



Topology and Orchestration Specification for Cloud Applications Version 1.0

Committee Specification Draft 03

19 July 2012

Specification URIs

This version:

<http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd03/TOSCA-v1.0-csd03.doc> (Authoritative)
<http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd03/TOSCA-v1.0-csd03.html>
<http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd03/TOSCA-v1.0-csd03.pdf>

Previous version:

<http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd02/TOSCA-v1.0-csd02.doc> (Authoritative)
<http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd02/TOSCA-v1.0-csd02.html>
<http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd02/TOSCA-v1.0-csd02.pdf>

Latest version:

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

Technical Committee:

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

Chairs:

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

Editors:

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

Additional artifacts:

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

- XML schemas: <http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd03/schemas/>

Declared XML namespace:

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

Abstract:

The concept of a “service template” is used to specify the “topology” (or structure) and “orchestration” (or invocation of management behavior) of IT services (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 document was last revised or approved by the OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC on the above date. The level of approval is also listed above. Check the "Latest version" location noted above for possible later revisions of this document.

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

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

Citation format:

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

[TOSCA-v1.0]

Topology and Orchestration Specification for Cloud Applications Version 1.0. 19 July 2012.
OASIS Committee Specification Draft 03.

<http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd03/TOSCA-v1.0-csd03.html>.

Notices

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.

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

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

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

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

Table of Contents

1	Introduction.....	6
2	Language Design	7
2.1	Dependencies on Other Specifications	7
2.2	Notational Conventions.....	7
2.3	Normative References	7
2.4	Non-Normative References	8
2.5	Namespaces.....	8
2.6	Language Extensibility.....	8
2.7	Overall Language Structure.....	9
2.7.1	Syntax.....	9
2.7.2	Properties	9
3	Core Concepts and Usage Pattern	13
3.1	Core Concepts.....	13
3.2	Service Templates and Artifacts.....	14
3.3	Archive Format for Cloud Applications	15
3.4	Use Cases	16
3.4.1	Services as Marketable Entities	16
3.4.2	Portability of Service Templates.....	17
3.4.3	Service Composition	17
3.4.4	Relation to Virtual Images	17
4	Node Types	18
4.1	Syntax.....	18
4.2	Properties.....	20
4.3	Derivation Rules	23
4.4	Example	24
5	Relationship Types.....	26
5.1	Syntax.....	26
5.2	Properties.....	26
5.3	Derivation Rules	28
5.4	Example	28
6	Topology Template.....	30
6.1	Syntax	30
6.2	Properties.....	32
6.3	Example	36
7	Plans.....	38
7.1	Syntax	38
7.2	Properties.....	38
7.3	Use of Process Modeling Languages	39
7.4	Example	39
8	Cloud Service Archive (CSAR).....	41
8.1	Overall Structure of a CSAR.....	41
8.2	TOSCA Meta File.....	41
8.3	Example	42

9	Security Considerations	46
10	Conformance	47
Appendix A.	Portability and Interoperability Considerations	48
Appendix B.	Acknowledgements	49
Appendix C.	Complete TOSCA Grammar	51
Appendix D.	TOSCA Schema.....	57
Appendix E.	Sample	69
E.1	Sample Service Topology Definition	69
Appendix F.	Revision History	73

1 Introduction

2 IT services (or just *services* in what follows) are the main asset within IT environments in general, and in
3 cloud environments in particular. The advent of cloud computing suggests the utility of standards that
4 enable the (semi-) automatic creation and management of services (a.k.a. service automation). These
5 standards describe a service and how to manage it independent of the supplier creating the service and
6 independent of any particular cloud provider and the technology hosting the service. Making service
7 topologies (i.e. the individual components of a service and their relations) and their orchestration plans
8 (i.e. the management procedures to create and modify a service) interoperable elements, enables their
9 exchange between different environments. This specification explains how to define services in a portable
10 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].

This specification follows XML naming and design rules as described in [UNCEFACT XMLNDR], i.e. uses upper camel-case notation for XML element names and lower camel-case notation for XML attribute names.

2.3 Normative References

- | | |
|---------------------|---|
| [RFC2119] | S. Bradner, <i>Key words for use in RFCs to Indicate Requirement Levels</i> , http://www.ietf.org/rfc/rfc2119.txt , IETF RFC 2119, March 1997. |
| [RFC 2396] | Uniform Resource Identifiers (URI): Generic Syntax, RFC 2396, available via http://www.faqs.org/rfcs/rfc2396.html |
| [BPEL 2.0] | OASIS Web Services Business Process Execution Language (WS-BPEL) 2.0, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf |
| [BPMN 2.0] | OMG Business Process Model and Notation (BPMN) Version 2.0, http://www.omg.org/spec/BPMN/2.0/ |
| [OVF] | Open Virtualization Format Specification Version 1.1.0, http://www.dmtf.org/standards/published_documents/DSP0243_1.1.0.pdf |
| [WSDL 1.1] | Web Services Description Language (WSDL) Version 1.1, W3C Note, http://www.w3.org/TR/2001/NOTE-wsdl-20010315 |
| [XML Base] | XML Base (Second Edition), W3C Recommendation, http://www.w3.org/TR/xmlbase/ |
| [XML Infoset] | XML Information Set, W3C Recommendation, http://www.w3.org/TR/2001/REC-xml-infoset-20011024/ |
| [XML Namespaces] | Namespaces in XML 1.0 (Second Edition), W3C Recommendation, http://www.w3.org/TR/REC-xml-names/ |
| [XML Schema Part 1] | XML Schema Part 1: Structures, W3C Recommendation, October 2004, http://www.w3.org/TR/xmlschema-1/ |
| [XML Schema Part 2] | XML Schema Part 2: Datatypes, W3C Recommendation, October 2004, http://www.w3.org/TR/xmlschema-2/ |

- 54 **[XMLSpec]** XML Specification, W3C Recommendation, February 1998,
 55 <http://www.w3.org/TR/1998/REC-xml-19980210>
- 56 **[XPath 1.0]** XML Path Language (XPath) Version 1.0, W3C Recommendation, November
 57 1999, <http://www.w3.org/TR/1999/REC-xpath-19991116>
- 58 **[UNCEFACT XMLNDR]** UN/CEFACT XML Naming and Design Rules Technical Specification,
 59 Version 3.0,
 60 [http://www.unece.org/fileadmin/DAM/cefact/xml/UNCEFACT+XML+NDR+V3p0.p](http://www.unece.org/fileadmin/DAM/cefact/xml/UNCEFACT+XML+NDR+V3p0.pdf)
 61 [df](http://www.unece.org/fileadmin/DAM/cefact/xml/UNCEFACT+XML+NDR+V3p0.pdf)
 62

63 2.4 Non-Normative References

64
65

66 2.5 Namespaces

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

72

Prefix	Namespace
ste	http://docs.oasis-open.org/tosca/ns/2011/12
xs	http://www.w3.org/2001/XMLSchema
wsdl	http://schemas.xmlsoap.org/wsdl/
bpmn	http://www.omg.org/bpmn/2.0

73 Table 1: Prefixes and namespaces used in this specification

74

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

78 2.6 Language Extensibility

79 The TOSCA extensibility mechanism allows:

- 80 • Attributes from other namespaces to appear on any TOSCA element
- 81 • Elements from other namespaces to appear within TOSCA elements
- 82 • Extension attributes and extension elements MUST NOT contradict the semantics of any attribute
 83 or element from the TOSCA namespace

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

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

88 2.7 Overall Language Structure

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

93 2.7.1 Syntax

```
94 1 <ServiceTemplate id="ID"  
95 2     name="string"?  
96 3     targetNamespace="anyURI">  
97 4  
98 5     <Extensions>?  
99 6         <Extension namespace="anyURI"  
100 7             mustUnderstand="yes|no"?/>+  
101 8     </Extensions>  
102 9  
103 10    <Import namespace="anyURI"?  
104 11        location="anyURI"?  
105 12        importType="anyURI"/>*  
106 13  
107 14    <Types>?  
108 15        <xs:schema .../>*  
109 16    </Types>  
110 17  
111 18    (  
112 19        <TopologyTemplate>  
113 20            ...  
114 21        </TopologyTemplate>  
115 22    |  
116 23        <TopologyTemplateReference reference="xs:QName">  
117 24    )?  
118 25  
119 26    <NodeTypes>?  
120 27        ...  
121 28    </NodeTypes>  
122 29  
123 30    <RelationshipTypes>?  
124 31        ...  
125 32    </RelationshipTypes>  
126 33  
127 34    <Plans>?  
128 35        ...  
129 36    </Plans>  
130 37  
131 38 </ServiceTemplate>
```

132 2.7.2 Properties

133 The `ServiceTemplate` element has the following properties:

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

136

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

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

141
142 Note: The `name` attribute for elements defined in this specification can generally be used as
143 descriptive, human-readable name.

