

---

# Topology and Orchestration Specification for Cloud Applications Version 1.0

## Working Draft 03

12 March 2012

### Technical Committee:

[OASIS Topology and Orchestration Specification for Cloud Applications \(TOSCA\) TC](#)

### Chairs:

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

### Editors:

Arvind Srinivasan ([arvindsr@us.ibm.com](mailto:arvindsr@us.ibm.com)), IBM  
Thomas Spatzier ([thomas.spatzier@de.ibm.com](mailto:thomas.spatzier@de.ibm.com)), IBM

### Declared XML namespaces:

- <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 (or simply “services” from here on). 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 [Working Draft](#) (WD) has been produced by one or more TC Members; it has not yet been voted on by the TC or [approved](#) as a Committee Draft (Committee Specification Draft or a Committee Note Draft). The OASIS document [Approval Process](#) begins officially with a TC vote to approve a WD as a Committee Draft. A TC may approve a Working Draft, revise it, and re-approve it any number of times as a Committee Draft.

Copyright © OASIS Open 2012. 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.

---

# Table of Contents

1	Introduction.....	5
2	Language Design .....	5
2.1	Dependencies on Other Specifications .....	5
2.2	Notational Conventions.....	5
2.3	Normative References .....	5
2.4	Non-Normative References .....	6
2.5	Namespaces.....	6
2.6	Language Extensibility.....	6
2.7	Overall Language Structure.....	7
2.7.1	Syntax.....	7
2.7.2	Properties .....	7
3	Core Concepts and Usage Pattern .....	10
3.1	Core Concepts.....	10
3.2	Use Cases .....	12
3.2.1	Services as Marketable Entities .....	12
3.2.2	Portability of Service Templates.....	12
3.2.3	Service Composition .....	13
3.2.4	Relation to Virtual Images .....	13
4	Node Types .....	13
4.1	Syntax.....	13
4.2	Properties.....	15
4.3	Derivation Rules .....	17
4.4	Example .....	18
5	Relationship Types.....	19
5.1	Syntax.....	19
5.2	Properties.....	19
5.3	Example .....	20
6	Topology Template.....	20
6.1	Syntax.....	20
6.2	Properties.....	22
6.3	Example .....	26
7	Plans.....	27
7.1	Syntax.....	27
7.2	Properties.....	27
7.3	Use of Process Modeling Languages.....	28
7.4	Example .....	28
8	Security Considerations .....	29
9	Conformance .....	30
Appendix A.	Portability and Interoperability Considerations .....	31
Appendix B.	Complete TOSCA Grammar .....	32
Appendix C.	TOSCA Schema.....	37
Appendix D.	Sample .....	48
D.1	Sample Service Topology Definition .....	48

Appendix E. Revision History ..... 52

---

# 1 Introduction

IT services (or just *services* in what follows) are the main asset within IT environments in general, and in cloud environments in particular. The advent of cloud computing suggests the utility of standards that enable the (semi-) automatic creation and management of services (a.k.a. service automation). These standards describe a service and how to manage it independent of the supplier creating the service and independent of any particular cloud provider and the technology hosting the service. Making service topologies (i.e. the individual components of a service and their relations) and their orchestration plans (i.e. the management procedures to create and modify a service) interoperable artifacts, enables their exchange between different environments. This specification explains how to define services in a portable and interoperable manner in a *Service Template* document.

## 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:

- WSDL 1.1
- XML Schema 1.0

and relates to:

- OVF 1.1

### 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].

### 2.3 Normative References

- |                  |   |
|------------------|---|
| [RFC2119]        | S. Bradner, <i>Key words for use in RFCs to Indicate Requirement Levels</i> , <a href="http://www.ietf.org/rfc/rfc2119.txt">http://www.ietf.org/rfc/rfc2119.txt</a> , IETF RFC 2119, March 1997.              |
| [BPEL 2.0]       | OASIS Web Services Business Process Execution Language (WS-BPEL) 2.0, <a href="http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf">http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf</a>               |
| [BPMN 2.0]       | OMG Business Process Model and Notation (BPMN) Version 2.0 - Beta 1, <a href="http://www.omg.org/spec/BPMN/2.0/">http://www.omg.org/spec/BPMN/2.0/</a>  |
| [OVF]            | Open Virtualization Format Specification Version 1.1.0, <a href="http://www.dmtf.org/standards/published_documents/DSP0243_1.1.0.pdf">http://www.dmtf.org/standards/published_documents/DSP0243_1.1.0.pdf</a> |
| [WSDL 1.1]       | Web Services Description Language (WSDL) Version 1.1, W3C Note, <a href="http://www.w3.org/TR/2001/NOTE-wsdl-20010315">http://www.w3.org/TR/2001/NOTE-wsdl-20010315</a>                                       |
| [XML Infoset]    | XML Information Set, W3C Recommendation, <a href="http://www.w3.org/TR/2001/REC-xml-infoset-20011024/">http://www.w3.org/TR/2001/REC-xml-infoset-20011024/</a>  |
| [XML Namespaces] | Namespaces in XML 1.0 (Second Edition), W3C Recommendation, <a href="http://www.w3.org/TR/REC-xml-names/">http://www.w3.org/TR/REC-xml-names/</a>   |

- 43 **[XML Schema Part 1]** XML Schema Part 1: Structures, W3C Recommendation, October 2004,  
44 <http://www.w3.org/TR/xmlschema-1/>
- 45 **[XML Schema Part 2]** XML Schema Part 2: Datatypes, W3C Recommendation, October 2004,  
46 <http://www.w3.org/TR/xmlschema-2/>
- 47 **[XMLSpec]** XML Specification, W3C Recommendation, February 1998,  
48 <http://www.w3.org/TR/1998/REC-xml-19980210>
- 49 **[XPath 1.0]** XML Path Language (XPath) Version 1.0, W3C Recommendation, November  
50 1999, <http://www.w3.org/TR/1999/REC-xpath-19991116>
- 51

52 **2.4 Non-Normative References**

53 **[Reference]** [Full reference citation]  
54

55 **2.5 Namespaces**

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

61

Prefix	Namespace
ste	<a href="http://docs.oasis-open.org/tosca/ns/2011/12">http://docs.oasis-open.org/tosca/ns/2011/12</a>
xs	<a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a>
wSDL	<a href="http://schemas.xmlsoap.org/wSDL/">http://schemas.xmlsoap.org/wSDL/</a>
bpmn	<a href="http://www.omg.org/bpmn/2.0">http://www.omg.org/bpmn/2.0</a>

62 **Table 1** Prefixes and namespaces used in this specification

63

64 All information items defined by TOSCA are identified by one of the XML namespace URIs above [XML  
65 Namespaces]. A normative XML Schema [XML Schema Part 1, XML Schema Part 2] document for  
66 TOSCA can be obtained by dereferencing one of the XML namespace URIs.

67 **2.6 Language Extensibility**

68 The TOSCA extensibility mechanism allows:

- 69 • Attributes from other namespaces to appear on any TOSCA element
- 70 • Elements from other namespaces to appear within TOSCA elements
- 71 • Extension attributes and extension elements MUST NOT contradict the semantics of any attribute  
72 or element from the TOSCA namespace

73 The specification differentiates between mandatory and optional extensions (the section below explains  
74 the syntax used to declare extensions). If a mandatory extension is used, a compliant implementation

75 MUST understand the extension. If an optional extension is used, a compliant implementation MAY  
76 ignore the extension.

## 77 2.7 Overall Language Structure

78 A *Service Template* is an XML document that consists of a Topology Template, Node Types, Relationship  
79 Types and Plans. This section explains the overall structure of a Service Template, the extension  
80 mechanism, and import features. Later sections describe in detail Topology Templates, Node Types,  
81 Relationship Types and Plans.

### 82 2.7.1 Syntax

```
83 1 <ServiceTemplate id="ID"  
84 2     name="string"?  
85 3     targetNamespace="anyURI">  
86 4  
87 5     <Extensions>?  
88 6         <Extension namespace="anyURI"  
89 7             mustUnderstand="yes|no"?/>+  
90 8     </Extensions>  
91 9  
92 10    <Import namespace="anyURI"?  
93 11        location="anyURI"?  
94 12        importType="anyURI"/>*  
95 13  
96 14    <Types>?  
97 15        <xs:schema .../>*  
98 16    </Types>  
99 17  
100 18    (  
101 19        <TopologyTemplate>  
102 20            ...  
103 21        </TopologyTemplate>  
104 22    |  
105 23        <TopologyTemplateReference reference="xs:QName">  
106 24    )?  
107 25  
108 26    <NodeTypes>?  
109 27        ...  
110 28    </NodeTypes>  
111 29  
112 30    <RelationshipTypes>?  
113 31        ...  
114 32    </RelationshipTypes>  
115 33  
116 34    <Plans>?  
117 35        ...  
118 36    </Plans>  
119 37  
120 38 </ServiceTemplate>
```

### 121 2.7.2 Properties

122 The `ServiceTemplate` element has the following properties:

- 123 • `id`: This attribute specifies the identifier of the Service Template. The identifier of the Service  
124 Template MUST be unique within the target namespace.

125

126 Note: For elements defined in this specification, the value of the `id` attribute of an element is  
127 used as the local name part of the fully-qualified name (QName) of that element, by which it can  
128 be referenced from within another definition.

- 129 • `name`: This optional attribute specifies the name of the Service Template.

130

131 Note: The `name` attribute for elements defined in this specification can generally be used as  
132 descriptive, human-readable name.

- 133 • `targetNamespace`: The value of this attribute is the namespace for the Service Template.

- 134 • `Extensions`: This element specifies namespaces of TOSCA extension attributes and  
135 extension elements. The element is optional.

136 If present, the `Extensions` element MUST include at least one `Extension` element. The  
137 `Extension` element is used to specify a namespace of TOSCA extension attributes and  
138 extension elements, and indicates whether they are mandatory or optional.

139 The attribute `mustUnderstand` is used to specify whether the extension must be understood  
140 by a compliant implementation. If the `mustUnderstand` attribute has value "yes" (which is the  
141 default value for this attribute) the extension is mandatory. Otherwise, the extension is optional. If  
142 a TOSCA implementation does not support one or more of the extensions with  
143 `mustUnderstand="yes"`, then the Service Template MUST be rejected. Optional extensions  
144 MAY be ignored. It is not necessary to declare optional extensions.

145 The same extension URI MAY be declared multiple times in the `Extensions` element. If an  
146 extension URI is identified as mandatory in one `Extension` element and optional in another,  
147 then the mandatory semantics have precedence and MUST be enforced. The extension  
148 declarations in an `Extensions` element MUST be treated as an unordered set.

- 149 • `Import`: This element declares a dependency on external Service Template, XML Schema  
150 definitions, or WSDL definitions. Any number of `Import` elements MAY appear as children of  
151 the `ServiceTemplate` element.

152 The `namespace` attribute specifies an absolute URI that identifies the imported definitions. This  
153 attribute is optional. An `Import` element without a `namespace` attribute indicates that external  
154 definitions are in use, which are not namespace-qualified. If a `namespace` attribute is specified  
155 then the imported definitions MUST be in that namespace. If no namespace is specified then the  
156 imported definitions MUST NOT contain a `targetNamespace` specification. The namespace  
157 `http://www.w3.org/2001/XMLSchema` is imported implicitly. Note, however, that there is no implicit  
158 XML Namespace prefix defined for `http://www.w3.org/2001/XMLSchema`.

159 The `location` attribute contains a URI indicating the location of a document that contains  
160 relevant definitions. The location URI MAY be a relative URI, following the usual rules for  
161 resolution of the URI base [XML Base, RFC 2396]. The `location` attribute is optional. An  
162 `Import` element without a `location` attribute indicates that external definitions are used but  
163 makes no statement about where those definitions might be found. The `location` attribute is a  
164 hint and a TOSCA compliant implementation is not obliged to retrieve the document being  
165 imported from the specified location.

166 The mandatory `importType` attribute identifies the type of document being imported by  
167 providing an absolute URI that identifies the encoding language used in the document. The value  
168 of the `importType` attribute MUST be set to `http://docs.oasis-open.org/tosca/ns/2011/12` when  
169 importing Service Template documents, to `http://schemas.xmlsoap.org/wsdl/` when importing  
170 WSDL 1.1 documents, and to `http://www.w3.org/2001/XMLSchema` when importing an XSD  
171 document.

172 According to these rules, it is permissible to have an `Import` element without `namespace` and  
173 `location` attributes, and only containing an `importType` attribute. Such an `Import`



174 element indicates that external definitions of the indicated type are in use that are not  
175 namespace-qualified, and makes no statement about where those definitions might be found.

176 A Service Template MUST define or import all Topology Template, Node Types, Relationship  
177 Types, Plans, WSDL definitions, and XML Schema documents it uses. In order to support the use  
178 of definitions from namespaces spanning multiple documents, a Service Template MAY include  
179 more than one import declaration for the same namespace and importType. Where a service  
180 template has more than one import declaration for a given namespace and importType, each  
181 declaration MUST include a different location value. `Import` elements are conceptually  
182 unordered. A Service Template MUST be rejected if the imported documents contain conflicting  
183 definitions of a component used by the importing Service Template.

