs:choice minOccurs="0" maxOccurs="unbounded">
4495 884         <xs:element name="Include">
4496 885             <xs:complexType>
4497 886                 <xs:attribute name="pattern" type="xs:string" use="required"/>
4498 887                 </xs:complexType>
4499 888             </xs:element>
4500 889             <xs:element name="Exclude">
4501 890                 <xs:complexType>
4502 891                     <xs:attribute name="pattern" type="xs:string" use="required"/>
4503 892                     </xs:complexType>
4504 893                 </xs:element>
4505 894             </xs:choice>
4506 895             <xs:attribute name="reference" type="xs:anyURI" use="required"/>
4507 896         </xs:complexType>
4508 897
4509 898 <xs:complexType name="tRequiredContainerFeatures">
4510 899     <xs:sequence>
4511 900         <xs:element name="RequiredContainerFeature"
4512 901             type="tRequiredContainerFeature" maxOccurs="unbounded"/>
4513 902     </xs:sequence>
4514 903 </xs:complexType>
4515 904
4516 905 <xs:complexType name="tRequiredContainerFeature">
4517 906     <xs:attribute name="feature" type="xs:anyURI" use="required"/>
4518 907 </xs:complexType>
4519 908
4520 909 <xs:simpleType name="tBoolean">
4521 910     <xs:restriction base="xs:string">
4522 911         <xs:enumeration value="yes"/>
4523 912         <xs:enumeration value="no"/>
4524 913     </xs:restriction>
4525 914 </xs:simpleType>
4526 915
4527 916 <xs:simpleType name="importedURI">
4528 917     <xs:restriction base="xs:anyURI"/>
4529 918 </xs:simpleType>
4530 919
4531 920 </xs:schema>

```

4532

Appendix E. Sample

4533

This appendix contains the full sample used in this specification.

4534

E.1 Sample Service Topology Definition

4535

```
01 <Definitions name="MyServiceTemplateDefinition"
4536 02     targetNamespace="http://www.example.com/sample">
4537 03     <Types>
4538 04         <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
4539 05             elementFormDefault="qualified"
4540 06             attributeFormDefault="unqualified">
4541 07             <xs:element name="ApplicationProperties">
4542 08                 <xs:complexType>
4543 09                     <xs:sequence>
4544 10                         <xs:element name="Owner" type="xs:string"/>
4545 11                         <xs:element name="InstanceName" type="xs:string"/>
4546 12                         <xs:element name="AccountID" type="xs:string"/>
4547 13                     </xs:sequence>
4548 14                 </xs:complexType>
4549 15             </xs:element>
4550 16             <xs:element name="AppServerProperties">
4551 17                 <xs:complexType>
4552 18                     <xs:sequence>
4553 19                         <element name="HostName" type="xs:string"/>
4554 20                         <element name="IPAddress" type="xs:string"/>
4555 21                         <element name="HeapSize" type="xs:positiveInteger"/>
4556 22                         <element name="SoapPort" type="xs:positiveInteger"/>
4557 23                     </xs:sequence>
4558 24                 </xs:complexType>
4559 25             </xs:element>
4560 26         </xs:schema>
4561 27     </Types>
4562 28
4563 29     <ServiceTemplate id="MyServiceTemplate">
4564 30
4565 31         <Tags>
4566 32             <Tag name="author" value="someone@example.com"/>
4567 33         </Tags>
4568 34
4569 35         <TopologyTemplate id="SampleApplication">
4570 36
4571 37             <NodeTemplate id="MyApplication"
4572 38                 name="My Application"
4573 39                 nodeType="abc:Application">
4574 40                 <Properties>
4575 41                     <ApplicationProperties>
4576 42                         <Owner>Frank</Owner>
4577 43                         <InstanceName>Thomas' favorite application</InstanceName>
4578 44                     </ApplicationProperties>
4579 45                 </Properties>
4580 46             </NodeTemplate>
4581 47
4582 48             <NodeTemplate id="MyAppServer"
4583 49                 name="My Application Server"
```