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

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

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

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

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

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

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

170 The `location` attribute contains a URI indicating the location of a document that contains
171 relevant definitions. The location URI MAY be a relative URI, following the usual rules for
172 resolution of the URI base [[XML Base](#), [RFC 2396](#)]. The `location` attribute is optional. An
173 `Import` element without a `location` attribute indicates that external definitions are used but
174 makes no statement about where those definitions might be found. The `location` attribute is a
175 hint and a TOSCA compliant implementation is not obliged to retrieve the document being
176 imported from the specified location.

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

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

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

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

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

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

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

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

214
215 The main ingredients of a Topology Template are a set of Node Templates and Relationship
216 Templates. The Node Templates are the nodes of the directed graph. The Relationship
217 Templates are the directed edges between the nodes; each indicates the semantics of the
218 corresponding relationships.

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

223
224 Note that either zero or one Topology Template MUST occur in a Service Template, either
225 defined in place via a `TopologyTemplate` element or referenced via a
226 `TopologyTemplateReference`.

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

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

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

233 A Service Template document can be intended to be instantiated into a service instance or it can be
234 intended to be composed into other Service Templates. A Service Template document intended to be
235 instantiated MUST contain either a `TopologyTemplate` or a `TopologyTemplateReference`,
236 but not both. A Service Template document intended to be composed MUST include at least one of a
237 `NodeTypes`, `RelationshipTypes`, or `Plans` element. This technique supports a modular definition
238 of Service Templates. For example, one document can contain only Node Types that are referenced by a
239 Service Template document that contains just a Topology Template and Plans. Similarly, Node Type
240 Properties can be defined in separate XML Schema Definitions that are imported and referenced when
241 defining a Node Type.

242 Example of the use of a type definition:

```
243 <Types>  
244   <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"  
245     elementFormDefault="qualified"  
246     attributeFormDefault="unqualified">  
247     <xs:element name="ProjectProperties">  
248       <xs:complexType>  
249         <xs:sequence>  
250           <xs:element name="Owner" type="xs:string"/>  
251           <xs:element name="ProjectName" type="xs:string"/>  
252           <xs:element name="AccountID" type="xs:string"/>  
253         </xs:sequence>  
254       </xs:complexType>  
255     </xs:element>  
256   </xs:schema>  
257 </Types>
```

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

```
261 1 <documentation source="anyURI"? xml:lang="language"?>  
262 2   ...  
263 3 </documentation>
```

264 Example of use of a documentation:

```
265 <ServiceTemplate id="myService" name="My Service" ...>  
266   <documentation xml:lang="EN">  
267     This is a simple example of the usage of the documentation  
268     element as nested under a ServiceTemplate element.  
269   </documentation>  
270 </ServiceTemplate>
```

273

3 Core Concepts and Usage Pattern

274
275

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

276

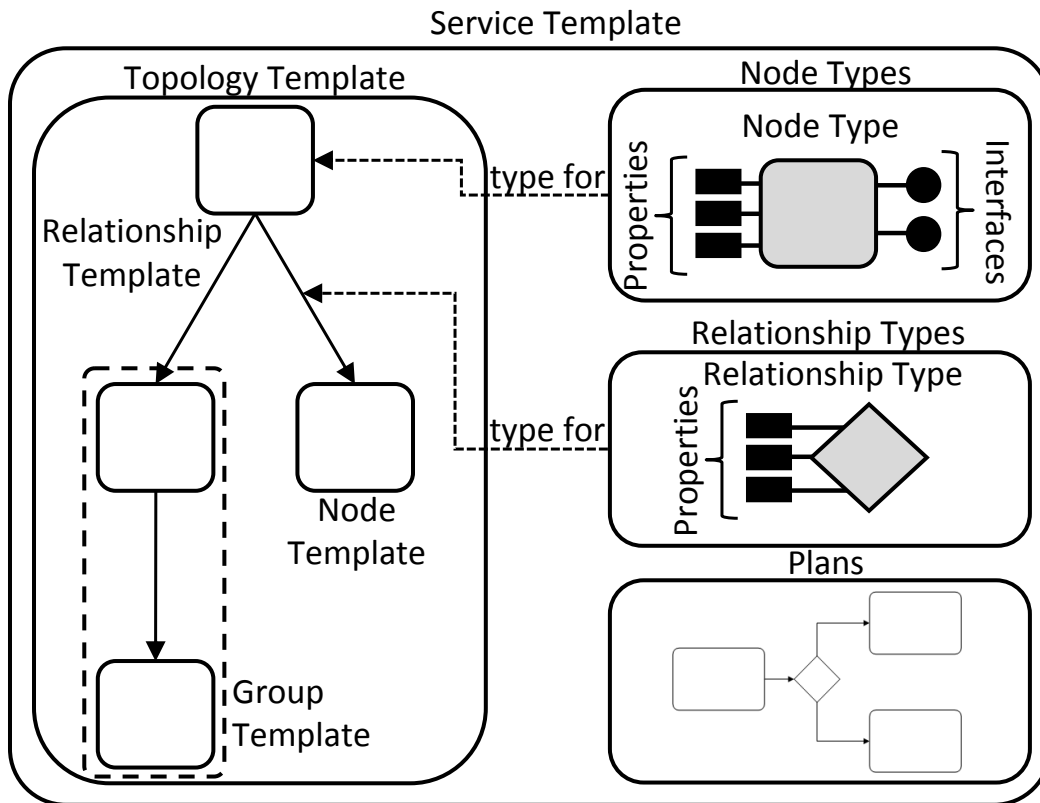
3.1 Core Concepts

277
278
279
280
281
282

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

283
284
285
286
287
288
289

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



290
291

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

292
293
294
295
296

For example, consider a service that consists of an application server, a process engine, and a process model. A Topology Template defining that service would include one Node Template of Node Type "application server", another Node Template of Node Type "process engine", and a third Node Template of Node Type "process model". The application server Node Type defines properties like the IP address of an instance of this type, an operation for installing the application server with the corresponding IP

297 address, and an operation for shutting down an instance of this application server. A constraint in the
298 Node Template can specify a range of IP addresses available when making a concrete application server
299 available.

300 A *Relationship Template* specifies the occurrence of a relationship between nodes in a Topology
301 Template. Each Relationship Template refers to a Relationship Type that defines the semantics and any
302 properties of the relationship. Relationship Types are defined separately for reuse purposes. The
303 Relationship Template indicates the elements it connects and the direction of the relationship by defining
304 one source and one target element (in nested `SourceElement` and `TargetElement` elements). The
305 Relationship Template also defines any constraints with the optional `RelationshipConstraints`
306 element.

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

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

318 For example, the application server Node Template will be instantiated by installing an actual application
319 server at a concrete IP address considering the specified range of IP addresses. Next, the process
320 engine Node Template will be instantiated by installing a concrete process engine on that application
321 server (as indicated by the “hosted by” relationship template). Finally, the process model Node Template
322 will be instantiated by deploying the process model on that process engine (as indicated by the “deployed
323 on” relationship template).

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

334 3.2 Service Templates and Artifacts

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

341 TOSCA distinguishes two kinds of artifacts: *implementation artifacts* and *deployment artifacts*. An
342 implementation artifact represents the executable of an operation of a node type, and a deployment
343 artifact represents the executable for materializing instances of a node. For example, a REST operation
344 to store an image may have an implementation artifact that is a WAR file. The node type this REST
345 operation is associated with may have the image itself as a deployment artifact.

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

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

349 The operations of a node type perform management actions on (instances of) the node type. The
350 implementations of such operations can be provided as implementation artifacts. Thus, the
351 implementation artifacts of the corresponding operations have to be deployed in the management
352 environment before any management operation can be started. In other words, “a TOSCA supporting
353 environment” (i.e. a so-called TOSCA container) must be able to process the set of implementation
354 artifacts types required to execute those management operations. One such management operation
355 could be the instantiation of a node type.

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

360 3.3 Archive Format for Cloud Applications

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



366
367

Figure 2: Structure of the CSAR

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

372 Each CSAR must contain a subdirectory called *TOSCA-Metadata*. This subdirectory must contain a so-
373 called *TOSCA meta file*. This file is named *TOSCA* and has the file extension *.meta*. It represents
374 metadata of the other files in the CSAR. This metadata is given in the format of name/value pairs. These
375 name/value pairs are organized in blocks. Each block provides metadata of a certain artifact of the CSAR.
376 An empty line separates the blocks in the *TOSCA meta file*.