184 Documents (or namespaces) imported by an imported document (or namespace) are not  
185 transitively imported by a TOSCA compliant implementation. In particular, this means that if an  
186 external item is used by an element enclosed in the Service Template, then a document (or  
187 namespace) that defines that item MUST be directly imported by the Service Template. This  
188 requirement does not limit the ability of the imported document itself to import other documents or  
189 namespaces.

190 • `Types`: This element specifies XML definitions introduced within the Service Template  
191 document. Such definitions are provided within one or more separate Schema Definitions (usually  
192 `xs:schema` elements). The `Types` element defines XML definitions within a Service Template  
193 file without having to define these XML definitions in separate files and import them. Note, that an  
194 `xs:schema` element nested in the `Types` element MUST be a valid XML schema definition. In  
195 case the `targetNamespace` attribute of a nested `xs:schema` element is not specified, all  
196 definitions within this element become part of the target namespace of the encompassing  
197 `ServiceTemplate` element.

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

201 • `TopologyTemplate`: This element specifies in place the topological structure of an IT service  
202 by means of a directed graph.

203  
204 The main ingredients of a Topology Template are a set of Node Templates and Relationship  
205 Templates. The Node Templates are the nodes of the directed graph. The Relationship  
206 Templates are the directed edges between the nodes; each indicates the semantics of the  
207 corresponding relationships.

208 • `TopologyTemplateReference`: This element references a Topology Template. Its  
209 `reference` attribute specifies the QName of the definition available by reference in the  
210 document under definition. The namespace of the referenced Topology Template MUST be  
211 imported into the Service Template by means of an `Import` element.

212  
213 Note that either zero or one Topology Template MUST occur in a Service Template, either  
214 defined in place via a `TopologyTemplate` element or referenced via a  
215 `TopologyTemplateReference`.

216 • `NodeTypes`: This element specifies the types of Node (Templates), i.e., their properties and  
217 behavior.

218 • `RelationshipTypes`: This element specifies the types of relationships, i.e. the kind of links  
219 between Node Templates within a Service Template, and their properties.

220 • `Plans`: This element specifies the operational behavior of the service. Each `Plan` contained in  
221 the `Plans` element specifies how to create, terminate or manage the service.

222 A Service Template document can be intended to be instantiated into a service instance or it can be  
223 intended to be composed into other Service Templates. A Service Template document intended to be  
224 instantiated MUST contain either a `TopologyTemplate` or a `TopologyTemplateReference`,  
225 but not both. A Service Template document intended to be composed MUST include at least one of a  
226 `NodeTypes`, `RelationshipTypes`, or `Plans` element. This technique supports a modular definition  
227 of Service Templates. For example, one document can contain only Node Types that are referenced by a  
228 Service Template document that contains just a Topology Template and Plans. Similarly, Node Type  
229 Properties can be defined in separate XML Schema Definitions that are imported and referenced when  
230 defining a Node Type.

231 Example of the use of a type definition:

```
232 <Types>  
233   <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"  
234     elementFormDefault="qualified"  
235     attributeFormDefault="unqualified">  
236     <xs:element name="ProjectProperties">  
237       <xs:complexType>  
238         <xs:sequence>  
239           <xs:element name="Owner" type="xs:string"/>  
240           <xs:element name="ProjectName" type="xs:string"/>  
241           <xs:element name="AccountID" type="xs:string"/>  
242         </xs:sequence>  
243       </xs:complexType>  
244     </xs:element>  
245   </xs:schema>  
246 </Types>
```

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

```
250 1 <documentation source="anyURI"? xml:lang="language"?>  
251 2   ...  
252 3 </documentation>
```

253 Example of use of a documentation:

```
254 <ServiceTemplate id="myService" name="My Service" ...>  
255   <documentation xml:lang="EN">  
256     This is a simple example of the usage of the documentation  
257     element as nested under a ServiceTemplate element.  
258   </documentation>  
259 </ServiceTemplate>
```

## 262 3 Core Concepts and Usage Pattern

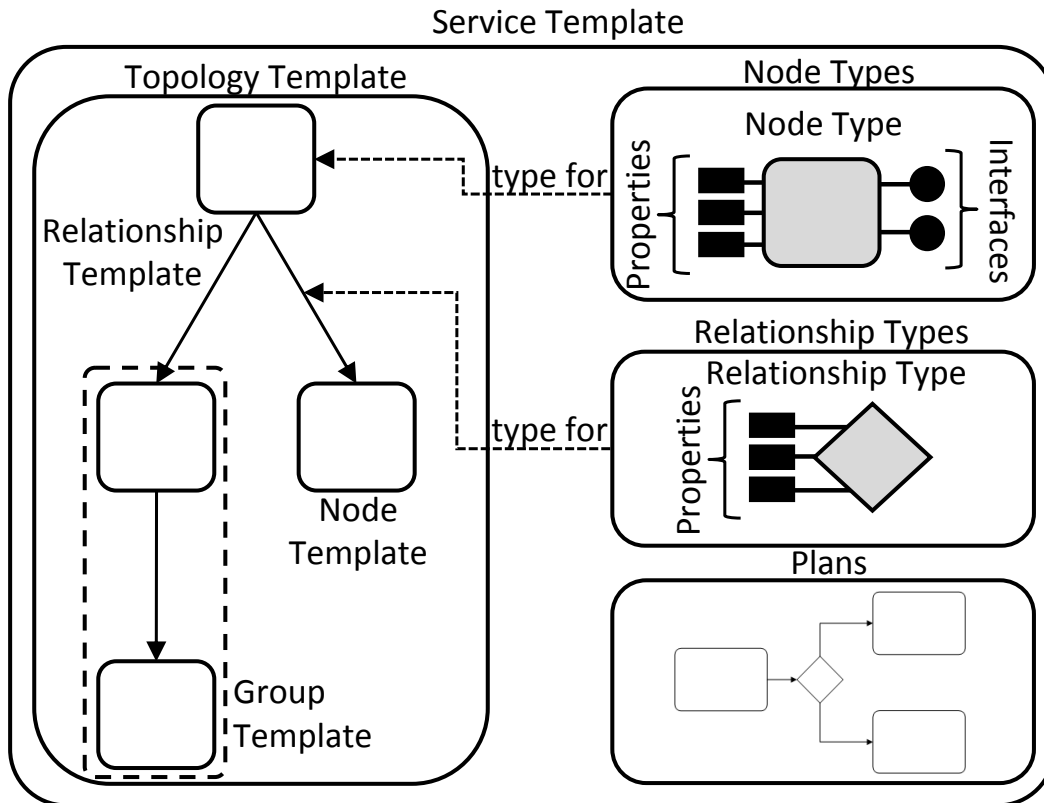
263 The main concepts behind TOSCA are described and some usage patterns of Service Templates are  
264 sketched.

### 265 3.1 Core Concepts

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

271

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



279

280

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

281 For example, consider a service that consists of an application server, a process engine, and a process  
 282 model. A Topology Template defining that service would include one Node Template of Node Type  
 283 "application server", another Node Template of Node Type "process engine", and a third Node Template  
 284 of Node Type "process model". The application server Node Type defines properties like the IP address  
 285 of an instance of this type, an operation for installing the application server with the corresponding IP  
 286 address, and an operation for shutting down an instance of this application server. A constraint in the  
 287 Node Template can specify a range of IP addresses available when making a concrete application server  
 288 available.

289 A *Relationship Template* specifies the occurrence of a relationship between nodes in a Topology  
 290 Template. Each Relationship Template refers to a Relationship Type that defines the semantics and any  
 291 properties of the relationship. Relationship Types are defined separately for reuse purposes. The  
 292 Relationship Template indicates the elements it connects and the direction of the relationship by defining  
 293 one source and one target element (in nested `SourceElement` and `TargetElement` elements). The  
 294 Relationship Template also defines any constraints with the optional `RelationshipConstraints`  
 295 element.

296 For example, a relationship can be established between the process engine Node Template and  
 297 application server Node Template with the meaning "hosted by", and between the process model Node  
 298 Template and process engine Node Template with meaning "deployed on".

299 A deployed service is an instance of a Service Template. More precisely, the instance is derived by  
 300 instantiating the Topology Template of its Service Template, most often by running a special plan defined

301 for the Service Template, often referred to as build plan. The build plan will provide actual values for the  
302 various properties of the various Node Templates and Relationship Templates of the Topology Template.  
303 These values can come from input passed in by users as triggered by human interactions defined within  
304 the build plan, by automated operations defined within the build plan (such as a directory lookup), or the  
305 templates can specify default values for some properties. The build plan will typically make use of  
306 operations of the Node Types of the Node Templates.

307 For example, the application server Node Template will be instantiated by installing an actual application  
308 server at a concrete IP address considering the specified range of IP addresses. Next, the process  
309 engine Node Template will be instantiated by installing a concrete process engine on that application  
310 server (as indicated by the “hosted by” relationship template). Finally, the process model Node Template  
311 will be instantiated by deploying the process model on that process engine (as indicated by the “deployed  
312 on” relationship template).

313 *Plans* defined in a Service Template describe the management aspects of service instances, especially  
314 their creation and termination. These plans are defined as process models, i.e. a workflow of one or more  
315 steps. Instead of providing another language for defining process models, the specification relies on  
316 existing languages like BPMN or BPEL. Relying on existing standards in this space facilitates portability  
317 and interoperability, but any language for defining process models can be used. The TOSCA metamodel  
318 provides containers to either refer to a process model (via *Plan Model Reference*) or to include the actual  
319 model in the plan (via *Plan Model*). A process model can contain tasks (using BPMN terminology) that  
320 refer to operations of Interfaces of Node Templates or any other interface (e.g. the invocation of an  
321 external service for licensing); in doing so, a plan can directly manipulate nodes of the topology of a  
322 service or interact with external systems.

## 323 **3.2 Use Cases**

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

### 325 **3.2.1 Services as Marketable Entities**

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

334 Making a concrete instance of a Topology Template can be done by running a corresponding Plan (so-  
335 called instantiating management plan, a.k.a. build plan). This build plan could be provided by the service  
336 developer who also creates the Service Template. The build plan can be adapted to the concrete  
337 environment of a particular service provider. Other management plans useful in various states of the  
338 whole lifecycle of a service could be specified as part of a Service Template. Similar to build plans such  
339 management plans can be adapted to the concrete environment of a particular service provider.

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

### 347 **3.2.2 Portability of Service Templates**

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

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

### 356 3.2.3 Service Composition

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

### 363 3.2.4 Relation to Virtual Images

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

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

## 376 4 Node Types

377 This chapter specifies how *Node Types* are defined. A Node Type is a reusable entity that defines the  
378 type of one or more Node Templates. As such, a Node Type defines observable properties via *Node*  
379 *Type Properties*. A Node Type can inherit properties from another Node Type by means of the  
380 *DerivedFrom* element. The functions that can be performed on (an instance of) a corresponding  
381 Node Template are defined by the *Interfaces* of the Node Type. Finally, interfaces supporting  
382 management Policies are defined for a Node Type.

### 383 4.1 Syntax

```
384 1 <NodeTypes>?  
385 2  
386 3 <NodeType id="ID"  
387 4     name="string"?>+  
388 5  
389 6 <NodeTypeProperties element="QName"?  
390 7     type="QName"?/>?  
391 8  
392 9 <DerivedFrom nodeTypeRef="QName"/>?  
393 10  
394 11 <InstanceStates>?  
395 12 <InstanceState state="anyURI">+  
396 13 </InstanceStates>  
397 14  
398 15 <Interfaces>?  
399 16
```

```

400 17     <Interface>+
401 18
402 19     (
403 20         <WSDL portType="QName"
404 21             operation="NCName"?>+
405 22     |
406 23         <REST method="GET | PUT | POST | DELETE"
407 24             requestURI="anyURI"
408 25             requestPayload="QName"?
409 26             responsePayload="QName"?>+
410 27     |
411 28         <Operation name="NCame">+
412 29
413 30         <InputParameters>?
414 31
415 32             <InputParamter name="string"
416 33                 type="string"
417 34                 required="yes|no">+
418 35
419 36         </InputParameters>
420 37
421 38         <OutputParameters>?
422 39
423 40             <OutputParamter name="string"
424 41                 type="string"
425 42                 required="yes|no">+
426 43
427 44         </OutputParameters>
428 45
429 46         <Implementations>
430 47
431 48             <Implementation implementationID="anyURI"?
432 49                 language="anyURI"?>+
433 50         (
434 51             <ImplementationProper>?
435 52                 code
436 53             </ImplementationProper>
437 54         |
438 55             <ImplementationReference ref="anyURI"/>?
439 56         )
440 57         <Implementation>
441 58
442 59         </Implementations>
443 60     </Operation>
444 61 )
445 62
446 63 </Interface>
447 64
448 65 </Interfaces>
449 66
450 67 <Policies>?
451 68     <Policy name="string" type="anyURI">+
452 69         policy specific content
453 70     </Policy>
454 71 </Policies>
455 72
456 73 <DeploymentArtifacts>?
457 74     <DeploymentArtifact name="string" type="anyURI">+

```

```
458 75         artifact specific content
459 76     </DeploymentArtifact>
460 77 </DeploymentArtifacts>
461 78
462 79 </NodeType>
463 80
464 81 </NodeTypes>
```

## 4.2 Properties

466 The `NodeType` element has the following properties:

- 467 • `id`: This attribute specifies the identifier of the Node Type. The identifier of the Node Type **MUST**  
468 be unique within the target namespace.
- 469 • `name`: This optional attribute specifies the name of the Node Type.
- 470 • `NodeTypeProperties`: These are the observable properties of the Node Type, such as its  
471 configuration and state.
- 472 • `DerivedFrom`: This is an optional reference to another Node Type from which this Node Type  
473 derives. Conflicting definitions are resolved by the rule that local new definitions always override  
474 derived definitions. See section 4.3 Derivation Rules for details.
- 475 • `InstanceStates`: This optional element lists the set of states an instance of this Node Type  
476 can occupy at runtime.
- 477 • `Interfaces`: These are the definitions of functions that can be performed on (instances of) this  
478 Node Type.
- 479 • `Policies`: The nested list of elements provides information related to a particular management  
480 aspect like billing or monitoring.
- 481 • `DeploymentArtifacts`: This element specifies deployment artifacts relevant for the Node  
482 Type. A deployment artifact is an entity that – if specified – is needed for creating an instance of  
483 the corresponding Node Type. For example, a virtual image could be a deployment artifact of a  
484 JEE server.