```

4584 50         nodeType="abc:ApplicationServer"
4585 51         minInstances="0"
4586 52         maxInstances="unbounded"/>
4587 53
4588 54     <RelationshipTemplate id="MyDeploymentRelationship"
4589 55         relationshipType="abc:deployedOn">
4590 56         <SourceElement id="MyApplication"/>
4591 57         <TargetElement id="MyAppServer"/>
4592 58     </RelationshipTemplate>
4593 59
4594 60 </TopologyTemplate>
4595 61
4596 62 <Plans>
4597 63     <Plan id="DeployApplication"
4598 64         name="Sample Application Build Plan"
4599 65         planType="http://docs.oasis-
4600 66             open.org/tosca/ns/2011/12/PlanTypes/BuildPlan"
4601 67         planLanguage="http://www.omg.org/spec/BPMN/20100524/MODEL">
4602 68
4603 69         <Precondition expressionLanguage="www.example.com/text"> ?
4604 70             Run only if funding is available
4605 71         </Precondition>
4606 72
4607 73         <PlanModel>
4608 74             <process name="DeployNewApplication" id="p1">
4609 75                 <documentation>This process deploys a new instance of the
4610 76                     sample application.
4611 77                 </documentation>
4612 78
4613 79                 <task id="t1" name="CreateAccount"/>
4614 80
4615 81                 <task id="t2" name="AcquireNetworkAddresses"
4616 82                     isSequential="false"
4617 83                     loopDataInput="t2Input.LoopCounter"/>
4618 84                 <documentation>Assumption: t2 gets data of type "input"
4619 85                     as input and this data has a field names "LoopCounter"
4620 86                     that contains the actual multiplicity of the task.
4621 87                 </documentation>
4622 88
4623 89                 <task id="t3" name="DeployApplicationServer"
4624 90                     isSequential="false"
4625 91                     loopDataInput="t3Input.LoopCounter"/>
4626 92
4627 93                 <task id="t4" name="DeployApplication"
4628 94                     isSequential="false"
4629 95                     loopDataInput="t4Input.LoopCounter"/>
4630 96
4631 97                 <sequenceFlow id="s1" targetRef="t2" sourceRef="t1"/>
4632 98                 <sequenceFlow id="s2" targetRef="t3" sourceRef="t2"/>
4633 99                 <sequenceFlow id="s3" targetRef="t4" sourceRef="t3"/>
4634 100             </process>
4635 101         </PlanModel>
4636 102     </Plan>
4637 103
4638 104     <Plan id="RemoveApplication"
4639 105         planType="http://docs.oasis-
4640 106             open.org/tosca/ns/2011/12/PlanTypes/TerminationPlan"
4641 107         planLanguage="http://docs.oasis-

```



```

4642 108         open.org/wsbpel/2.0/process/executable">
4643 109         <PlanModelReference reference="prj:RemoveApp"/>
4644 110     </Plan>
4645 111 </Plans>
4646 112
4647 113 </ServiceTemplate>
4648 114
4649 115 <NodeType name="Application">
4650 116     <documentation xml:lang="EN">
4651 117         A reusable definition of a node type representing an
4652 118         application that can be deployed on application servers.
4653 119     </documentation>
4654 120     <NodeTypeProperties element="ApplicationProperties"/>
4655 121     <InstanceStates>
4656 122         <InstanceState state="http://www.example.com/started"/>
4657 123         <InstanceState state="http://www.example.com/stopped"/>
4658 124     </InstanceStates>
4659 125     <Interfaces>
4660 126         <Interface name="DeploymentInterface">
4661 127             <Operation name="DeployApplication">
4662 128                 <InputParameters>
4663 129                     <InputParamter name="InstanceName"
4664 130                         type="xs:string"/>
4665 131                     <InputParamter name="AppServerHostname"
4666 132                         type="xs:string"/>
4667 133                     <InputParamter name="ContextRoot"
4668 134                         type="xs:string"/>
4669 135                 </InputParameters>
4670 136             </Operation>
4671 137         </Interface>
4672 138     </Interfaces>
4673 139 </NodeType>
4674 140
4675 141 <NodeType name="ApplicationServer"
4676 142     targetNamespace="http://www.example.com/sample">
4677 143     <NodeTypeProperties element="AppServerProperties"/>
4678 144     <Interfaces>
4679 145         <Interface name="MyAppServerInterface">
4680 146             <Operation name="AcquireNetworkAddress"/>
4681 147             <Operation name="DeployApplicationServer"/>
4682 148         </Interface>
4683 149     </Interfaces>
4684 150 </NodeType>
4685 151
4686 152 <RelationshipType name="deployedOn">
4687 153     <documentation xml:lang="EN">
4688 154         A reusable definition of relation that expresses deployment of
4689 155         an artifact on a hosting environment.
4690 156     </documentation>
4691 157 </RelationshipType>
4692 158
4693 159 </Definitions>

```

Appendix F. Revision History

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

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

wd-13	2012-10-26	Thomas Spatzier, Derek Palma	<p>Changes for JIRA Issue TOSCA-10: More fixes to formatting and references in document according to OASIS staff comments</p> <p>Changes for JIRA Issues TOSCA-36/37: Added <code>PolicyType</code> and <code>PolicyTemplate</code> elements to allow for reusable definitions of policies.</p> <p>Changes for JIRA Issue TOSCA-57: Restructure TOSCA schema to allow for better modular definitions and separation of concerns.</p> <p>Changes for JIRA Issue TOSCA-59: Rewording to clarify overriding of deployment artifacts of Node Templates.</p> <p>Some additional minor changes in wording.</p> <p>Changes for JIRA Issue TOSCA-63: clarifying rewording</p>
wd-14	2012-11-19	Thomas Spatzier	<p>Changes for JIRA Issue TOSCA-76: Add 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>
wd-16	2013-04-15	Derek Palma, Thomas Spatzier	<p>Changes for JIRA Issue TOSCA-82: Non-material change on namespace name use</p> <p>Changes for JIRA Issue TOSCA-83: fix broken references in document</p>

4696