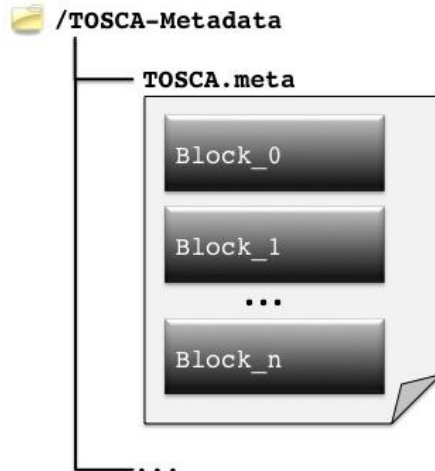


Figure 3: Structure of the TOSCA Meta File

377
378
379
380
381
382
383

The first block of the TOSCA meta file (Block_0 in Figure 3) provides metadata of the CSAR itself (e.g. its version, creator etc). Each other block begins with a name/value pair that points to an artifact within the CSAR by means of a pathname. The remaining name/value pairs in a block are the proper metadata of the pointed to artifact. For example, a corresponding name/value pair specifies the MIME-type of the artifact.

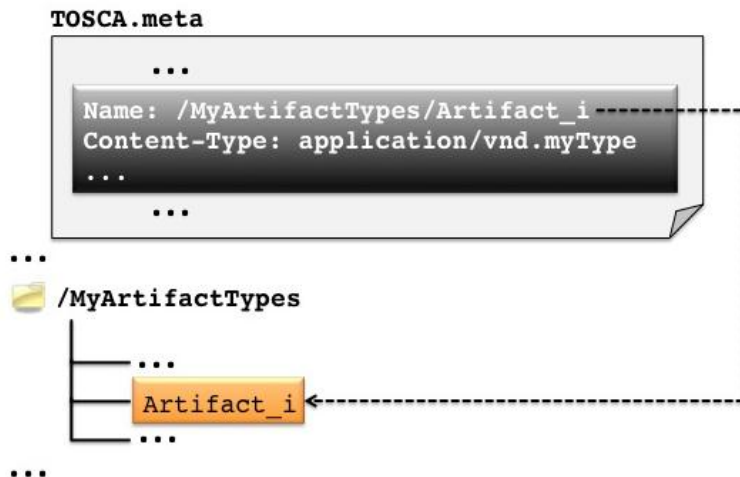


Figure 4: Providing Metadata for Artifacts

384
385
386

3.4 Use Cases

The specification supports at least the following major use cases.

3.4.1 Services as Marketable Entities

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

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

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

411 **3.4.2 Portability of Service Templates**

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

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

420 **3.4.3 Service Composition**

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

427 **3.4.4 Relation to Virtual Images**

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

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

440 4 Node Types

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

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

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

452 4.1 Syntax

```
453 1 <NodeTypes>?  
454 2  
455 3 <NodeType id="ID"  
456 4     name="string"?  
457 5     abstract="yes|no"?  
458 6     final="yes|no"?>+  
459 7  
460 8 <NodeTypeProperties element="QName"?  
461 9     type="QName"?/>?  
462 10  
463 11 <DerivedFrom nodeTypeRef="QName"/>?  
464 12  
465 13 <InstanceStates>?  
466 14 <InstanceState state="anyURI">+  
467 15 </InstanceStates>  
468 16  
469 17 <Interfaces>?  
470 18  
471 19 <Interface name="NCName | anyURI">+  
472 20  
473 21 <Operation name="NCName">*  
474 22  
475 23 (  
476 24 <WSDL portType="QName"  
477 25     operation="NCName"/>  
478 26 |  
479 27 <REST method="GET | PUT | POST | DELETE"  
480 28     abs_path="anyURI"?  
481 29     absoluteURI="anyURI"?  
482 30     requestBody="QName"?  
483 31     responseBody="QName"?>  
484 32  
485 33 <Parameters>?  
486 34 <Parameter name="string" required="yes|no"/>+  
487 35 </Parameters>  
488 36  
489 37 <Headers>?  
490 38 <Header name="string" required="yes|no"/>+
```

```

491 39         </Headers>
492 40
493 41     </REST>
494 42     |
495 43     <ScriptOperation>
496 44
497 45         <InputParameters>?
498 46
499 47             <InputParamter name="string"
500 48                 type="string"
501 49                 required="yes|no"/>+
502 50
503 51         </InputParameters>
504 52
505 53         <OutputParameters>?
506 54
507 55             <OutputParamter name="string"
508 56                 type="string"
509 57                 required="yes|no"/>+
510 58
511 59         </OutputParameters>
512 60
513 61     </ScriptOperation>
514 62 )
515 63
516 64 </Operation>
517 65
518 66 <ImplementationArtifacts>?
519 67
520 68     <ImplementationArtifact operationName="string"?
521 69         type="anyURI">+
522 70
523 71     <RequiredContainerCapabilities>?
524 72         <RequiredContainerCapability capability="anyURI"/>+
525 73     </RequiredContainerCapabilities>
526 74
527 75         artifact specific content
528 76
529 77     <ImplementationArtifact>
530 78
531 79 </ImplementationArtifacts>
532 80
533 81 </Interface>
534 82
535 83 </Interfaces>
536 84
537 85 <Policies>?
538 86     <Policy name="string" type="anyURI">+
539 87         policy specific content
540 88     </Policy>
541 89 </Policies>
542 90
543 91 <DeploymentArtifacts>?
544 92     <DeploymentArtifact name="string" type="anyURI">+
545 93         artifact specific content
546 94     </DeploymentArtifact>
547 95 </DeploymentArtifacts>
548 96

```

```
549 97     </NodeType>
550 98
551 99     </NodeTypes>
```

552 4.2 Properties

553 The `NodeType` element has the following properties:

- 554 • `id`: This attribute specifies the identifier of the Node Type. The identifier of the Node Type MUST
555 be unique within the target namespace.
- 556 • `name`: This optional attribute specifies the name of the Node Type.
- 557 • `abstract`: This optional attribute specifies that no instances can be created from Node
558 Templates that use this Node Type as their type.

559
560 As a consequence, the corresponding abstract Node Type referenced by any Node Template has
561 to be substituted by a Node Type derived from the abstract Node Type at the latest during the
562 instantiation time of a Node Template.

563
564 **Note:** an abstract Node Type MUST not be declared as final.

- 565 • `final`: This optional attribute specifies that no other Node Types MUST be derived from the
566 specific Node Type.

567
568 **Note:** a final Node Type MUST not be declared as abstract.

- 569 • `NodeTypeProperties`: This element specifies the structure of the observable properties of
570 the Node Type, such as its configuration and state, by means of XML schema.

- 571 • `DerivedFrom`: This is an optional reference to another Node Type from which this Node Type
572 derives. Conflicting definitions are resolved by the rule that local new definitions always override
573 derived definitions. See section 4.3 Derivation Rules for details.

- 574 • `InstanceStates`: This optional element lists the set of states an instance of this Node Type
575 can occupy at runtime.

- 576 • `Interfaces`: These are the definitions of functions that can be performed on (instances of) this
577 Node Type.

- 578 • `Policies`: The nested list of elements provides information related to a particular management
579 aspect like billing or monitoring.

- 580 • `DeploymentArtifacts`: This element specifies deployment artifacts relevant for the Node
581 Type. A deployment artifact is an entity that – if specified – is needed for creating an instance of
582 the corresponding Node Type. For example, a virtual image could be a deployment artifact of a
583 JEE server.

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

- 585 • The `element` attribute provides the QName of an XML element defining the structure of the
586 Node Type Properties.
- 587 • The `type` attribute provides the QName of an XML (complex) type defining the structure of the
588 Node Type Properties.

589 The `DerivedFrom` element has the following properties:

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

592 The `InstanceStates` element has the following properties:

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

594 The `InstanceState` element has the following properties:

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

596 The `Interface` element has the following properties:

- 597 • `name`: The name of the interface. This name is either a URI or it is an NCName that MUST be
598 unique in the target namespace.

- 599 • `Operation`: This element defines an operation available to manage particular aspects of the
600 Node Type. The `name` attribute of the `Operation` element defines the name of the operation
601 and MUST be unique within the containing `Interface` of the Node Type.
602

603 Note that an interface of a Node Type typically defines one or more operations. However,
604 interface definitions MAY also not contain any operation definition but just the definition of
605 implementation artifacts. This allows for the definition of interfaces with just the description of
606 operation signatures but not implementation artifacts in a Node Type. A derived Node Type can
607 then provide the implementation artifacts for operations without having to redefine operation
608 signatures provided by the Node Type derived from.

- 609 • `ImplementationArtifacts`: This element specifies a set of implementation artifacts for
610 operations in an interface.

611 The `Operation` element has the following properties:

- 612 • `WSDL`: The operation is implemented by means of Web Service technology. The port type and
613 operation of the Web Service are specified in the corresponding attributes of the `WSDL` element.
- 614 • `REST`: The operation is implemented as a REST API.
- 615 • `ScriptOperation`: The operation is implemented by scripts.