485 The `NodeTypeProperties` element has one but not both of the following properties:

- 486 • The `element` attribute provides the QName of an XML element defining the structure of the  
487 Node Type Properties.
- 488 • The `type` attribute provides the QName of an XML (complex) type defining the structure of the  
489 Node Type Properties.

490 The `DerivedFrom` element has the following properties:

- 491 • `nodeTypeRef`: The QName specifies the Node Type from which this Node Type derives its  
492 definitions.

493 The `InstanceStates` element has the following properties:

- 494 • `InstanceState`: specifies a potential state.

495 The `InstanceState` element has the following properties:

- 496 • `state`: a URI that represents a potential state.

497 The `Interface` element has one of the following properties:

- 498 • `WSDL`: specifies a WSDL port type or an operation of a port type as part of the Interface.

- 499 • `REST`: specifies an HTTP request as part of the Interface.
- 500 • `Operation`: specifies a proprietary implementation as part of the Interface.

501 The `WSDL` element has the following properties:

- 502 • `portType`: This is the QName of the port type that contains the definition of one or more  
503 operations defined as part of the Interface. Note that the namespace of the `portType` MUST be  
504 imported.
- 505 • `operation`: This optional attribute specifies the name of a single operation of the port type to  
506 become part of the Node Type Interface. If this attribute is not specified, the complete WSDL port  
507 type becomes part of the Node Type Interface.

508 The `REST` element has the following properties:

- 509 • `method`: This is the name of an HTTP method (often in a REST-style Interface).
- 510 • `requestURI`: This is the requestURI necessary to create the request.
- 511 • `requestPayload`: The QName specifies the schema of the HTTP message payload passed  
512 with the request to the HTTP processor.
- 513 • `responsePayload`: The QName specifies the schema of the payload passed with the  
514 response message from the HTTP processor.

515 **Note:** The combination of `method` and `requestURI` SHOULD uniquely identify a `REST` element  
516 within the Service Template.

517 The `Operation` element has the following properties:

- 518 • `name`: This is the name of the operation. The name of an operation SHOULD be unique within a  
519 Node Type.
- 520 • `InputParameters`: This optional property contains one or more nested `InputParameter`  
521 elements. Each such element specifies three attributes: the `name` of the parameter, its `type`,  
522 and whether it has to be available as input (`required` attribute with a value of “yes”, which is  
523 the default) or not (value “no”).  
524 Note that the types of the parameters specified for an operation MUST comply with the type  
525 systems of the languages of implementations’ proper.
- 526 • `OutputParameters`: This optional property contains one or more nested  
527 `OutputParameter` elements. Each such element specifies the `name` of the parameter and its  
528 `type`.
- 529 • `Implementations`: This element contains one or more `Implementation` elements, each  
530 of which encompasses either the actual `code` of the implementation of the operation or a  
531 reference to the code. The `implementationID` attribute of the `Implementation` element  
532 allows for providing different implementations for the same operation – this is necessary because  
533 the implementation often depends on the environment the operation will run in. The `language`  
534 attribute allows to specify the language of the implementation, e.g. one implementation might be  
535 provided as a perl script, another one as php, and so on.

536 The `Policy` element has the following properties:

- 537 • The `type` attribute specifies the kind of policy (e.g. management practice) supported by an  
538 instance of the Node Type containing this element. The `name` attribute defines the name of the  
539 policy. The name value MUST be unique within a given Node Type containing the current  
540 definition of the Policy.



541 Consider a hypothetical billing policy. In this example the type `www.sample.com/BillingPractice`  
542 could define a policy for billing usage of a service instance. The policy specific content can define  
543 the interface providing the operations to perform billing. Further content could specify the  
544 granularity of the base for payment, e.g. it could provide an enumeration with the possible values  
545 “service”, “resource”, and “labor”. A value of “service” might specify that an instance of the  
546 corresponding node will be billed during its instance lifetime. A value of “resource” might specify  
547 that the resources consumed by an instance will be billed. A value of “labor” might specify that the  
548 use of a plan affecting a node instance will be billed.

549 The `DeploymentArtifact` element has the following properties:

- 550 • `name`: The attribute specifies the name of the artifact. Note, that uniqueness of the name within  
551 the scope of the encompassing Node Type SHOULD be guaranteed by the definition.
- 552 • `type`: The attribute specifies the type of the deployment artifact definition that is related to the  
553 Node Type, i.e. the attribute gives a hint how to interpret the body of the  
554 `DeploymentArtifact` element.

555 Note, that the combination of `name` and `type` SHOULD be unique within the scope of the Node  
556 Type.

- 557 • The body of this element contains the type-specific content.

558  
559 For example, if the `type` attribute contains the value  
560 `http://docs.oasis-open.org/tosca/ns/2011/12/deploymentArtifacts/ovfRef`, the body will contain an  
561 XML fragment with a reference to an OVF package and a mapping between service template  
562 data and elements of the respective OVF envelope.

### 563 4.3 Derivation Rules

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

- 565 • **Node Type Properties**: It is assumed that the XML element (or type) representing the Node Type  
566 Properties extends the XML element (or type) of the Node Type Properties of the Node Type  
567 referenced in the `DerivedFrom` element.
- 568 • **Instance States**: The set of instance states of this Node Type consists of the set union of the  
569 instances states defined by the Nodes Type derived from and the instance states defined by this  
570 Node Type. A set of instance states of the same name will be combined into a single instance  
571 state of the same name.
- 572 • **WSDL**: The set of WSDL operations of this Node Type consists of the set union of the WSDL  
573 operations defined by the Node Type derived from and the WSDL operations defined by the Node  
574 Type. A WSDL operation defined by this Node Type substitutes a WSDL operation with the same  
575 name and signature of the Node Type derived from.
- 576 • **REST**: The set of REST requests of this Node Type consists of the set union of the REST  
577 requests defined by the Nodes Type derived from and the REST requests defined by this Node  
578 Type. A REST request defined by this Node Type substitutes a REST request with the same  
579 identity of the Node Type derived from.
- 580 • **Operation**: The set of operations of this Node Type consists of the set union of the operations  
581 defined by the Nodes Type derived from and the operations defined by this Node Type. An  
582 operation defined by this Node Type substitutes an operation with the same name of the Node  
583 Type derived from.
- 584 • **Deployment Artifacts**: The set of deployment artifacts of this Node Type consists of the set union  
585 of the deployment artifacts defined by the Nodes Type derived from and the deployment artifacts

586 defined by this Node Type. A deployment artifact defined by this Node Type substitutes a  
587 deployment artifact with the same name and type of the Node Type derived from.

- 588 • Policies: The set of policies of this Node Type consists of the set union of the policies defined by  
589 the Nodes Type derived from and the policies defined by this Node Type. A policy defined by this  
590 Node Type substitutes a policy with the same name and type of the Node Type derived from.

## 591 4.4 Example

592 The following example defines the Node Type "Project". It is defined in a Service Template "myService"  
593 within the target namespace "http://www.ibm.com/sample". Thus, by importing the corresponding  
594 namespace in another Service Template, the Project Node Type is available for use in the other Service  
595 Template.

```
596 <ServiceTemplate id="myService" name="My Service"  
597     targetNamespace="http://www.ibm.com/sample">  
598  
599     <NodeTypes>  
600  
601         <NodeType id="Project" name="My Project">  
602  
603             <documentation xml:lang="EN">  
604                 A reusable definition of a node type supporting  
605                 the creation of new projects.  
606             </documentation>  
607  
608             <NodeTypeProperties element="ProjectProperties"/>  
609  
610             <InstanceStates>  
611                 <InstanceState state="www.my.com/active"/>  
612                 <InstanceState state="www.my.com/onHalt"/>  
613             </InstanceStates>  
614  
615             <Interfaces>  
616                 <Interface>  
617                     <Operation name="CreateProject">  
618                         <InputParameters>  
619                             <InputParamter name="ProjectName"  
620                                 type="string"/>  
621                             <InputParamter name="Owner"  
622                                 type="string"/>  
623                             <InputParamter name="AccountID"  
624                                 type="string"/>  
625                         </InputParameters>  
626                         <Implementations>  
627                             <Implementation>  
628                                 ...  
629                             </Implementation>  
630                         </Implementations>  
631                     </Operation>  
632                 </Interface>  
633             </Interfaces>  
634  
635         </NodeType>  
636  
637     </NodeTypes>  
638  
639 </ServiceTemplate>
```

640 The Node Type “Project” has three Node Type Properties defined as an XML element in the `Types`  
641 element definition of the Service Template document: `Owner`, `ProjectName` and `AccountID` which are all  
642 of type “string”. An instance of the Node Type “Project” could be “active” (more precise in state  
643 `www.my.com/active`) or “on hold” (more precise in state “`www.my.com/onHold`”). A single Interface is  
644 defined for this Node Type, and this Interface is defined by an Operation, i.e. its actual implementation is  
645 defined by the definition of the Operation. The Operation has the name `CreateProject` and two Input  
646 Parameters (exploiting the default value “yes” of the attribute `required` of the `InputParameter`  
647 element). The names of these two Input Parameters are `ProjectName` and `AccountID`, both of type  
648 “string”.

## 649 5 Relationship Types

650 This chapter specifies how *Relationship Types* are defined. A Relationship Type is a reusable entity that  
651 defines the type of one or more Relationship Templates between Node Templates. A Relationship Type  
652 can define observable properties via *Relationship Type Properties*. Furthermore, it defines the potential  
653 states an instance of it might reveal at runtime.

### 654 5.1 Syntax

```
655 1 <RelationshipTypes>  
656 2  
657 3   <RelationshipType id="ID"  
658 4     name="string"?  
659 5     semantics="anyURI"  
660 6     cascadingDeletion="yes|no"?>+  
661 7  
662 8     <RelationshipTypeProperties element="QName"?  
663 9       type="QName"?/>?  
664 10  
665 11   <InstanceStates>?  
666 12     <InstanceState state="anyURI">+  
667 13   </InstanceStates>  
668 14  
669 15 </RelationshipType>  
670 16  
671 17 </RelationshipTypes>
```

### 672 5.2 Properties

673 The `RelationshipType` element has the following properties:

- 674 • `id`: This attribute specifies the identifier of the Relationship Type. The identifier of the  
675 Relationship Type MUST be unique within the target namespace.
- 676 • `name`: This optional attribute specifies the name of the Relationship Type.
- 677 • `semantics`: The meaning or expected behavior of an instance of this Relationship Type.
- 678 • `cascadingDeletion`: If set to “yes” the target of an instance of a Relationship Template of  
679 this RelationshipType is automatically deleted when the source of the instance of the Relationship  
680 Template is deleted.

681 The `RelationshipTypeProperties` element has the following properties:

- 682 • `element`: The QName value of this attribute refers to an XML element defining the structure of  
683 the Relationship Type Properties.
- 684 • `type`: The QName value of this attribute refers to an XML (complex) type defining the structure  
685 of the Relationship Type Properties.

686 Either the `element` attribute or the `type` attribute MUST be specified, but not both.

687 The `InstanceStates` element has the following properties:

- 688 • `InstanceState`: specifies a potential state.

689 The `InstanceState` element has the following properties:

- 690 • `state`: a URI that represents a potential state.

## 691 5.3 Example

692 The following example defines the Relationship Type “processDeployedOn”. The meaning of this  
693 Relationship Type is that “a process is deployed on a hosting environment” (indicated by the URI value of  
694 the `semantics` attribute). When the source of an instance of a Relationship Template referring to this  
695 Relationship Type is deleted, its target is automatically deleted as well. The Relationship Type has  
696 Relationship Type Properties defined in the `Types` section of the same Service Template document as  
697 the “ProcessDeployedOnProperties” element. The states an instance of this Relationship Type can be in  
698 are also listed.

```
699 <RelationshipTypes>
700
701   <RelationshipType id="processDeployedOn"
702     name="Process is deployed on"
703     semantics="www.my.com/RelSemantics/procDeployedOn"
704     cascadingDeletion="yes">
705
706     <RelationshipTypeProperties element="ProcessDeployedOnProperties"/>
707
708     <InstanceStates>
709       <InstanceState state="www.my.com/successfullyDeployed"/>
710       <InstanceState state="www.my.com/failed"/>
711     </InstanceStates>
712
713   </RelationshipType>
714
715 </RelationshipTypes>
```

## 716 6 Topology Template

717 This chapter specifies how *Topology Templates* are defined. A Topology Template defines the overall  
718 structure of an IT service, i.e. the components it consists of, the relations between those components, as  
719 well as grouping of components. The components of a service are referred to as *Node Templates*, the  
720 relations between the components are referred to as *Relationship Templates*, and groupings are referred  
721 to as *Group Templates*.

### 722 6.1 Syntax

```
723 1   <TopologyTemplate id="ID"
724 2     name="string"?>
725 3
726 4   (
727 5     <NodeTemplate id="ID"
728 6       name="string"?
729 7       nodeType="QName"
730 8       minInstances="int"?
731 9       maxInstances="int|string"?>
732 10
733 11     <PropertyDefaults?>
734 12       XML fragment
```

```

735 13      </PropertyDefaults>
736 14
737 15      <PropertyConstraints>?
738 16
739 17          <PropertyConstraint property="string"
740 18              constraintType="anyURI">+
741 19              constraint?
742 20          </PropertyConstraint>
743 21
744 22      </PropertyConstraints>
745 23
746 24      <Policies>?
747 25          <Policy name="string" type="anyURI">+
748 26              policy specific content
749 27          </Policy>
750 28      </Policies>
751 29
752 30      <EnvironmentConstraints>?
753 31          <EnvironmentConstraint constraintType="anyURI">+
754 32              constraint type specific content?
755 33          </EnvironmentConstraint>
756 34      </EnvironmentConstraints>
757 35
758 36      <DeploymentArtifacts>?
759 37          <DeploymentArtifact name="string" type="anyURI">+
760 38              artifact specific content
761 39          </DeploymentArtifact>
762 40      </DeploymentArtifacts>
763 41
764 42  </NodeTemplate>
765 43  |
766 44  <RelationshipTemplate id="ID"
767 45              name="string"?
768 46              relationshipType="QName">
769 47
770 48      <SourceElement id="IDREF"/>
771 49
772 50      ( <TargetElement id="IDREF"/>
773 51          |
774 52          <TargetElementReference id="QName"/>
775 53      )
776 54
777 55      <PropertyDefaults>?
778 56          XML fragment
779 57      </PropertyDefaults>
780 58
781 59      <PropertyConstraints>?
782 60
783 61          <PropertyConstraint property="string"
784 62              constraintType="anyURI">+
785 63              constraint?
786 64          </PropertyConstraint>
787 65
788 66      </PropertyConstraints>
789 67
790 68      <RelationshipConstraints>?
791 69
792 70          <RelationshipConstraint constraintType="anyURI">+

```

```

793 71         constraint?
794 72         </RelationshipConstraint>
795 73
796 74     </RelationshipConstraints>
797 75
798 76 </RelationshipTemplate>
799 77 |
800 78 <GroupTemplate id="ID"
801 79     name="string"?
802 80     minInstances="int"?
803 81     maxInstances="int|string"?>
804 82
805 83     (
806 84     <NodeTemplate ... />
807 85     |
808 86     <RelationshipTemplate ... />
809 87     |
810 88     <GroupTemplate ... />
811 89     )+
812 90
813 91     <Policies>?
814 92     <Policy name="string" type="anyURI">+
815 93     policy specific content
816 94     </Policy>
817 95     </Policies>
818 96
819 97 </GroupTemplate>
820 98 )+
821 99
822 100 </TopologyTemplate>

```

## 823 6.2 Properties

824 The `TopologyTemplate` element has the following properties:

- 825 • `id`: This attribute specifies the identifier of the Topology Template. The identifier of the Topology  
826 Template **MUST** be unique within the target namespace.
- 827 • `name`: This optional attribute specifies the name of the Topology Template.
- 828 • `NodeTemplate`: This is a kind of a component making up the IT service.
- 829 • `RelationshipTemplate`: This is a kind of relationship between the components (nodes or  
830 groups) of the service.
- 831 • `GroupTemplate`: This is a grouping of node templates, relationship templates, or (nested)  
832 group templates within the Topology Templates to express a special association between the  
833 grouped elements.

834 A Topology Template can contain any number of Node Templates, Relationship Templates, or Group  
835 Templates (i.e. “elements”). For each specified Relationship Template (either defined as a direct child of  
836 the Topology Template or within a Group Template) the source element and target element **MUST** be  
837 specified in the Topology Template except for target elements that are referenced (via a target element  
838 reference).

839 The `NodeTemplate` element has the following properties:

- 840 • `id`: This attribute specifies the identifier of the Node Template. The identifier of the Node  
841 Template **MUST** be unique within the target namespace.

- 842 • `name`: This optional attribute specifies the name of the Node Template.
- 843 • `nodeType`: The QName value of this attribute refers to the Node Type providing the type of the  
844 Node Template.
- 845 • `minInstances`: This integer attribute specifies the minimum number of instances to be created  
846 when instantiating the Node Template. The default value of this attribute is 1..The value of  
847 `minInstances` MUST NOT be less than 0.
- 848 • `maxInstances`: This attribute specifies the maximum number of instances that can be created  
849 when instantiating the Node Template. The default value of this attribute is 1. If the string is set to  
850 “unbounded”, an unbounded number of instances can be created. The value of `maxInstances`  
851 MUST be 1 or greater and MUST NOT be less than the value specified for `minInstances`.
- 852 • `PropertyDefaults`: Specifies initial values for one or more of the Node Type Properties of  
853 the Node Type providing the property definitions in the concrete context of the Node Template.  
854 The initial values are specified by providing an instance document of the XML schema of the  
855 corresponding Node Type Properties. This instance document considers the inheritance structure  
856 deduced by the `DerivedFrom` property of the Node Type referenced by the `nodeType`  
857 attribute of the Node Template.  
858  
859 The instance document of the XML schema might not validate against the existence constraints  
860 of the corresponding schema: not all node type properties might have an initial value assigned,  
861 i.e. mandatory elements or attributes might be missing in the instance provided by the Property  
862 Defaults element. Once the defined Node Template has been instantiated, any XML  
863 representation of the Node Type properties MUST validate according to the associated XML  
864 schema definition.
- 865 • `PropertyConstraints`: Specifies constraints on the use of one or more of the Node Type  
866 Properties of the Node Type providing the property definitions for the Node Template.  
867 Each constraint is specified by means of a separate nested `PropertyConstraint` element.  
868 This element contains the actual encoding of the constraint.
- 869 • `Policies`: Specifies policies of the Node Template. Each policy is specified by means of a  
870 separate nested `Policy` element. This element contains the actual policy specific content of the  
871 policy.  
872 Note, that a policy specified in the Node Template overrides any policy of the same name and  
873 type that might be specified with the Node Type of this Node Template.  
874  
875 Any policies of the Node Type that are not overridden are combined with the policies of the Node  
876 Template.
- 877 • `EnvironmentConstraints`: The nested `EnvironmentConstraint` elements of the  
878 Node Template under definition constrain the runtime environment for the corresponding  
879 component of a service. For example, constraints on network security settings of the hosting  
880 environment or requirements on the existence of certain resources might be defined within the  
881 environment constraints definition of a Node Template.
- 882 • `DeploymentArtifacts`: This element specifies the deployment artifacts relevant for the  
883 Node Template under definition.  
884  
885 Its nested `DeploymentArtifact` elements specify details about individual deployment  
886 artifacts. The name attribute of a `DeploymentArtifact` element specifies the name of the  
887 artifact. Uniqueness of the name within the scope of the encompassing Node Template SHOULD  
888 be guaranteed by the definition. The `type` attribute of a `DeploymentArtifact` element  
889 specifies the type of the deployment artifact definition that is related to the Node Template, i.e.

890 the attribute gives a hint how to interpret the body of the `DeploymentArtifact` element. The  
891 body of this element contains the type-specific content.

892

893 For example, if the `type` attribute contains the value  
894 `http://docs.oasis-open.org/tosca/ns/2011/12/deploymentArtifacts/ovfRef`, the body will contain an  
895 XML fragment with a reference to an OVF package and a mapping between service template  
896 data and elements of the respective OVF envelope.

897

898 Note, that a deployment artifact specified with the Node Template under definition overrides any  
899 deployment artifact of the same name and the same type specified with the Node Type given as  
900 value of the `nodeType` attribute of the Node Template under definition.

901

902 Otherwise, the deployment artifacts of the Node Type given as value of the `nodeType` attribute  
903 of the Node Template under definition and the deployment artifacts defined with the Node  
904 Template are combined.

905

906 The `PropertyConstraint` element has the following properties:

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

912

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

917 The `Policy` element has the following properties:

- 918 • `type`: This attribute specifies the kind of policy (e.g. management practice) supported by an  
919 instance of the Node Type containing this element.
- 920 • `name`: This attribute defines the name of the policy. The name MUST be unique within a given  
921 Node Type containing the `Policy` element.

922 The `EnvironmentConstraint` element has the following properties:

- 923 • `constraintType`: The constraint type is specified by means of a URI, which defines both the  
924 semantic meaning of the constraint as well as the format of the constraint content.

925 The `RelationshipTemplate` element has the following properties:

- 926 • `id`: This attribute specifies the identifier of the Relationship Template. The identifier of the  
927 Relationship Template MUST be unique within the target namespace.
- 928 • `name`: This optional attribute specifies the name of the Relationship Template.
- 929 • `relationshipType`: The QName value of this property refers to the Relationship Type  
930 providing the type of the Relationship Template.
- 931 • `SourceElement`: The `id` attribute of this element references a Node Template or Group  
932 Template within the same Service Template document that is the source of the Relationship  
933 Template.



- 934 • `TargetElement`: The `id` attribute of this element references a Node Template or Group  
935 Template within the same Service Template document that is the target of the Relationship  
936 Template.
- 937 • `TargetElementReference`: The `id` attribute of this element refers by QName to an  
938 imported Node Template or Group Template that is the target of the Relationship Template. The  
939 referenced Node Template or Group Template will typically be the root node or root group of the  
940 corresponding Topology Template. In some cases a non-root Node Template or non-root Group  
941 Template might be referenced to support access to particular resources from a larger service, for  
942 example. Either `TargetElement` or `TargetElementReference` MUST be specified but  
943 not both.
- 944 • `PropertyDefaults`: Specifies initial values for one or more of the Relationship Type  
945 properties of the Relationship Type providing the property definitions in the concrete context of  
946 the Relationship Template.  
947 The initial values are specified by providing an instance document of the XML schema of the  
948 corresponding Relationship Type properties.  
949 The instance document of the XML schema might not validate against the existence constraints  
950 of the corresponding schema: not all relationship type properties might have an initial value  
951 assigned, i.e. mandatory elements or attributes might be missing in the instance provided by the  
952 Property Defaults element. Once the defined Relationship Template has been instantiated, any  
953 XML representation of the Relationship Type properties MUST validate according to the  
954 associated XML schema definition.
- 955 • `PropertyConstraints`: Specifies constraints on the use of one or more of the Relationship  
956 Type properties of the Relationship Type providing the property definitions for the Relationship  
957 Template.  
958 Each constraint is specified by means of a separate nested `PropertyConstraint` element.  
959 This element contains the actual encoding of the constraint.
- 960 • `RelationshipConstraints`: Specifies constraints on the use of the relationship.  
961 Each constraint is specified by means of a separate nested `RelationshipConstraint`  
962 element. This element can contain the actual encoding of the constraint, or its  
963 `constraintType` attribute already denotes the constraint itself. The constraint type is  
964 specified by means of a URI, which defines both the semantic meaning of the constraint as well  
965 as the format of any content.

966 The `GroupTemplate` element has the following properties:

- 967 • `id`: This attribute specifies the identifier of the Group Template. The identifier of the Group  
968 Template MUST be unique within the target namespace.
- 969 • `name`: This optional attribute specifies the name of the Group Template.
- 970 • `minInstances`: This integer attribute specifies the minimum number of instances to be created  
971 when instantiating the Group Template. The default value of this attribute is 1. The value of  
972 `minInstances` MUST NOT be less than 0.
- 973 • `maxInstances`: This attribute specifies the maximum number of instances that can be created  
974 when instantiating the Group Template. The default value of this attribute is 1. If the string is set  
975 to “unbounded”, an unbounded number of instances can be created. The value of  
976 `maxInstances` MUST be 1 or greater and MUST NOT be less than the value specified for  
977 `minInstances`.
- 978 • `NodeTemplate`: This is a node template contained within, or grouped by the Group Template.

- 979 • RelationshipTemplate: This is a relationship template contained within, or grouped by the  
980 Group.
- 981 • GroupTemplate: This is a Group Template of a nested group contained within, or grouped by  
982 the Group Template.
- 983 • Policies: Specifies policies of the Group Template. Each policy is specified by means of a  
984 separate nested Policy element. This element contains the actual policy specific content of the  
985 policy.