616 The `ImplementationArtifacts` element has the following properties:

- 617 • `ImplementationArtifact`: An implementation artifact of an operation. For example, a
618 servlet might be an implementation artifact for a REST API.

619
620 Multiple implementation artifacts might be required for a single operation, e.g. in case a script
621 operation is realized using different script languages in different environments.

622 The `WSDL` element has the following properties:

- 623 • `portType`: This is the QName of the port type that contains the definition of the operation
624 defined as part of the interface. Note that the corresponding namespace MUST be imported.
- 625 • `operation`: This attribute specifies the name of an operation of the port type to become part of
626 the interface.

627 The `REST` element has the following properties:

- 628 • `method`: The HTTP method to be used for building the REST request. If no method is explicitly
629 specified, GET is assumed as default.

- 630 • `abs_path`: The absolute path of the URI that represents the target resource of the request.

631 Note, that the proper network location of the URI MUST be set as value of the Host header field
632 of the request when using `abs_path` instead of `absoluteURI`.

- 633 • `absoluteURI`: The absolute URI of the resource.

- 634 Note, that either the `abs_path` or the `absoluteURI` MUST be specified.
- 635 • `requestBody`: The data passed in the body of the request message. The QName value of this
636 attribute identifies the specification of the body, e.g. it refers to an XML Schema Definition
637 document.
 - 638 • `responseBody`: The data returned in the body of the response message. The QName value of
639 this attribute identifies the specification of the body, e.g. it refers to an XML Schema Definition
640 document.
 - 641 • `Parameters`: This nested element describes a list of parameters as nested `Parameter`
642 elements. Each `Parameter` has a `name` attribute and a `required` attribute that indicates
643 whether the parameter is required or not. This list is the base for building the query string of the
644 URI.
 - 645 • `Headers`: This nested element describes a list of HTTP request headers as nested `Header`
646 elements. Each `Header` has a `name` attribute and a `required` attribute that indicates whether
647 the header is required or not. Only those headers SHOULD be listed that might be important for
648 specifying the semantics of the request; otherwise, the HTTP client will set HTTP headers as
649 usual.

650 The `ScriptOperation` element has the following properties:

- 651 • `InputParameters`: This optional property contains one or more nested `InputParameter`
652 elements. Each such element specifies three attributes: the `name` of the parameter, its `type`,
653 and whether it must be available as input (`required` attribute with a value of “yes”, which is the
654 default) or not (value “no”). Note that the types of the parameters specified for an operation
655 MUST comply with the type systems of the languages of implementations.
- 656 • `OutputParameters`: This optional property contains one or more nested
657 `OutputParameter` elements. Each such element specifies three attributes: the `name` of the
658 parameter, its `type`, and whether it must be available as output (`required` attribute with a
659 value of “yes”, which is the default) or not (value “no”). Note that the types of the parameters
660 specified for an operation MUST comply with the type systems of the languages of
661 implementations.

662 The `ImplementationArtifact` element has the following properties:

- 663 • `operationName`: This optional attribute specifies the name of the operation that is
664 implemented by the actual implementation artifact. If not specified, the implementation artifact is
665 assumed to provide the implementation for all operations defined within the containing interface.
666 For example, a .WAR file could provide the implementation for all REST operations of an
667 interface.
- 668 • `type`: The type of the implementation artifact determines the specific content of the
669 `ImplementationArtifact` element. For example, a script might be provided in place or by
670 reference. A corresponding value of the `type` attribute indicates this.
- 671 • `RequiredContainerCapabilities`: An implementation of an operation might depend on
672 certain capabilities of the environment it is executed in. For example, an implementation of an
673 operation might use a particular interface for manipulating images, EJBs etc.
674 Each such dependency is explicitly declared by a separate
675 `RequiredContainerCapability` element. The `capability` attribute of this element is
676 a URI that denotes the corresponding requirement on the environment.

677 The `Policy` element has the following properties:

- 678
- 679
- 680
- 681
- The `type` attribute specifies the kind of policy (e.g. management practice) supported by an instance of the Node Type containing this element. The `name` attribute defines the name of the policy. The name value MUST be unique within a given Node Type containing the current definition of the Policy.

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

690 The `DeploymentArtifact` element has the following properties:

- 691
- 692
- `name`: The attribute specifies the name of the artifact. Note, that uniqueness of the name within the scope of the encompassing Node Type SHOULD be guaranteed by the definition.

- 693
- 694
- 695
- `type`: The attribute specifies the type of the deployment artifact definition that is related to the Node Type, i.e. the attribute gives a hint how to interpret the body of the `DeploymentArtifact` element.

696 Note, that the combination of name and type SHOULD be unique within the scope of the Node
697 Type.

- 698
- The body of this element contains the type-specific content.

699

700 For example, if the `type` attribute contains the value

701 `http://docs.oasis-open.org/tosca/ns/2011/12/deploymentArtifacts/ovfRef`, the body will contain an
702 XML fragment with a reference to an OVF package and a mapping between service template
703 data and elements of the respective OVF envelope.

704 4.3 Derivation Rules

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

- 706
- 707
- 708
- **Node Type Properties:** It is assumed that the XML element (or type) representing the Node Type Properties extends the XML element (or type) of the Node Type Properties of the Node Type referenced in the `DerivedFrom` element.

- 709
- 710
- 711
- 712
- **Instance States:** The set of instance states of the Node Type under definition consists of the set union of the instances states defined by the Nodes Type derived from and the instance states defined by the Node Type under definition. A set of instance states of the same name will be combined into a single instance state of the same name.

- 713
- 714
- 715
- **Interfaces:** The set of interfaces of the Node Type under definition consists of the set union of interfaces defined by the Node Type derived from and the interfaces defined by the Node Type under definition.

716 Two interfaces of the same name will be combined into a single, derived interface with the same
717 name. The set of operations of the derived interface consists of the set union of operations
718 defined by both interfaces. If an operation defined by the Node Type under definition has the
719 same name as an operation of the Node Type derived from, the former operation substitutes the
720 latter one.

- 721
- 722
- 723
- **Implementation Artifacts:** The set of implementation artifacts of the Node Type under definition consists of the set union of implementation artifacts defined by the Node Type derived from and the implementation artifacts defined by the Node Type under definition.

724 If an implementation artifact defined by the Node Type under definition has the same operation
725 name and type as an implementation artifact of the Node Type derived from, the former
726 implementation artifact substitutes the latter one.

- 727 • Deployment Artifacts: The set of deployment artifacts of the Node Type under definition consists
728 of the set union of the deployment artifacts defined by the Nodes Type derived from and the
729 deployment artifacts defined by the Node Type under definition. A deployment artifact defined by
730 the Node Type under definition substitutes a deployment artifact with the same name and type of
731 the Node Type derived from.
- 732 • Policies: The set of policies of the Node Type under definition consists of the set union of the
733 policies defined by the Nodes Type derived from and the policies defined by the Node Type under
734 definition. A policy defined by the Node Type under definition substitutes a policy with the same
735 name and type of the Node Type derived from.

736 4.4 Example

737 The following example defines the Node Type “Project”. It is defined in a Service Template “myService”
738 within the target namespace “http://www.ibm.com/sample”. Thus, by importing the corresponding
739 namespace in another Service Template, the Project Node Type is available for use in the other Service
740 Template.

```
741 <ServiceTemplate id="myService" name="My Service"  
742     targetNamespace="http://www.ibm.com/sample">  
743  
744     <NodeTypes>  
745  
746         <NodeType id="Project" name="My Project">  
747  
748             <documentation xml:lang="EN">  
749                 A reusable definition of a node type supporting  
750                 the creation of new projects.  
751             </documentation>  
752  
753             <NodeTypeProperties element="ProjectProperties"/>  
754  
755             <InstanceStates>  
756                 <InstanceState state="www.example.com/active"/>  
757                 <InstanceState state="www.example.com/onHalt"/>  
758             </InstanceStates>  
759  
760             <Interfaces>  
761                 <Interface name="ProjectInterface">  
762                     <Operation name="CreateProject">  
763                         <ScriptOperation>  
764                             <InputParameters>  
765                                 <InputParamter name="ProjectName"  
766                                     type="string"/>  
767                                 <InputParamter name="Owner"  
768                                     type="string"/>  
769                                 <InputParamter name="AccountID"  
770                                     type="string"/>  
771                             </InputParameters>  
772                         </ScriptOperation>  
773                     </Operation>  
774                 <ImplementationArtifacts>  
775                     <ImplementationArtifact operationName="CreateProject"  
776                         type="http://www.example.com/ScriptArtifact/PythonReference">
```



```
777         scripts/phython/createProject.py
778         </ImplementationArtifact>
779     </ImplementationArtifacts>
780 </Interface>
781 </Interfaces>
782
783 </NodeType>
784
785 </NodeTypes>
786
787 </ServiceTemplate>
```

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

797

5 Relationship Types

798 This chapter specifies how *Relationship Types* are defined. A Relationship Type is a reusable entity that
 799 defines the type of one or more Relationship Templates between Node Templates. As such, a
 800 Relationship Type can define the structure of observable properties via *Relationship Type Properties*. The
 801 functions that can be performed on (an instance of) a corresponding Relationship Template are defined
 802 by the *Interfaces* of the Relationship Type. Furthermore, a Relationship Type defines the potential states
 803 an instance of it might reveal at runtime. A Relationship Type can inherit the definitions just listed from
 804 another Relationship Type by means of the `DerivedFrom` element.