### 986 6.3 Example

987 The following Service Template defines a Topology Template in-place. The corresponding Topology  
988 Template contains two Node Templates called "MyApplication" and "MyAppServer". These Node  
989 Templates have the node types "Application" and "ApplicationServer", respectively, the definitions of  
990 which are imported by the Import element. The Node Template "MyApplication" is instantiated exactly  
991 once. Two of its Node Type Properties are initialized by a corresponding PropertyDefaults  
992 element. The Node Template "MyAppServer" can be instantiated as many times as needed. The  
993 "MyApplication" Node Template is connected with the "MyAppServer" Node Template via the Relationship  
994 Template named "MyDeploymentRelationship"; the behavior and semantics of the Relationship Template  
995 is defined in the Relationship Type "deployedOn" in the same Service Template document, saying that  
996 "MyApplication" is deployed on "MyAppServer". When instantiating the "SampleApplication" Topology  
997 Template, instances of "MyApplication" and "MyAppServer" are related by means of corresponding  
998 instances of "MyDeploymentRelationship".

```

999 <ServiceTemplate id="myService"
1000     name="My Service"
1001     targetNamespace="http://www.ibm.com/sample"
1002     xmlns:abc="http://www.ibm.com/sample">
1003
1004     <Import namespace="http://www.ibm.com/sample"
1005         importType="http://docs.oasis-open.org/tosca/ns/2011/12"/>
1006
1007     <TopologyTemplate id="SampleApplication">
1008
1009         <NodeTemplate id="MyApplication"
1010             name="My Application"
1011             nodeType="abc:Application">
1012             <PropertyDefaults>
1013                 <ApplicationProperties>
1014                     <Owner>Frank</Owner>
1015                     <InstanceName>Thomas' favorite application</InstanceName>
1016                 </ApplicationProperties>
1017             </PropertyDefaults>
1018         </NodeTemplate/>
1019
1020         <NodeTemplate id="MyAppServer"
1021             name="My Application Server"
1022             nodeType="abc:ApplicationServer"
1023             minInstances="0"
1024             maxInstances="unbounded"/>
1025
1026         <RelationshipTemplate id="MyDeploymentRelationship"
1027             relationshipType="deployedOn">
1028             <SourceElement id="MyApplication"/>
1029             <TargetElement id="MyAppServer"/>
1030         </RelationshipTemplate>
1031

```

```
1032 </TopologyTemplate>
1033
1034 </ServiceTemplate>
```

## 1035 7 Plans

1036 The operational management behavior of a Service Template is invoked by means of orchestration plans,  
1037 or more simply, *Plans*. Plans consist of individual steps (aka tasks or activities) to be performed and the  
1038 definition of the potential order of these steps. The execution of a step can be performed by one of the  
1039 functions offered via the interfaces of a Node Template, by invoking operations of a Service Template  
1040 API, or by invoking other operations being required in the context of a specific service. Plans are  
1041 classified by a type, and the following two plan types are defined as part of the TOSCA specification.  
1042 *Build plans* specify how instances of their associated Service Templates are made, and *termination plans*  
1043 specify how an instance of a Service Template is removed from the environment. Other plan types for  
1044 managing existing service instances throughout their life time are termed *modification plans*, and it is  
1045 expected that such plan types will be defined subsequently by authors of service templates and domain  
1046 expert groups.

### 1047 7.1 Syntax

```
1048 1 <Plans>
1049 2
1050 3   <Plan id="ID"
1051 4       name="string"?
1052 5       planType="anyURI"
1053 6       languageUsed="anyURI">+
1054 7
1055 8       <PreCondition expressionLanguage="anyURI">?
1056 9         condition
1057 10      </PreCondition>
1058 11
1059 12      ( <PlanModel>
1060 13        actual plan
1061 14      </PlanModel>
1062 15      |
1063 16      <PlanModelReference reference="anyURI"/>
1064 17    )
1065 18
1066 19   </Plan>
1067 20
1068 21 </Plans>
```

### 1069 7.2 Properties

1070 The `Plans` element contains one or more `Plan` elements which have the following properties:

- 1071 • `id`: This attribute specifies the identifier of the Plan. The identifier of the Plan MUST be unique  
1072 within the target namespace.
- 1073 • `name`: This optional attribute specifies the name of the Plan.
- 1074 • `planType`: The value of the attribute specifies the type of the plan as an indication on what the  
1075 effect of executing the plan on a service will have. The plan type is specified by means of a URI,  
1076 allowing for an extensibility mechanism for authors of service templates to define new plan types  
1077 over time.

1078 The following plan types are defined as part of the TOSCA specification.

- 1079           ○ <http://docs.oasis-open.org/tosca/ns/2011/12/PlanTypes/BuildPlan> - This URI defines the  
1080 *build plan* plan type for plans used to initially create a new instance of a service from a  
1081 Service Template.
- 1082           ○ <http://docs.oasis-open.org/tosca/ns/2011/12/PlanTypes/TerminationPlan> - This URI  
1083 defines the *termination plan* plan type for plans used to terminate the existence of a  
1084 service instance.

1085 Note that all other plan types for managing service instances throughout their life time will be  
1086 considered and referred to as *modification plans* in general.

- 1087 • `languageUsed`: This attribute denotes the process modeling language (or metamodel) used to  
1088 specify the plan. For example, "<http://www.omg.org/spec/BPMN/2.0/>" would specify that BPMN  
1089 2.0 has been used to model the plan.

- 1090 • `PreCondition`: This optional element specifies a condition that needs to be satisfied in order  
1091 for the plan to be executed. The `expressionLanguage` attribute of this element specifies the  
1092 expression language the nested condition is provided in.

1093 Typically, the precondition will be an expression in the instance state attribute of some of the  
1094 node templates or relationship templates of the topology template. It will be evaluated based on  
1095 the actual values of the corresponding attributes at the time the plan is requested to be executed.  
1096 Note, that any other kind of pre-condition is allowed.

- 1097 • `PlanModel`: This property contains the actual model content.

- 1098 • `PlanModelReference`: This property points to the model content. Its reference attribute  
1099 contains a URI of the model of the plan.

1100

1101 An instance of the `Plan` element MUST either contain the actual plan as instance of the  
1102 `PlanModel` element, or point to the model via the `PlanModelReference` element.

## 1103 7.3 Use of Process Modeling Languages

1104 TOSCA does not specify a separate metamodel for defining plans. Instead, it is assumed that a process  
1105 modelling language (a.k.a. metamodel) like BPEL [BPEL 2.0] or BPMN [BPMN 2.0] is used to define  
1106 plans. The specification favours the use of BPMN for modeling plans.

## 1107 7.4 Example

1108 The following defines two Plans, one Plan for creating a new instance of the "SampleApplication"  
1109 Topology Template (the plan is named "DeployApplication"), and one Plan for removing instances of  
1110 "SampleApplication". The Plan "DeployApplication" is a build plan specified in BPMN; the process model  
1111 is immediately included in the Plan Model (note that the BPMN model is incomplete but used to show the  
1112 mechanism of the `PlanModel` element). The Plan can only run when the PreCondition "Run only if  
1113 funding is available" is satisfied. The Plan "RemoveApplication" is a termination plan specified in BPEL;  
1114 the corresponding BPEL definition is defined elsewhere and only referenced by the  
1115 `PlanModelReference` element.

1116 <Plans>

1117

```
1118   <Plan id="DeployApplication"  
1119     name="Sample Application Build Plan"  
1120     planType=  
1121       "http://docs.oasis-open.org/tosca/ns/2011/12/PlanTypes/BuildPlan"  
1122     languageUsed="http://www.omg.org/spec/BPMN/2.0/">
```

1123

```
1124     <PreCondition expressionLanguage="www.my.com/text">?
```

```
1125       Run only if funding is available
```

```

1126     </PreCondition>
1127
1128     <PlanModel>
1129         <process name="DeployNewApplication" id="p1">
1130             <documentation>This process deploys a new instance of the
1131                 sample application.
1132             </documentation>
1133
1134             <task id="t1" name="CreateAccount"/>
1135
1136             <task id="t2" name="AcquireNetworkAddresses"
1137                 isSequential="false"
1138                 loopDataInput="t2Input.LoopCounter"/>
1139             <documentation>Assumption: t2 gets data of type "input"
1140                 as input and this data has a field names "LoopCounter"
1141                 that contains the actual multiplicity of the task.
1142             </documentation>
1143
1144             <task id="t3" name="DeployApplicationServer"
1145                 isSequential="false"
1146                 loopDataInput="t3Input.LoopCounter"/>
1147
1148             <task id="t4" name="DeployApplication"
1149                 isSequential="false"
1150                 loopDataInput="t4Input.LoopCounter"/>
1151
1152             <sequenceFlow id="s1" targetRef="t2" sourceRef="t1"/>
1153             <sequenceFlow id="s2" targetRef="t3" sourceRef="t2"/>
1154             <sequenceFlow id="s3" targetRef="t4" sourceRef="t3"/>
1155         </process>
1156     </PlanModel>
1157 </Plan>
1158
1159 <Plan id="RemoveApplication"
1160     planType="http://docs.oasis-
1161     open.org/tosca/ns/2011/12/PlanTypes/TerminationPlan"
1162     languageUsed=
1163     "http://docs.oasis-open.org/wsbpel/2.0/process/executable">
1164     <PlanModelReference reference="prj:RemoveApp"/>
1165 </Plan>
1166
1167 </Plans>

```

## 8 Security Considerations

1168 TOSCA does not mandate the use of any specific mechanism or technology for client authentication.  
1169 However, a client MUST provide a principal or the principal MUST be obtainable by the infrastructure.  
1170

---

1171 **9 Conformance**

1172 **This section is to be done.**

---

1173 **Appendix A. Portability and Interoperability**  
1174 **Considerations**

1175 This section illustrates the portability and interoperability aspects addressed by Service Templates:

1176 Portability - The ability to take Service Templates created in one vendor's environment and use them in  
1177 another vendor's environment.

1178 Interoperability - The capability for multiple components (e.g. a task of a plan and the definition of a  
1179 topology node) to interact using well-defined messages and protocols. This enables combining  
1180 components from different vendors allowing seamless management of services.

1181 Portability demands support of TOSCA artifacts.

## Appendix B. Complete TOSCA Grammar

```

1183 1 <ServiceTemplate id="ID"
1184 2     name="string"?
1185 3     targetNamespace="anyURI">
1186 4
1187 5 <Extensions>?
1188 6 <Extension namespace="anyURI"
1189 7     mustUnderstand="yes|no"?/>+
1190 8 </Extensions>
1191 9
1192 10 <Import namespace="anyURI"?
1193 11     location="anyURI"?
1194 12     importType="anyURI"/>*
1195 13
1196 14 <Types>?
1197 15 <xs:schema .../>*
1198 16 </Types>
1199 17
1200 18 (
1201 19 <TopologyTemplateReference reference="QName"/>
1202 20 |
1203 21 <TopologyTemplate id="ID"
1204 22     name="string"?>
1205 23
1206 24 (
1207 25 <NodeTemplate id="ID"
1208 26     name="string"?
1209 27     nodeType="QName"
1210 28     minInstances="int"?
1211 29     maxInstances="int|string"?>
1212 30
1213 31 <PropertyDefaults>?
1214 32     XML fragment
1215 33 </PropertyDefaults>
1216 34
1217 35 <PropertyConstraints>?
1218 36
1219 37 <PropertyConstraint property="string"
1220 38     constraintType="anyURI">+
1221 39     constraint?
1222 40 </PropertyConstraint>
1223 41
1224 42 </PropertyConstraints>
1225 43
1226 44 <Policies>?
1227 45 <Policy name="string" type="anyURI">+
1228 46     policy specific content
1229 47 </Policy>
1230 48 </Policies>
1231 49
1232 50 <EnvironmentConstraints>?
1233 51 <EnvironmentConstraint constraintType="anyURI">+
1234 52     constraint type specific content?
1235 53 </EnvironmentConstraint>

```



```

1236 54     </EnvironmentConstraints>
1237 55
1238 56     <DeploymentArtifacts>?
1239 57         <DeploymentArtifact name="string" type="anyURI">+
1240 58             artifact specific content
1241 59         </DeploymentArtifact>
1242 60     </DeploymentArtifacts>
1243 61
1244 62 </NodeTemplate>
1245 63 |
1246 64 <RelationshipTemplate id="ID"
1247 65     name="string"?
1248 66     relationshipType="QName">+
1249 67
1250 68     <SourceElement id="IDREF"/>
1251 69
1252 70     ( <TargetElement id="IDREF"/>
1253 71     |
1254 72     <TargetElementReference id="QName"/>
1255 73     )
1256 74
1257 75     <PropertyDefaults>?
1258 76         XML fragment
1259 77     </PropertyDefaults>
1260 78
1261 79     <PropertyConstraints>?
1262 80
1263 81     <PropertyConstraint property="string"
1264 82         constraintType="anyURI">+
1265 83         constraint?
1266 84     </PropertyConstraint>
1267 85
1268 86 </PropertyConstraints>
1269 87
1270 88     <RelationshipConstraints>?
1271 89
1272 90     <RelationshipConstraint constraintType="anyURI">+
1273 91         constraint?
1274 92     </RelationshipConstraint>
1275 93
1276 94     </RelationshipConstraints>
1277 95
1278 96 </RelationshipTemplate>
1279 97 |
1280 98 <GroupTemplate id="ID"
1281 99     name="string"?
1282 100     minInstances="int"?
1283 101     maxInstances="int|string"?>
1284 102
1285 103     (
1286 104         <NodeTemplate ... />
1287 105     |
1288 106         <RelationshipTemplate ... />
1289 107     |
1290 108         <GroupTemplate ... />
1291 109     )+
1292 110
1293 111 <Policies>?