5.1 Syntax

```

806 1 <RelationshipTypes>
807 2
808 3   <RelationshipType id="ID"
809 4       name="string"?
810 5       abstract="yes|no"?
811 6       final="yes|no"?
812 7       cascadingDeletion="yes|no"?>+
813 8
814 9   <RelationshipTypeProperties element="QName"?
815 10       type="QName"?/>?
816 11
817 12   <DerivedFrom relationshipTypeRef="QName"/>?
818 13
819 14   <InstanceStates>?
820 15       <InstanceState state="anyURI">+
821 16   </InstanceStates>
822 17
823 18   <SourceInterfaces>?
824 19       <Interface name="NCName | anyURI">+
825 20       ...
826 21   </Interface>
827 22 </SourceInterfaces>
828 23
829 24   <TargetInterfaces>?
830 25       <Interface name="NCName | anyURI">+
831 26       ...
832 27   </Interface>
833 28 </TargetInterfaces>
834 29
835 30   <ValidSource typeRef="QName"/>?
836 31
837 32   <ValidTarget typeRef="QName"/>?
838 33
839 34 </RelationshipType>
840 35
841 36 </RelationshipTypes>
  
```

5.2 Properties

842 The `RelationshipType` element has the following properties:

- 844 • `id`: This attribute specifies the identifier of the Relationship Type. The identifier of the
 845 Relationship Type MUST be unique within the target namespace.

- 846 • `name`: This optional attribute specifies the name of the Relationship Type.
- 847 • `abstract`: This optional attribute specifies that no instances can be created from Relationship
- 848 Templates that use this Relationship Type as their type.
- 849
- 850 As a consequence, the corresponding abstract Relationship Type referenced by any Relationship
- 851 Template has to be substituted by a Relationship Type derived from the abstract Relationship
- 852 Type at the latest during the instantiation time of a Relationship Template.
- 853
- 854 Note: an abstract Relationship Type MUST not be declared as final.
- 855 • `final`: This optional attribute specifies that no other Relationship Types MUST be derived from
- 856 the specific Relationship Type.
- 857
- 858 Note: a final Relationship Type MUST not be declared as abstract.
- 859 • `cascadingDeletion`: If set to “yes” the target of an instance of a Relationship Template of
- 860 this RelationshipType is automatically deleted when the source of the instance of the Relationship
- 861 Template is deleted.
- 862 • `RelationshipTypeProperties`: This element specifies the structure of the observable
- 863 properties of the Relationship Type, such as its configuration and state, by means of XML
- 864 schema.
- 865 • `DerivedFrom`: This is an optional reference to another Relationship Type from which this
- 866 Relationship Type derives. Conflicting definitions are resolved by the rule that local new
- 867 definitions always override derived definitions. See section 5.3 Derivation Rules for details.
- 868 • `InstanceStates`: This optional element lists the set of states an instance of this Node Type
- 869 can occupy at runtime.
- 870 • `SourceInterfaces`: This optional element contains definitions of manageability interfaces
- 871 that can be performed on the source of a relationship of this Relationship Type to actually
- 872 establish the relationship between the source and the target in the deployed service.
- 873 Those interface definitions are contained in nested `Interface` elements, the content of which
- 874 is that described for Node Type interfaces (see section 4.2).
- 875 • `TargetInterfaces`: This optional element contains definitions of manageability interfaces
- 876 that can be performed on the target of a relationship of this Relationship Type to actually
- 877 establish the relationship between the source and the target in the deployed service.
- 878 Those interface definitions are contained in nested `Interface` elements, the content of which
- 879 is that described for Node Type interfaces (see section 4.2).
- 880 • `ValidSource`: This optional element specifies by means of its `typeRef` attribute the QName
- 881 of a Node Type that is allowed as a valid origin for relationships defined using the Relationship
- 882 Type under definition. Node Types derived from the specified Node Type MUST also be accepted
- 883 as valid origin of relationships.
- 884 If not specified, any Node Type is allowed to be the origin of the relationship.
- 885 • `ValidTarget`: This optional element specifies by means of its `typeRef` attribute the QName
- 886 of a Node Type that is allowed as a valid target for relationships defined using the Relationship
- 887 Type under definition. Node Types derived from the specified Node Type MUST also be accepted
- 888 as valid target of relationships.
- 889 If not specified, any Node Type is allowed to be the origin of the relationship.

890 The `RelationshipTypeProperties` element has the following properties:

891 • `element`: The QName value of this attribute refers to an XML element defining the structure of
892 the Relationship Type Properties.

893 • `type`: The QName value of this attribute refers to an XML (complex) type defining the structure
894 of the Relationship Type Properties.

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

896 The `InstanceStates` element has the following properties:

897 • `InstanceState`: specifies a potential state.

898 The `InstanceState` element has the following properties:

899 • `state`: a URI that represents a potential state.

900 5.3 Derivation Rules

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

902 • Relationship Type Properties: It is assumed that the XML element (or type) representing the
903 Relationship Type properties of the Relationship Type under definition extends the XML element
904 (or type) of the Relationship Type properties of the Relationship Type referenced in the
905 `DerivedFrom` element.

906 • Instance States: The resulting set of instance states of the Relationship Type under definition
907 consists of the set union of the instances states defined by the Relationship Type derived from
908 and the instance states explicitly defined by the Relationship Type under definition. Instance
909 states with the same state attribute will be combined into a single instance state of the same
910 state.

911 • Valid source and target: A Node Type specified as a valid source or target, respectively, of the
912 Relationship Type under definition MUST be of a subtype defined as valid source or target,
913 respectively, of the Relationship Type derived from.

914
915 If the Relationship Type derived from has no valid source or target defined, the Node Types of the
916 `ValidSource` or `ValidTarget` elements of the Relationship Type under definition are not
917 restricted.

918
919 If the Relationship Type under definition has no source or target defined, only the Node Types
920 defined as source or target of the Relationship Type derived from are valid origins or destinations
921 of the Relationship Type under definition.

922 5.4 Example

923 The following example defines the Relationship Type “`processDeployedOn`”. The meaning of this
924 Relationship Type is that “a process is deployed on a hosting environment”. When the source of an
925 instance of a Relationship Template referring to this Relationship Type is deleted, its target is
926 automatically deleted as well. The Relationship Type has Relationship Type Properties defined in the
927 `Types` section of the same Service Template document as the “`ProcessDeployedOnProperties`” element.
928 The states an instance of this Relationship Type can be in are also listed.

```
929 <RelationshipTypes>  
930  
931   <RelationshipType id="processDeployedOn"  
932     name="Process is deployed on"  
933     cascadingDeletion="yes">  
934  
935     <RelationshipTypeProperties element="ProcessDeployedOnProperties"/>
```

```
936
937     <InstanceStates>
938         <InstanceState state="www.example.com/successfullyDeployed"/>
939         <InstanceState state="www.example.com/failed"/>
940     </InstanceStates>
941
942 </RelationshipType>
943
944 </RelationshipTypes>
```

945

6 Topology Template

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

6.1 Syntax

```
952 1 <TopologyTemplate id="ID"  
953 2     name="string"?>  
954 3  
955 4   (  
956 5     <NodeTemplate id="ID"  
957 6         name="string"?  
958 7         nodeType="QName"  
959 8         minInstances="int"?  
960 9         maxInstances="int|string"?>  
961 10  
962 11     <PropertyDefaults>?  
963 12         XML fragment  
964 13     </PropertyDefaults>  
965 14  
966 15     <PropertyConstraints>?  
967 16  
968 17         <PropertyConstraint property="string"  
969 18             constraintType="anyURI">+  
970 19             constraint?  
971 20         </PropertyConstraint>  
972 21     </PropertyConstraints>  
973 22  
974 23     <Policies>?  
975 24         <Policy name="string" type="anyURI">+  
976 25             policy specific content  
977 26         </Policy>  
978 27     </Policies>  
979 28  
980 29     <EnvironmentConstraints>?  
981 30         <EnvironmentConstraint constraintType="anyURI">+  
982 31             constraint type specific content?  
983 32         </EnvironmentConstraint>  
984 33     </EnvironmentConstraints>  
985 34  
986 35     <DeploymentArtifacts>?  
987 36         <DeploymentArtifact name="string" type="anyURI">+  
988 37             artifact specific content  
989 38         </DeploymentArtifact>  
990 39     </DeploymentArtifacts>  
991 40  
992 41     <ImplementationArtifacts>?  
993 42         <ImplementationArtifact operationName="string"  
994 43             type="anyURI">+  
995 44  
996 45     <RequiredContainerCapabilities>?
```

```

997 46         <RequiredContainerCapability capability="anyURI"/>+
998 47         </RequiredContainerCapabilities>
999 48         artifact specific content
1000 49         <ImplementationArtifact>
1001 50         </ImplementationArtifacts>
1002 51
1003 52     </NodeTemplate>
1004 53     |
1005 54     <RelationshipTemplate id="ID"
1006 55         name="string"?
1007 56         relationshipType="QName">
1008 57
1009 58         <SourceElement id="IDREF"/>
1010 59
1011 60         ( <TargetElement id="IDREF"/>
1012 61         |
1013 62         <TargetElementReference id="QName"/>
1014 63         )
1015 64
1016 65         <PropertyDefaults>?
1017 66         XML fragment
1018 67         </PropertyDefaults>
1019 68
1020 69         <PropertyConstraints>?
1021 70
1022 71         <PropertyConstraint property="string"
1023 72             constraintType="anyURI">+
1024 73             constraint?
1025 74         </PropertyConstraint>
1026 75
1027 76         </PropertyConstraints>
1028 77
1029 78         <RelationshipConstraints>?
1030 79
1031 80         <RelationshipConstraint constraintType="anyURI">+
1032 81             constraint?
1033 82         </RelationshipConstraint>
1034 83
1035 84         </RelationshipConstraints>
1036 85
1037 86     </RelationshipTemplate>
1038 87     |
1039 88     <GroupTemplate id="ID"
1040 89         name="string"?
1041 90         minInstances="int"?
1042 91         maxInstances="int|string"?>
1043 92
1044 93         (
1045 94         <NodeTemplate ... />
1046 95         |
1047 96         <RelationshipTemplate ... />
1048 97         |
1049 98         <GroupTemplate ... />
1050 99         )+
1051 100
1052 101         <Policies>?
1053 102         <Policy name="string" type="anyURI">+
1054 103         policy specific content

```

```
1055 104         </Policy>
1056 105         </Policies>
1057 106
1058 107         </GroupTemplate>
1059 108     )+
1060 109
1061 110     </TopologyTemplate>
```

1062 6.2 Properties

1063 The `TopologyTemplate` element has the following properties:

- 1064 • `id`: This attribute specifies the identifier of the Topology Template. The identifier of the Topology
1065 Template MUST be unique within the target namespace.
- 1066 • `name`: This optional attribute specifies the name of the Topology Template.
- 1067 • `NodeTemplate`: This is a kind of a component making up the IT service.
- 1068 • `RelationshipTemplate`: This is a kind of relationship between the components (nodes or
1069 groups) of the service.
- 1070 • `GroupTemplate`: This is a grouping of node templates, relationship templates, or (nested)
1071 group templates within the Topology Templates to express a special association between the
1072 grouped elements.

1073 A Topology Template can contain any number of Node Templates, Relationship Templates, or Group
1074 Templates (i.e. “elements”). For each specified Relationship Template (either defined as a direct child of
1075 the Topology Template or within a Group Template) the source element and target element MUST be
1076 specified in the Topology Template except for target elements that are referenced (via a target element
1077 reference).

1078 The `NodeTemplate` element has the following properties:

- 1079 • `id`: This attribute specifies the identifier of the Node Template. The identifier of the Node
1080 Template MUST be unique within the target namespace.
- 1081 • `name`: This optional attribute specifies the name of the Node Template.
- 1082 • `nodeType`: The QName value of this attribute refers to the Node Type providing the type of the
1083 Node Template.
1084
1085 Note: If the Node Type referenced by `nodeType` attribute of a Node Template is declared as
1086 abstract, no instances of the specific Node Template can be created. Instead, a substitution of the
1087 Node Type with a specialized, derived Node Type has to be done at the latest during the
1088 instantiation time of the Node Template.
- 1089 • `minInstances`: This integer attribute specifies the minimum number of instances to be created
1090 when instantiating the Node Template. The default value of this attribute is 1..The value of
1091 `minInstances` MUST NOT be less than 0.
- 1092 • `maxInstances`: This attribute specifies the maximum number of instances that can be created
1093 when instantiating the Node Template. The default value of this attribute is 1. If the string is set to
1094 “unbounded”, an unbounded number of instances can be created. The value of `maxInstances`
1095 MUST be 1 or greater and MUST NOT be less than the value specified for `minInstances`.
- 1096 • `PropertyDefaults`: Specifies initial values for one or more of the Node Type Properties of
1097 the Node Type providing the property definitions in the concrete context of the Node Template.

1098 The initial values are specified by providing an instance document of the XML schema of the
1099 corresponding Node Type Properties. This instance document considers the inheritance structure
1100 deduced by the `DerivedFrom` property of the Node Type referenced by the `nodeType`
1101 attribute of the Node Template.
1102

1103 The instance document of the XML schema might not validate against the existence constraints
1104 of the corresponding schema: not all Node Type properties might have an initial value assigned,
1105 i.e. mandatory elements or attributes might be missing in the instance provided by the Property
1106 Defaults element. Once the defined Node Template has been instantiated, any XML
1107 representation of the Node Type properties MUST validate according to the associated XML
1108 schema definition.

- 1109 • `PropertyConstraints`: Specifies constraints on the use of one or more of the Node Type
1110 Properties of the Node Type providing the property definitions for the Node Template.

1111 Each constraint is specified by means of a separate nested `PropertyConstraint` element.
1112 This element contains the actual encoding of the constraint.

- 1113 • `Policies`: Specifies policies of the Node Template. Each policy is specified by means of a
1114 separate nested `Policy` element. This element contains the actual policy specific content of the
1115 policy.

1116 Note, that a policy specified in the Node Template overrides any policy of the same name and
1117 type that might be specified with the Node Type of this Node Template.
1118

1119 Any policies of the Node Type that are not overridden are combined with the policies of the Node
1120 Template.

- 1121 • `EnvironmentConstraints`: The nested `EnvironmentConstraint` elements of the
1122 Node Template under definition constrain the runtime environment for the corresponding
1123 component of a service. For example, constraints on network security settings of the hosting
1124 environment or requirements on the existence of certain resources might be defined within the
1125 environment constraints definition of a Node Template.

- 1126 • `DeploymentArtifacts`: This element specifies the deployment artifacts relevant for the
1127 Node Template under definition.
1128

1129 Its nested `DeploymentArtifact` elements specify details about individual deployment
1130 artifacts. The name attribute of a `DeploymentArtifact` element specifies the name of the
1131 artifact. Uniqueness of the name within the scope of the encompassing Node Template SHOULD
1132 be guaranteed by the definition. The `type` attribute of a `DeploymentArtifact` element
1133 specifies the type of the deployment artifact definition that is related to the Node Template, i.e.
1134 the attribute gives a hint how to interpret the body of the `DeploymentArtifact` element. The
1135 body of this element contains the type-specific content.
1136

1137 For example, if the `type` attribute contains the value
1138 <http://docs.oasis-open.org/tosca/ns/2011/12/deploymentArtifacts/ovfRef>, the body will contain an
1139 XML fragment with a reference to an OVF package and a mapping between service template
1140 data and elements of the respective OVF envelope.
1141

1142 Note, that a deployment artifact specified with the Node Template under definition overrides any
1143 deployment artifact of the same name and the same type specified with the Node Type given as
1144 value of the `nodeType` attribute of the Node Template under definition.
1145

1146 Otherwise, the deployment artifacts of the Node Type given as value of the `nodeType` attribute
1147 of the Node Template under definition and the deployment artifacts defined with the Node
1148 Template are combined.

- 1149 • `ImplementationArtifact`: An implementation of an operation. For example, a servlet
1150 might be an implementation artifact for a REST API. Multiple implementation artifacts might be
1151 required for a single operation, e.g. in case a script operation is realized using different script
1152 languages in different environments.

1153

1154 The `operationName` attribute specifies the name of the operation that is implemented by the
1155 implementation artifact under definition. The `type` attribute determines the specific content of the
1156 `ImplementationArtifact` element. For example, a script might be provided in place or by
1157 reference. A corresponding value of the `type` attribute indicates this.