```

```

1294 112     <Policy name="string" type="anyURI">+
1295 113         policy specific content
1296 114     </Policy>
1297 115 </Policies>
1298 116
1299 117     </GroupTemplate>
1300 118 )+
1301 119
1302 120 </TopologyTemplate>
1303 121 )?
1304 122
1305 123 <NodeTypes>?
1306 124
1307 125     <NodeType id="ID"
1308 126         name="string"?>+
1309 127
1310 128     <NodeTypeProperties element="QName"?
1311 129         type="QName"?/>?
1312 130
1313 131     <DerivedFrom nodeTypeRef="QName"/>?
1314 132
1315 133     <InstanceStates>?
1316 134         <InstanceState state="anyURI">+
1317 135     </InstanceStates>
1318 136
1319 137     <Interfaces>?
1320 138
1321 139     <Interface>+
1322 140
1323 141     (
1324 142     <WSDL portType="QName"
1325 143         operation="NCName"?>+
1326 144     |
1327 145     <REST method="GET | PUT | POST | DELETE"
1328 146         requestURI="anyURI"
1329 147         requestPayload="QName"?
1330 148         responsePayload="QName"?>+
1331 149     |
1332 150     <Operation name="NCame">+
1333 151
1334 152     <InputParameters>?
1335 153
1336 154         <InputParamter name="string"
1337 155             type="string"
1338 156             required="yes|no">+
1339 157
1340 158     </InputParameters>
1341 159
1342 160     <OutputParameters>?
1343 161
1344 162         <OutputParamter name="string"
1345 163             type="string"
1346 164             required="yes|no">+
1347 165
1348 166     </OutputParameters>
1349 167
1350 168     <Implementations>
1351 169

```

```

1352 170         <Implementation implementationID="anyURI"?
1353 171             language="anyURI"?>+
1354 172         (
1355 173             <ImplementationProper>?
1356 174                 code
1357 175             </ImplementationProper>
1358 176             |
1359 177             <ImplementationReference ref="anyURI"/>?
1360 178         )
1361 179         <Implementation>
1362 180
1363 181     </Implementations>
1364 182 </Operation>
1365 183 )
1366 184
1367 185 </Interface>
1368 186
1369 187 </Interfaces>
1370 188
1371 189 <DeploymentArtifacts>?
1372 190     <DeploymentArtifact name="string" type="anyURI">+
1373 191         artifact specific content
1374 192     </DeploymentArtifact>
1375 193 </DeploymentArtifacts>
1376 194
1377 195
1378 196 <Policies>?
1379 197
1380 198     <Policy name="string" type="anyURI">+
1381 199         policy specific content
1382 200     </Policy>
1383 201
1384 202 </Policies>
1385 203
1386 204 </NodeType>
1387 205
1388 206 </NodeTypes>
1389 207
1390 208 <RelationshipTypes>?
1391 209
1392 210     <RelationshipType id="ID"
1393 211         name="string"?
1394 212         semantics="anyURI"
1395 213         cascadingDeletion="yes|no"?>+
1396 214
1397 215     <RelationshipTypeProperties element="QName"?
1398 216         type="QName"?/>?
1399 217
1400 218     <InstanceStates>?
1401 219         <InstanceState state="anyURI">+
1402 220     </InstanceStates>
1403 221
1404 222     </RelationshipType>
1405 223
1406 224 </RelationshipTypes>
1407 225
1408 226 <Plans>?
1409 227

```

```
1410 228     <Plan id="ID"
1411 229         name="string"?
1412 230         planType="anyURI"
1413 231         languageUsed="anyURI">+
1414 232
1415 233     <PreCondition expressionLanguage="anyURI">?
1416 234         condition
1417 235     </PreCondition>
1418 236
1419 237     ( <PlanModel>
1420 238         actual plan
1421 239     </PlanModel>
1422 240     |
1423 241     <PlanModelReference reference="anyURI"/>
1424 242     )
1425 243
1426 244     </Plan>
1427 245
1428 246     </Plans>
1429 247
1430 248 </ServiceTemplate>
```

## Appendix C. TOSCA Schema

```
1432 1 <?xml version="1.0" encoding="UTF-8"?>
1433 2 <xs:schema targetNamespace="http://docs.oasis-open.org/tosca/ns/2011/12"
1434 3   elementFormDefault="qualified" attributeFormDefault="unqualified"
1435 4   xmlns="http://docs.oasis-open.org/tosca/ns/2011/12"
1436 5   xmlns:xs="http://www.w3.org/2001/XMLSchema">
1437 6
1438 7   <xs:import namespace="http://www.w3.org/XML/1998/namespace"
1439 8     schemaLocation="http://www.w3.org/2001/xml.xsd"/>
1440 9
1441 10  <xs:element name="documentation" type="tDocumentation"/>
1442 11  <xs:complexType name="tDocumentation" mixed="true">
1443 12    <xs:sequence>
1444 13      <xs:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
1445 14    </xs:sequence>
1446 15    <xs:attribute name="source" type="xs:anyURI"/>
1447 16    <xs:attribute ref="xml:lang"/>
1448 17  </xs:complexType>
1449 18
1450 19  <xs:complexType name="tExtensibleElements">
1451 20    <xs:sequence>
1452 21      <xs:element ref="documentation" minOccurs="0" maxOccurs="unbounded"/>
1453 22      <xs:any namespace="##other" processContents="lax" minOccurs="0"
1454 23        maxOccurs="unbounded"/>
1455 24    </xs:sequence>
1456 25    <xs:anyAttribute namespace="##other" processContents="lax"/>
1457 26  </xs:complexType>
1458 27
1459 28  <xs:complexType name="tImport">
1460 29    <xs:complexContent>
1461 30      <xs:extension base="tExtensibleElements">
1462 31        <xs:attribute name="namespace" type="xs:anyURI"/>
1463 32        <xs:attribute name="location" type="xs:anyURI"/>
1464 33        <xs:attribute name="importType" type="importedURI" use="required"/>
1465 34      </xs:extension>
1466 35    </xs:complexContent>
1467 36  </xs:complexType>
1468 37
1469 38  <xs:element name="ServiceTemplate">
1470 39    <xs:complexType>
1471 40      <xs:complexContent>
1472 41        <xs:extension base="tServiceTemplate"/>
1473 42      </xs:complexContent>
1474 43    </xs:complexType>
1475 44  </xs:element>
1476 45
1477 46  <xs:complexType name="tServiceTemplate">
1478 47    <xs:complexContent>
1479 48      <xs:extension base="tExtensibleElements">
1480 49        <xs:sequence>
1481 50          <xs:element name="Import" type="tImport" minOccurs="0"
1482 51            maxOccurs="unbounded"/>
1483 52          <xs:element name="Types" minOccurs="0">
1484 53            <xs:complexType>
```

```

1485 54     <xs:sequence>
1486 55     <xs:any namespace="##other" processContents="lax" minOccurs="0"
1487 56         maxOccurs="unbounded"/>
1488 57     </xs:sequence>
1489 58 </xs:complexType>
1490 59 </xs:element>
1491 60 <xs:element name="Extensions" minOccurs="0">
1492 61     <xs:complexType>
1493 62         <xs:sequence>
1494 63             <xs:element name="Extension" type="tExtension"
1495 64                 maxOccurs="unbounded"/>
1496 65         </xs:sequence>
1497 66     </xs:complexType>
1498 67 </xs:element>
1499 68 <xs:choice minOccurs="0">
1500 69     <xs:element name="TopologyTemplateReference">
1501 70         <xs:complexType>
1502 71             <xs:attribute name="reference" type="xs:QName"/>
1503 72         </xs:complexType>
1504 73     </xs:element>
1505 74     <xs:element name="TopologyTemplate" type="tTopologyTemplate"/>
1506 75 </xs:choice>
1507 76 <xs:element name="NodeTypes" type="tNodeTypes" minOccurs="0"/>
1508 77 <xs:element name="RelationshipTypes" type="tRelationshipTypes"
1509 78     minOccurs="0"/>
1510 79 <xs:element name="Plans" type="tPlans" minOccurs="0"/>
1511 80 </xs:sequence>
1512 81 <xs:attribute name="id" type="xs:ID" use="required"/>
1513 82 <xs:attribute name="name" type="xs:string" use="optional"/>
1514 83 <xs:attribute name="targetNamespace" type="xs:anyURI"/>
1515 84 </xs:extension>
1516 85 </xs:complexContent>
1517 86 </xs:complexType>
1518 87
1519 88 <xs:complexType name="tDeploymentArtifact">
1520 89     <xs:complexContent>
1521 90         <xs:extension base="tExtensibleElements">
1522 91             <xs:attribute name="name" type="xs:string" use="required"/>
1523 92             <xs:attribute name="type" type="xs:anyURI" use="required"/>
1524 93         </xs:extension>
1525 94     </xs:complexContent>
1526 95 </xs:complexType>
1527 96
1528 97 <xs:element name="NodeTemplate" type="tNodeTemplate"/>
1529 98 <xs:complexType name="tNodeTemplate">
1530 99     <xs:complexContent>
1531 100         <xs:extension base="tExtensibleElements">
1532 101             <xs:sequence>
1533 102                 <xs:element name="PropertyDefaults" minOccurs="0">
1534 103                     <xs:complexType>
1535 104                         <xs:sequence>
1536 105                             <xs:any namespace="##other" processContents="lax"/>
1537 106                         </xs:sequence>
1538 107                     </xs:complexType>
1539 108                 </xs:element>
1540 109                 <xs:element name="PropertyConstraints" minOccurs="0">
1541 110                     <xs:complexType>
1542 111                         <xs:sequence>

```

```

1543 112     <xs:element name="PropertyConstraint"
1544 113         type="tPropertyConstraint" maxOccurs="unbounded"/>
1545 114     </xs:sequence>
1546 115     </xs:complexType>
1547 116 </xs:element>
1548 117 <xs:element name="Policies" minOccurs="0">
1549 118     <xs:complexType>
1550 119         <xs:sequence>
1551 120             <xs:element name="Policy" type="tPolicy"
1552 121                 maxOccurs="unbounded"/>
1553 122         </xs:sequence>
1554 123     </xs:complexType>
1555 124 </xs:element>
1556 125 <xs:element name="DeploymentArtifacts" minOccurs="0">
1557 126     <xs:complexType>
1558 127         <xs:sequence>
1559 128             <xs:element name="DeploymentArtifact"
1560 129                 type="tDeploymentArtifact" maxOccurs="unbounded"/>
1561 130         </xs:sequence>
1562 131     </xs:complexType>
1563 132 </xs:element>
1564 133 <xs:element name="EnvironmentConstraints" minOccurs="0">
1565 134     <xs:complexType>
1566 135         <xs:sequence>
1567 136             <xs:element name="EnvironmentConstraint"
1568 137                 type="tEnvironmentConstraint" maxOccurs="unbounded"/>
1569 138         </xs:sequence>
1570 139     </xs:complexType>
1571 140 </xs:element>
1572 141 </xs:sequence>
1573 142 <xs:attribute name="id" type="xs:ID" use="required"/>
1574 143 <xs:attribute name="name" type="xs:string" use="optional"/>
1575 144 <xs:attribute name="nodeType" type="xs:QName" use="required"/>
1576 145 <xs:attribute name="minInstances" type="xs:int" use="optional"
1577 146     default="1"/>
1578 147 <xs:attribute name="maxInstances" use="optional" default="1">
1579 148     <xs:simpleType>
1580 149         <xs:union>
1581 150             <xs:simpleType>
1582 151                 <xs:restriction base="xs:nonNegativeInteger">
1583 152                     <xs:pattern value="([1-9]+[0-9]*)"/>
1584 153                 </xs:restriction>
1585 154             </xs:simpleType>
1586 155             <xs:simpleType>
1587 156                 <xs:restriction base="xs:string">
1588 157                     <xs:enumeration value="unbounded"/>
1589 158                 </xs:restriction>
1590 159             </xs:simpleType>
1591 160         </xs:union>
1592 161     </xs:simpleType>
1593 162 </xs:attribute>
1594 163 </xs:extension>
1595 164 </xs:complexContent>
1596 165 </xs:complexType>
1597 166
1598 167 <xs:complexType name="tPropertyConstraint">
1599 168     <xs:sequence>
1600 169 <xs:any namespace="##other" processContents="lax" minOccurs="0"/>