1158

1159 The nested `RequiredContainerCapabilities` element specifies certain capabilities of
1160 the environment an implementation of an operation might depend on. For example, an
1161 implementation of an operation might use a particular interface for manipulating images, EJBs
1162 etc. Each such dependency is explicitly declared by a separate
1163 `RequiredContainerCapability` element. The `capability` attribute of this element is
1164 a URI that denotes the corresponding requirement on the environment.

1165

1166 Note, that an implementation artifact specified with the Node Template under definition overrides
1167 any implementation artifact with the same `operationName` and the same `type` specified with
1168 the Node Type given as value of the `nodeType` attribute of the Node Template under definition.

1169

1170 Otherwise, the implementation artifacts of the Node Type given as value of the `nodeType`
1171 attribute of the Node Template under definition and the implementation artifacts defined with the
1172 Node Template are combined.

1173 The `PropertyConstraint` element has the following properties:

- 1174 • `property`: The string value of this property is an XPath expression pointing to the property
1175 within the Node Type Properties document that is constrained within the context of the Node
1176 Template. More than one constraint MUST NOT be defined for each property.
- 1177 • `constraintType`: The constraint type is specified by means of a URI, which defines both the
1178 semantic meaning of the constraint as well as the format of the content.

1179

1180 For example, a constraint type of `http://www.example.com/PropertyConstraints/unique` could
1181 denote that the reference property of the node template under definition has to be unique within a
1182 certain scope. The constraint type specific content of the respective `PropertyConstraint`
1183 element could then define the actual scope in which uniqueness has to be ensured in more detail.

1184 The `Policy` element has the following properties:

- 1185 • `type`: This attribute specifies the kind of policy (e.g. management practice) supported by an
1186 instance of the Node Type containing this element.
- 1187 • `name`: This attribute defines the name of the policy. The name MUST be unique within a given
1188 Node Type containing the `Policy` element.

1189 The `EnvironmentConstraint` element has the following properties:

- 1190 • `constraintType`: The constraint type is specified by means of a URI, which defines both the
1191 semantic meaning of the constraint as well as the format of the constraint content.

1192 The `RelationshipTemplate` element has the following properties:

- 1193 • `id`: This attribute specifies the identifier of the Relationship Template. The identifier of the
1194 Relationship Template MUST be unique within the target namespace.

- 1195
- `name`: This optional attribute specifies the name of the Relationship Template.
- 1196
- `relationshipType`: The QName value of this property refers to the Relationship Type
- 1197
- 1198
- 1199
- 1200
- 1201
- 1202
- 1203
- Note: If the Relationship Type referenced by the `relationshipType` attribute of a Relationship Template is declared as abstract, no instances of the specific Relationship Template can be created. Instead, a substitution of the Relationship Type with a specialized, derived Relationship Type has to be done at the latest during the instantiation time of the Relationship Template.
- `SourceElement`: The `id` attribute of this element references a Node Template or Group Template within the same Service Template document that is the source of the Relationship Template.
- 1204
- 1205
- 1206
- 1207
- 1208
- 1209
- 1210
- 1211
- If the Relationship Type referenced by the `relationshipType` attribute defines a constraint on the valid source of the relationship by means of its `ValidSource` element, the `id` attribute of `SourceElement` MUST reference a Node Template the type of which complies with the valid source constraint of the respective Relationship Type.
- `TargetElement`: The `id` attribute of this element references a Node Template or Group Template within the same Service Template document that is the target of the Relationship Template.
- 1212
- 1213
- 1214
- 1215
- 1216
- 1217
- 1218
- 1219
- If the Relationship Type referenced by the `relationshipType` attribute defines a constraint on the valid target of the relationship by means of its `ValidTarget` element, the `id` attribute of `TargetElement` MUST reference a Node Template the type of which complies with the valid target constraint of the respective Relationship Type.
- `TargetElementReference`: The `id` attribute of this element refers by QName to an imported Node Template or Group Template that is the target of the Relationship Template. The referenced Node Template or Group Template will typically be the root node or root group of the corresponding Topology Template. In some cases a non-root Node Template or non-root Group Template might be referenced to support access to particular resources from a larger service, for example.
- 1220
- 1221
- 1222
- 1223
- 1224
- 1225
- 1226
- 1227
- 1228
- 1229
- 1230
- 1231
- 1232
- If the Relationship Type referenced by the `relationshipType` attribute defines a constraint on the valid target of the relationship by means of its `ValidTarget` element, the `id` attribute of `TargetElementReference` MUST reference a Node Template the type of which complies with the valid target constraint of the respective Relationship Type.
- Either `TargetElement` or `TargetElementReference` MUST be specified but not both.
- `PropertyDefaults`: Specifies initial values for one or more of the Relationship Type properties of the Relationship Type providing the property definitions in the concrete context of the Relationship Template.
- 1233
- 1234
- 1235
- 1236
- 1237
- The initial values are specified by providing an instance document of the XML schema of the corresponding Relationship Type properties.
- 1238
- 1239
- 1240
- 1241
- The instance document of the XML schema might not validate against the existence constraints of the corresponding schema: not all Relationship Type properties might have an initial value assigned, i.e. mandatory elements or attributes might be missing in the instance provided by the `PropertyDefaults` element. Once the defined Relationship Template has been instantiated, any

- 1242 XML representation of the Relationship Type properties MUST validate according to the
1243 associated XML schema definition.
- 1244 • `PropertyConstraints`: Specifies constraints on the use of one or more of the Relationship
1245 Type properties of the Relationship Type providing the property definitions for the Relationship
1246 Template.
- 1247 Each constraint is specified by means of a separate nested `PropertyConstraint` element.
1248 This element contains the actual encoding of the constraint.
- 1249 • `RelationshipConstraints`: Specifies constraints on the use of the relationship.
- 1250 Each constraint is specified by means of a separate nested `RelationshipConstraint`
1251 element. This element can contain the actual encoding of the constraint, or its
1252 `constraintType` attribute already denotes the constraint itself. The constraint type is
1253 specified by means of a URI, which defines both the semantic meaning of the constraint as well
1254 as the format of any content.
- 1255 The `GroupTemplate` element has the following properties:
- 1256 • `id`: This attribute specifies the identifier of the Group Template. The identifier of the Group
1257 Template MUST be unique within the target namespace.
 - 1258 • `name`: This optional attribute specifies the name of the Group Template.
 - 1259 • `minInstances`: This integer attribute specifies the minimum number of instances to be created
1260 when instantiating the Group Template. The default value of this attribute is 1. The value of
1261 `minInstances` MUST NOT be less than 0.
 - 1262 • `maxInstances`: This attribute specifies the maximum number of instances that can be created
1263 when instantiating the Group Template. The default value of this attribute is 1. If the string is set
1264 to “unbounded”, an unbounded number of instances can be created. The value of
1265 `maxInstances` MUST be 1 or greater and MUST NOT be less than the value specified for
1266 `minInstances`.
 - 1267 • `NodeTemplate`: This is a node template contained within, or grouped by the Group Template.
 - 1268 • `RelationshipTemplate`: This is a relationship template contained within, or grouped by the
1269 Group.
 - 1270 • `GroupTemplate`: This is a Group Template of a nested group contained within, or grouped by
1271 the Group Template.
 - 1272 • `Policies`: Specifies policies of the Group Template. Each policy is specified by means of a
1273 separate nested `Policy` element. This element contains the actual policy specific content of the
1274 policy.