```

```

1601 170     </xs:sequence>
1602 171     <xs:attribute name="property" type="xs:string" use="required"/>
1603 172     <xs:attribute name="constraintType" type="xs:anyURI" use="required"/>
1604 173 </xs:complexType>
1605 174
1606 175 <xs:element name="TopologyTemplate" type="tTopologyTemplate"/>
1607 176 <xs:complexType name="tTopologyTemplate">
1608 177   <xs:complexContent>
1609 178     <xs:extension base="tTopologyElementCollection"/>
1610 179   </xs:complexContent>
1611 180 </xs:complexType>
1612 181
1613 182 <xs:element name="GroupTemplate" type="tGroupTemplate"/>
1614 183 <xs:complexType name="tGroupTemplate">
1615 184   <xs:complexContent>
1616 185     <xs:extension base="tTopologyElementCollection">
1617 186       <xs:sequence>
1618 187         <xs:element name="Policies" minOccurs="0">
1619 188           <xs:complexType>
1620 189             <xs:sequence>
1621 190               <xs:element name="Policy" type="tPolicy"
1622 191                 maxOccurs="unbounded"/>
1623 192             </xs:sequence>
1624 193           </xs:complexType>
1625 194         </xs:element>
1626 195       </xs:sequence>
1627 196       <xs:attribute name="minInstances" type="xs:int" use="optional"
1628 197         default="1"/>
1629 198       <xs:attribute name="maxInstances" use="optional" default="1">
1630 199         <xs:simpleType>
1631 200           <xs:union>
1632 201             <xs:simpleType>
1633 202               <xs:restriction base="xs:nonNegativeInteger">
1634 203                 <xs:pattern value="([1-9]+[0-9]*)"/>
1635 204               </xs:restriction>
1636 205             </xs:simpleType>
1637 206             <xs:simpleType>
1638 207               <xs:restriction base="xs:string">
1639 208                 <xs:enumeration value="unbounded"/>
1640 209               </xs:restriction>
1641 210             </xs:simpleType>
1642 211           </xs:union>
1643 212         </xs:simpleType>
1644 213       </xs:attribute>
1645 214     </xs:extension>
1646 215   </xs:complexContent>
1647 216 </xs:complexType>
1648 217
1649 218 <xs:complexType name="tTopologyElementCollection">
1650 219   <xs:complexContent>
1651 220     <xs:extension base="tExtensibleElements">
1652 221       <xs:choice maxOccurs="unbounded">
1653 222         <xs:element name="NodeTemplate" type="tNodeTemplate"/>
1654 223         <xs:element name="RelationshipTemplate"
1655 224           type="tRelationshipTemplate"/>
1656 225         <xs:element name="GroupTemplate" type="tGroupTemplate"/>
1657 226       </xs:choice>
1658 227     <xs:attribute name="id" type="xs:ID" use="required"/>

```



```

1659 228     <xs:attribute name="name" type="xs:string" use="optional"/>
1660 229     <xs:attribute name="targetNamespace" type="xs:anyURI"/>
1661 230     </xs:extension>
1662 231     </xs:complexContent>
1663 232 </xs:complexType>
1664 233
1665 234 <xs:element name="RelationshipTypes" type="tRelationshipTypes"/>
1666 235 <xs:complexType name="tRelationshipTypes">
1667 236   <xs:sequence>
1668 237     <xs:element name="RelationshipType" type="tRelationshipType"
1669 238       maxOccurs="unbounded"/>
1670 239   </xs:sequence>
1671 240   <xs:attribute name="targetNamespace" type="xs:anyURI"/>
1672 241 </xs:complexType>
1673 242
1674 243 <xs:element name="RelationshipType" type="tRelationshipType"/>
1675 244 <xs:complexType name="tRelationshipType">
1676 245   <xs:complexContent>
1677 246     <xs:extension base="tExtensibleElements">
1678 247       <xs:sequence>
1679 248         <xs:element name="RelationshipTypeProperties" minOccurs="0">
1680 249           <xs:complexType>
1681 250             <xs:attribute name="element" type="xs:QName"/>
1682 251             <xs:attribute name="type" type="xs:QName"/>
1683 252           </xs:complexType>
1684 253         </xs:element>
1685 254         <xs:element name="InstanceStates"
1686 255           type="tTopologyElementInstanceStates" minOccurs="0"/>
1687 256       </xs:sequence>
1688 257       <xs:attribute name="id" type="xs:ID" use="required"/>
1689 258       <xs:attribute name="name" type="xs:string" use="optional"/>
1690 259       <xs:attribute name="semantics" type="xs:anyURI" use="required"/>
1691 260       <xs:attribute name="cascadingDeletion" type="tBoolean"
1692 261         use="optional" default="no"/>
1693 262       <xs:attribute name="targetNamespace" type="xs:anyURI"/>
1694 263     </xs:extension>
1695 264   </xs:complexContent>
1696 265 </xs:complexType>
1697 266
1698 267 <xs:element name="RelationshipTemplate" type="tRelationshipTemplate"/>
1699 268 <xs:complexType name="tRelationshipTemplate">
1700 269   <xs:complexContent>
1701 270     <xs:extension base="tExtensibleElements">
1702 271       <xs:sequence>
1703 272         <xs:element name="SourceElement">
1704 273           <xs:complexType>
1705 274             <xs:attribute name="id" type="xs:IDREF" use="required"/>
1706 275           </xs:complexType>
1707 276         </xs:element>
1708 277         <xs:choice>
1709 278           <xs:element name="TargetElement">
1710 279             <xs:complexType>
1711 280               <xs:attribute name="id" type="xs:IDREF" use="required"/>
1712 281             </xs:complexType>
1713 282           </xs:element>
1714 283           <xs:element name="TargetElementReference">
1715 284             <xs:complexType>
1716 285               <xs:attribute name="id" type="xs:QName" use="required"/>

```

```

1717 286     </xs:complexType>
1718 287     </xs:element>
1719 288 </xs:choice>
1720 289 <xs:element name="PropertyDefaults" minOccurs="0">
1721 290   <xs:complexType>
1722 291     <xs:sequence>
1723 292       <xs:any namespace="##other" processContents="lax"/>
1724 293     </xs:sequence>
1725 294   </xs:complexType>
1726 295 </xs:element>
1727 296 <xs:element name="PropertyConstraints" minOccurs="0">
1728 297   <xs:complexType>
1729 298     <xs:sequence>
1730 299       <xs:element name="PropertyConstraint"
1731 300         type="tPropertyConstraint" maxOccurs="unbounded"/>
1732 301     </xs:sequence>
1733 302   </xs:complexType>
1734 303 </xs:element>
1735 304 <xs:element name="RelationshipConstraints" minOccurs="0">
1736 305   <xs:complexType>
1737 306     <xs:sequence>
1738 307       <xs:element name="RelationshipConstraint"
1739 308         maxOccurs="unbounded">
1740 309         <xs:complexType>
1741 310           <xs:sequence>
1742 311             <xs:any namespace="##other" processContents="lax"
1743 312               minOccurs="0"/>
1744 313           </xs:sequence>
1745 314           <xs:attribute name="constraintType" type="xs:anyURI"
1746 315             use="required"/>
1747 316         </xs:complexType>
1748 317       </xs:element>
1749 318     </xs:sequence>
1750 319   </xs:complexType>
1751 320 </xs:element>
1752 321 </xs:sequence>
1753 322 <xs:attribute name="id" type="xs:ID" use="required"/>
1754 323 <xs:attribute name="name" type="xs:string" use="optional"/>
1755 324 <xs:attribute name="relationshipType" type="xs:QName"
1756 325   use="required"/>
1757 326 </xs:extension>
1758 327 </xs:complexContent>
1759 328 </xs:complexType>
1760 329
1761 330 <xs:element name="NodeTypes" type="tNodeTypes"/>
1762 331 <xs:complexType name="tNodeTypes">
1763 332   <xs:sequence>
1764 333     <xs:element name="NodeType" type="tNodeType" maxOccurs="unbounded"/>
1765 334   </xs:sequence>
1766 335   <xs:attribute name="targetNamespace" type="xs:anyURI"/>
1767 336 </xs:complexType>
1768 337
1769 338 <xs:element name="NodeType" type="tNodeType"/>
1770 339 <xs:complexType name="tNodeType">
1771 340   <xs:complexContent>
1772 341     <xs:extension base="tExtensibleElements">
1773 342       <xs:sequence>
1774 343         <xs:element name="NodeTypeProperties" minOccurs="0">

```

```

1775 344     <xs:complexType>
1776 345     <xs:attribute name="element" type="xs:QName"/>
1777 346     <xs:attribute name="type" type="xs:QName"/>
1778 347     </xs:complexType>
1779 348 </xs:element>
1780 349 <xs:element name="DerivedFrom" minOccurs="0">
1781 350   <xs:complexType>
1782 351     <xs:attribute name="nodeTypeRef" type="xs:QName"
1783 352       use="required"/>
1784 353   </xs:complexType>
1785 354 </xs:element>
1786 355 <xs:element name="InstanceStates"
1787 356   type="tTopologyElementInstanceStates" minOccurs="0"/>
1788 357 <xs:element name="Interfaces" minOccurs="0">
1789 358   <xs:complexType>
1790 359     <xs:sequence>
1791 360       <xs:element name="Interface" maxOccurs="unbounded">
1792 361         <xs:complexType>
1793 362           <xs:choice>
1794 363             <xs:element name="WSDL" type="tWSDL" maxOccurs="unbounded"/>
1795 364             <xs:element name="REST" type="tREST" maxOccurs="unbounded"/>
1796 365             <xs:element name="Operation" maxOccurs="unbounded">
1797 366               <xs:complexType>
1798 367                 <xs:complexContent>
1799 368                   <xs:extension base="tOperation"/>
1800 369                 </xs:complexContent>
1801 370               </xs:complexType>
1802 371             </xs:element>
1803 372           </xs:choice>
1804 373         </xs:complexType>
1805 374       </xs:element>
1806 375     </xs:sequence>
1807 376   </xs:complexType>
1808 377 </xs:element>
1809 378 <xs:element name="Policies" minOccurs="0">
1810 379   <xs:complexType>
1811 380     <xs:sequence>
1812 381       <xs:element name="Policy" type="tPolicy"
1813 382         maxOccurs="unbounded"/>
1814 383     </xs:sequence>
1815 384   </xs:complexType>
1816 385 </xs:element>
1817 386 <xs:element name="DeploymentArtifacts" minOccurs="0">
1818 387   <xs:complexType>
1819 388     <xs:sequence>
1820 389       <xs:element name="DeploymentArtifact"
1821 390         type="tDeploymentArtifact" maxOccurs="unbounded"/>
1822 391     </xs:sequence>
1823 392   </xs:complexType>
1824 393 </xs:element>
1825 394 </xs:sequence>
1826 395   <xs:attribute name="id" type="xs:ID" use="required"/>
1827 396   <xs:attribute name="name" type="xs:string" use="optional"/>
1828 397   <xs:attribute name="targetNamespace" type="xs:anyURI"/>
1829 398 </xs:extension>
1830 399 </xs:complexContent>
1831 400 </xs:complexType>
1832 401

```

```

1833 402 <xs:element name="Plans" type="tPlans"/>
1834 403 <xs:complexType name="tPlans">
1835 404   <xs:sequence>
1836 405     <xs:element name="Plan" type="tPlan" maxOccurs="unbounded"/>
1837 406   </xs:sequence>
1838 407   <xs:attribute name="targetNamespace" type="xs:anyURI"/>
1839 408 </xs:complexType>
1840 409
1841 410 <xs:element name="Plan" type="tPlan"/>
1842 411 <xs:complexType name="tPlan">
1843 412   <xs:complexContent>
1844 413     <xs:extension base="tExtensibleElements">
1845 414       <xs:sequence>
1846 415         <xs:element name="Precondition" type="tCondition" minOccurs="0"/>
1847 416         <xs:choice>
1848 417           <xs:element name="PlanModel">
1849 418             <xs:complexType>
1850 419               <xs:sequence>
1851 420                 <xs:any namespace="##other" processContents="lax"/>
1852 421               </xs:sequence>
1853 422             </xs:complexType>
1854 423           </xs:element>
1855 424           <xs:element name="PlanModelReference">
1856 425             <xs:complexType>
1857 426               <xs:attribute name="reference" type="xs:anyURI"
1858 427                 use="required"/>
1859 428             </xs:complexType>
1860 429           </xs:element>
1861 430         </xs:choice>
1862 431       </xs:sequence>
1863 432       <xs:attribute name="id" type="xs:ID" use="required"/>
1864 433       <xs:attribute name="name" type="xs:string" use="optional"/>
1865 434       <xs:attribute name="planType" type="xs:anyURI" use="required"/>
1866 435       <xs:attribute name="languageUsed" type="xs:anyURI" use="required"/>
1867 436     </xs:extension>
1868 437   </xs:complexContent>
1869 438 </xs:complexType>
1870 439
1871 440 <xs:complexType name="tPolicy">
1872 441   <xs:complexContent>
1873 442     <xs:extension base="tExtensibleElements">
1874 443       <xs:attribute name="name" type="xs:string" use="required"/>
1875 444       <xs:attribute name="type" type="xs:anyURI" use="required"/>
1876 445     </xs:extension>
1877 446   </xs:complexContent>
1878 447 </xs:complexType>
1879 448
1880 449 <xs:complexType name="tEnvironmentConstraint">
1881 450   <xs:sequence>
1882 451     <xs:any namespace="##other" processContents="lax"/>
1883 452   </xs:sequence>
1884 453   <xs:attribute name="constraintType" type="xs:anyURI" use="required"/>
1885 454 </xs:complexType>
1886 455
1887 456 <xs:complexType name="tExtensions">
1888 457   <xs:complexContent>
1889 458     <xs:extension base="tExtensibleElements">
1890 459       <xs:sequence>

```

```

1891 460     <xs:element name="Extension" type="tExtension"
1892 461         maxOccurs="unbounded"/>
1893 462     </xs:sequence>
1894 463 </xs:extension>
1895 464 </xs:complexContent>
1896 465 </xs:complexType>
1897 466
1898 467 <xs:complexType name="tExtension">
1899 468     <xs:complexContent>
1900 469         <xs:extension base="tExtensibleElements">
1901 470             <xs:attribute name="namespace" type="xs:anyURI" use="required"/>
1902 471             <xs:attribute name="mustUnderstand" type="tBoolean" use="optional"
1903 472                 default="yes"/>
1904 473         </xs:extension>
1905 474     </xs:complexContent>
1906 475 </xs:complexType>
1907 476
1908 477 <xs:complexType name="tParameter">
1909 478     <xs:attribute name="name" type="xs:string" use="required"/>
1910 479     <xs:attribute name="type" type="xs:string" use="required"/>
1911 480     <xs:attribute name="required" type="tBoolean" use="optional"
1912 481         default="yes"/>
1913 482 </xs:complexType>
1914 483
1915 484 <xs:complexType name="tWSDL">
1916 485     <xs:attribute name="portType" type="xs:QName" use="required"/>
1917 486     <xs:attribute name="operation" type="xs:NCName" use="optional"/>
1918 487 </xs:complexType>
1919 488
1920 489 <xs:complexType name="tOperation">
1921 490     <xs:complexContent>
1922 491         <xs:extension base="tExtensibleElements">
1923 492             <xs:sequence>
1924 493                 <xs:element name="Implementations">
1925 494                     <xs:complexType>
1926 495                         <xs:sequence>
1927 496                             <xs:element name="Implementation" maxOccurs="unbounded">
1928 497                                 <xs:complexType>
1929 498                                     <xs:choice>
1930 499                                         <xs:element name="ImplementationProper" type="xs:anyType"
1931 500                                             minOccurs="0"/>
1932 501                                         <xs:element name="ImplementationReference" minOccurs="0">
1933 502                                             <xs:complexType>
1934 503                                                 <xs:attribute name="ref" type="xs:anyURI"/>
1935 504                                             </xs:complexType>
1936 505                                         </xs:element>
1937 506                                     </xs:choice>
1938 507                                         <xs:attribute name="implementationID" type="xs:anyURI"/>
1939 508                                         <xs:attribute name="language" type="xs:anyURI"/>
1940 509                                     </xs:complexType>
1941 510                                 </xs:element>
1942 511                             </xs:sequence>
1943 512                         </xs:complexType>
1944 513                     </xs:element>
1945 514                 <xs:element name="InputParameters" minOccurs="0">
1946 515                     <xs:complexType>
1947 516                         <xs:sequence>
1948 517                             <xs:element name="InputParameter" type="tParameter"

```

```

1949 518         maxOccurs="unbounded"/>
1950 519         </xs:sequence>
1951 520     </xs:complexType>
1952 521 </xs:element>
1953 522 <xs:element name="OutputParameters" minOccurs="0">
1954 523     <xs:complexType>
1955 524         <xs:sequence>
1956 525             <xs:element name="OutputParameter" type="tParameter"
1957 526                 maxOccurs="unbounded"/>
1958 527         </xs:sequence>
1959 528     </xs:complexType>
1960 529 </xs:element>
1961 530 </xs:sequence>
1962 531     <xs:attribute name="name" type="xs:NCName" use="required"/>
1963 532 </xs:extension>
1964 533 </xs:complexContent>
1965 534 </xs:complexType>
1966 535
1967 536 <xs:complexType name="tREST">
1968 537     <xs:attribute name="method" default="GET">
1969 538         <xs:simpleType>
1970 539             <xs:restriction base="xs:string">
1971 540                 <xs:enumeration value="GET"/>
1972 541                 <xs:enumeration value="PUT"/>
1973 542                 <xs:enumeration value="POST"/>
1974 543                 <xs:enumeration value="DELETE"/>
1975 544             </xs:restriction>
1976 545         </xs:simpleType>
1977 546     </xs:attribute>
1978 547     <xs:attribute name="requestURI" type="xs:anyURI" use="required"/>
1979 548     <xs:attribute name="requestPayload" type="xs:QName"/>
1980 549     <xs:attribute name="responsePayload" type="xs:QName"/>
1981 550 </xs:complexType>
1982 551
1983 552 <xs:complexType name="tCondition">
1984 553     <xs:sequence>
1985 554         <xs:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
1986 555     </xs:sequence>
1987 556     <xs:attribute name="expressionLanguage" type="xs:anyURI"
1988 557         use="required"/>
1989 558 </xs:complexType>
1990 559
1991 560 <xs:complexType name="tTopologyElementInstanceStates">
1992 561     <xs:sequence>
1993 562         <xs:element name="InstanceState" maxOccurs="unbounded">
1994 563             <xs:complexType>
1995 564                 <xs:attribute name="state" type="xs:anyURI" use="required"/>
1996 565             </xs:complexType>
1997 566         </xs:element>
1998 567     </xs:sequence>
1999 568 </xs:complexType>
2000 569
2001 570 <xs:simpleType name="tBoolean">
2002 571     <xs:restriction base="xs:string">
2003 572         <xs:enumeration value="yes"/>
2004 573         <xs:enumeration value="no"/>
2005 574     </xs:restriction>
2006 575 </xs:simpleType>

```

```
2007 576
2008 577 <xs:simpleType name="importedURI">
2009 578   <xs:restriction base="xs:anyURI"/>
2010 579 </xs:simpleType>
2011 580
2012 581 </xs:schema>
```

2013

## Appendix D. Sample

2014

This appendix contains the full sample used in this specification.

2015

### D.1 Sample Service Topology Definition

2016

```
<ServiceTemplate name="myService"
  targetNamespace="http://www.ibm.com/sample">
```

2017

2018

2019

```
<Types>
```

2020

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

2021

```
  elementFormDefault="qualified"
```

2022

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

2023

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

2024

```
<xs:complexType>
```

2025

```
<xs:sequence>
```

2026

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

2027

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

2028

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

2029

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

2030

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

2031

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

2032

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

2033

```
<xs:complexType>
```

2034

```
<xs:sequence>
```

2035

```
<element name="HostName" type="string"/>
```

2036

```
<element name="IPAddress" type="string"/>
```

2037

```
<element name="HeapSize" type="positiveInteger"/>
```

2038

```
<element name="SoapPort" type="positiveInteger"/>
```

2039

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

2040

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

2041

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

2042

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

2043

```
</Types>
```

2044

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

2045

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

2046

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

2047

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

2048

```
<PropertyDefaults>
```

2049

```
<ApplicationProperties>
```

2050

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

2051

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

2052

```
</ApplicationProperties>
```

2053

```
</PropertyDefaults>
```

2054

```
<NodeTemplate/>
```

2055

```
<NodeTemplate id="MyAppServer"
```

2056

```
  name="My Application Server"
```

2057

```
  nodeType="abc:ApplicationServer"
```

2058

```
  minInstances="0"
```

2059

```
  maxInstances="unbounded"/>
```

2060

```
<RelationshipTemplate id="MyDeploymentRelationship"
```

2061



```

2065         relationshipType="deployedOn">
2066     <SourceElement id="MyApplication"/>
2067     <TargetElement id="MyAppServer"/>
2068 </RelationshipTemplate>
2069
2070 </TopologyTemplate>
2071
2072 <NodeTypes>
2073     <NodeType name="Application">
2074         <documentation xml:lang="EN">
2075             A reusable definition of a node type representing an
2076             application that can be deployed on application servers.
2077         </documentation>
2078         <NodeTypeProperties element="ApplicationProperties"/>
2079         <InstanceStates>
2080             <InstanceState state="http://www.my.com/started"/>
2081             <InstanceState state="http://www.my.com/stopped"/>
2082         </InstanceStates>
2083         <Interfaces>
2084             <Interface>
2085                 <Operation name="DeployApplication">
2086                     <InputParameters>
2087                         <InputParamter name="InstanceName"
2088                             type="string"/>
2089                         <InputParamter name="AppServerHostname"
2090                             type="string"/>
2091                         <InputParamter name="ContextRoot"
2092                             type="string"/>
2093                     </InputParameters>
2094                     <Implementations>
2095                         <Implementation>
2096                             ...
2097                         </Implementation>
2098                     </Implementations>
2099                 </Operation>
2100             </Interface>
2101         </Interfaces>
2102     </NodeType>
2103     <NodeType name="ApplicationServer"
2104         targetNamespace="http://www.ibm.com/sample">
2105         <NodeTypeProperties element="AppServerProperties"/>
2106         <Interfaces>
2107             <Interface>
2108                 <Operation name="AcquireNetworkAddress">
2109                     <OutputParameters>
2110                         <OutputParamter name="Hostname"
2111                             type="string"/>
2112                         <OutputParamter name="IPAddress"
2113                             type="string"/>
2114                     </OutputParameters>
2115                     <Implementations>
2116                         <Implementation>
2117                             ...
2118                         </Implementation>
2119                     </Implementations>
2120                 </Operation>
2121                 <Operation name="DeployApplicationServer">
2122                     <InputParameters>

```

```

2123         <InputParamter name="Hostname"
2124             type="string"/>
2125         <InputParamter name="IPAddress"
2126             type="string"/>
2127         <InputParamter name="HeapSize"
2128             type="int"/>
2129         <InputParamter name="SoapPort"
2130             type="int"/>
2131     </InputParameters>
2132     <OutputParameters>
2133         <OutputParamter name="ServerID"
2134             type="string"/>
2135     </OutputParameters>
2136     <Implementations>
2137         <Implementation>
2138             ...
2139         </Implementation>
2140     </Implementations>
2141 </Operation>
2142 </Interface>
2143 </Interfaces>
2144 </NodeType>
2145 </NodeTypes>
2146
2147 <RelationshipTypes>
2148     <documentation xml:lang="EN">
2149         A reusable definition of relation that expresses deployment of
2150         an artifact on a hosting environment.
2151     </documentation>
2152     <RelationshipType name="deployedOn"
2153         semantics="www.my.com/RelSemantics/deployedOn">
2154     </RelationshipType>
2155 </RelationshipTypes>
2156
2157 <Plans>
2158     <Plan id="DeployApplication"
2159         name="Sample Application Build Plan"
2160         planType="http://docs.oasis-
2161             open.org/tosca/ns/2011/12/PlanTypes/BuildPlan"
2162         languageUsed="http://www.omg.org/spec/BPMN/2.0/">
2163
2164         <PreCondition expressionLanguage="www.my.com/text">?
2165             Run only if funding is available
2166         </PreCondition>
2167
2168     <PlanModel>
2169         <process name="DeployNewApplication" id="p1">
2170             <documentation>This process deploys a new instance of the
2171             sample application.
2172             </documentation>
2173
2174             <task id="t1" name="CreateAccount"/>
2175
2176             <task id="t2" name="AcquireNetworkAddresses"
2177                 isSequential="false"
2178                 loopDataInput="t2Input.LoopCounter"/>
2179             <documentation>Assumption: t2 gets data of type "input"
2180                 as input and this data has a field names "LoopCounter"

```

```
2181         that contains the actual multiplicity of the task.
2182     </documentation>
2183
2184     <task id="t3" name="DeployApplicationServer"
2185         isSequential="false"
2186         loopDataInput="t3Input.LoopCounter"/>
2187
2188     <task id="t4" name="DeployApplication"
2189         isSequential="false"
2190         loopDataInput="t4Input.LoopCounter"/>
2191
2192     <sequenceFlow id="s1" targetRef="t2" sourceRef="t1"/>
2193     <sequenceFlow id="s2" targetRef="t3" sourceRef="t2"/>
2194     <sequenceFlow id="s3" targetRef="t4" sourceRef="t3"/>
2195 </process>
2196 </PlanModel>
2197 </Plan>
2198
2199 <Plan id="RemoveApplication"
2200     planType="http://docs.oasis-
2201     open.org/tosca/ns/2011/12/PlanTypes/TerminationPlan"
2202     languageUsed="http://docs.oasis-
2203     open.org/wsbpel/2.0/process/executable">
2204     <PlanModelReference reference="prj:RemoveApp"/>
2205 </Plan>
2206 </Plans>
2207
2208 </ServiceTemplate>
```

2209

## Appendix E. Revision History

2210

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	Mike Edwards, Thomas Spatzier	Changes for JIRA Issue TOSCA-6: Reviewed and adapted normative statement keywords according to RFC2119.
wd-03	2012-03-06	Arvind Srinivasan, Mike Edwards, 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.

2211

2212