1275 6.3 Example

1276 The following Service Template defines a Topology Template in-place. The corresponding Topology
1277 Template contains two Node Templates called “MyApplication” and “MyAppServer”. These Node
1278 Templates have the node types “Application” and “ApplicationServer”, respectively, the definitions of
1279 which are imported by the `Import` element. The Node Template “MyApplication” is instantiated exactly
1280 once. Two of its Node Type Properties are initialized by a corresponding `PropertyDefaults`
1281 element. The Node Template “MyAppServer” can be instantiated as many times as needed. The
1282 “MyApplication” Node Template is connected with the “MyAppServer” Node Template via the Relationship
1283 Template named “MyDeploymentRelationship”; the behavior and semantics of the Relationship Template
1284 is defined in the Relationship Type “deployedOn” in the same Service Template document, saying that
1285 “MyApplication” is deployed on “MyAppServer”. When instantiating the “SampleApplication” Topology

1286 Template, instances of "MyApplication" and "MyAppServer" are related by means of corresponding
1287 instances of "MyDeploymentRelationship".

```
1288 <ServiceTemplate id="myService"  
1289     name="My Service"  
1290     targetNamespace="http://www.ibm.com/sample"  
1291     xmlns:abc="http://www.ibm.com/sample">  
1292  
1293     <Import namespace="http://www.ibm.com/sample"  
1294         importType="http://docs.oasis-open.org/tosca/ns/2011/12"/>  
1295  
1296     <TopologyTemplate id="SampleApplication">  
1297  
1298         <NodeTemplate id="MyApplication"  
1299             name="My Application"  
1300             nodeType="abc:Application">  
1301             <PropertyDefaults>  
1302                 <ApplicationProperties>  
1303                     <Owner>Frank</Owner>  
1304                     <InstanceName>Thomas' favorite application</InstanceName>  
1305                 </ApplicationProperties>  
1306             </PropertyDefaults>  
1307         </NodeTemplate/>  
1308  
1309         <NodeTemplate id="MyAppServer"  
1310             name="My Application Server"  
1311             nodeType="abc:ApplicationServer"  
1312             minInstances="0"  
1313             maxInstances="unbounded"/>  
1314  
1315         <RelationshipTemplate id="MyDeploymentRelationship"  
1316             relationshipType="deployedOn">  
1317             <SourceElement id="MyApplication"/>  
1318             <TargetElement id="MyAppServer"/>  
1319         </RelationshipTemplate>  
1320  
1321     </TopologyTemplate>  
1322 </ServiceTemplate>  
1323
```

1324

7 Plans

1325 The operational management behavior of a Service Template is invoked by means of orchestration plans,
1326 or more simply, *Plans*. Plans consist of individual steps (aka tasks or activities) to be performed and the
1327 definition of the potential order of these steps. The execution of a step can be performed by one of the
1328 functions offered via the interfaces of a Node Template, by invoking operations of a Service Template
1329 API, or by invoking other operations being required in the context of a specific service. Plans are
1330 classified by a type, and the following two plan types are defined as part of the TOSCA specification.
1331 *Build plans* specify how instances of their associated Service Templates are made, and *termination plans*
1332 specify how an instance of a Service Template is removed from the environment. Other plan types for
1333 managing existing service instances throughout their life time are termed *modification plans*, and it is
1334 expected that such plan types will be defined subsequently by authors of service templates and domain
1335 expert groups.

1336

7.1 Syntax

1337

1338

1339

1340

1341

1342

1343

1344

1345

1346

1347

1348

1349

1350

1351

1352

1353

1354

1355

1356

1357

```
1 <Plans>
2
3   <Plan id="ID"
4       name="string"?
5       planType="anyURI"
6       languageUsed="anyURI">+
7
8       <PreCondition expressionLanguage="anyURI">?
9         condition
10      </PreCondition>
11
12      ( <PlanModel>
13        actual plan
14      </PlanModel>
15      |
16      <PlanModelReference reference="anyURI"/>
17    )
18
19 </Plan>
20
21 </Plans>
```

1358

7.2 Properties

1359

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

1360

1361

1362

1363

1364

1365

1366

1367

- `id`: This attribute specifies the identifier of the Plan. The identifier of the Plan MUST be unique within the target namespace.
- `name`: This optional attribute specifies the name of the Plan.
- `planType`: The value of the attribute specifies the type of the plan as an indication on what the effect of executing the plan on a service will have. The plan type is specified by means of a URI, allowing for an extensibility mechanism for authors of service templates to define new plan types over time.
The following plan types are defined as part of the TOSCA specification.

- 1368 ○ <http://docs.oasis-open.org/tosca/ns/2011/12/PlanTypes/BuildPlan> - This URI defines the
1369 *build plan* plan type for plans used to initially create a new instance of a service from a
1370 Service Template.
- 1371 ○ <http://docs.oasis-open.org/tosca/ns/2011/12/PlanTypes/TerminationPlan> - This URI
1372 defines the *termination plan* plan type for plans used to terminate the existence of a
1373 service instance.

1374 Note that all other plan types for managing service instances throughout their life time will be
1375 considered and referred to as *modification plans* in general.

- 1376 • `languageUsed`: This attribute denotes the process modeling language (or metamodel) used to
1377 specify the plan. For example, "<http://www.omg.org/spec/BPMN/2.0/>" would specify that BPMN
1378 2.0 has been used to model the plan.

- 1379 • `PreCondition`: This optional element specifies a condition that needs to be satisfied in order
1380 for the plan to be executed. The `expressionLanguage` attribute of this element specifies the
1381 expression language the nested condition is provided in.

1382 Typically, the precondition will be an expression in the instance state attribute of some of the
1383 node templates or relationship templates of the topology template. It will be evaluated based on
1384 the actual values of the corresponding attributes at the time the plan is requested to be executed.
1385 Note, that any other kind of pre-condition is allowed.

- 1386 • `PlanModel`: This property contains the actual model content.
- 1387 • `PlanModelReference`: This property points to the model content. Its reference attribute
1388 contains a URI of the model of the plan.

1389 An instance of the `Plan` element MUST either contain the actual plan as instance of the
1390 `PlanModel` element, or point to the model via the `PlanModelReference` element.
1391

1392 7.3 Use of Process Modeling Languages

1393 TOSCA does not specify a separate metamodel for defining plans. Instead, it is assumed that a process
1394 modelling language (a.k.a. metamodel) like BPEL [BPEL 2.0] or BPMN [BPMN 2.0] is used to define
1395 plans. The specification favours the use of BPMN for modeling plans.

1396 7.4 Example

1397 The following defines two Plans, one Plan for creating a new instance of the "SampleApplication"
1398 Topology Template (the plan is named "DeployApplication"), and one Plan for removing instances of
1399 "SampleApplication". The Plan "DeployApplication" is a build plan specified in BPMN; the process model
1400 is immediately included in the Plan Model (note that the BPMN model is incomplete but used to show the
1401 mechanism of the `PlanModel` element). The Plan can only run when the `PreCondition` "Run only if
1402 funding is available" is satisfied. The Plan "RemoveApplication" is a termination plan specified in BPEL;
1403 the corresponding BPEL definition is defined elsewhere and only referenced by the
1404 `PlanModelReference` element.

```
1405 <Plans>
1406
1407   <Plan id="DeployApplication"
1408     name="Sample Application Build Plan"
1409     planType=
1410       "http://docs.oasis-open.org/tosca/ns/2011/12/PlanTypes/BuildPlan"
1411     languageUsed="http://www.omg.org/spec/BPMN/2.0/">
1412
1413     <PreCondition expressionLanguage="www.example.com/text">?
1414       Run only if funding is available
```

```

1415     </PreCondition>
1416
1417     <PlanModel>
1418         <process name="DeployNewApplication" id="p1">
1419             <documentation>This process deploys a new instance of the
1420                 sample application.
1421             </documentation>
1422
1423             <task id="t1" name="CreateAccount"/>
1424
1425             <task id="t2" name="AcquireNetworkAddresses"
1426                 isSequential="false"
1427                 loopDataInput="t2Input.LoopCounter"/>
1428             <documentation>Assumption: t2 gets data of type "input"
1429                 as input and this data has a field names "LoopCounter"
1430                 that contains the actual multiplicity of the task.
1431             </documentation>
1432
1433             <task id="t3" name="DeployApplicationServer"
1434                 isSequential="false"
1435                 loopDataInput="t3Input.LoopCounter"/>
1436
1437             <task id="t4" name="DeployApplication"
1438                 isSequential="false"
1439                 loopDataInput="t4Input.LoopCounter"/>
1440
1441             <sequenceFlow id="s1" targetRef="t2" sourceRef="t1"/>
1442             <sequenceFlow id="s2" targetRef="t3" sourceRef="t2"/>
1443             <sequenceFlow id="s3" targetRef="t4" sourceRef="t3"/>
1444         </process>
1445     </PlanModel>
1446 </Plan>
1447
1448 <Plan id="RemoveApplication"
1449     planType="http://docs.oasis-
1450     open.org/tosca/ns/2011/12/PlanTypes/TerminationPlan"
1451     languageUsed=
1452     "http://docs.oasis-open.org/wsbpel/2.0/process/executable">
1453     <PlanModelReference reference="prj:RemoveApp"/>
1454 </Plan>
1455
1456 </Plans>
1457

```

1458 8 Cloud Service Archive (CSAR)

1459 This section defines the metadata of a cloud service archive as well as its overall structure.

1460 8.1 Overall Structure of a CSAR

1461 A CSAR is a zip file containing at least two directories, the *TOSCA-Metadata* directory and the *Service-*
1462 *Template* directory. Beyond that, other directories may be contained in a CSAR, i.e. the creator of a
1463 CSAR has all freedom to define the content of a CSAR and the structuring of this content as appropriate
1464 for the cloud application.

1465 The TOSCA-Metadata directory contains metadata describing the other content of the CSAR. This
1466 metadata is referred to as *TOSCA meta file*. This file is named `TOSCA` and has the file extension `.meta`.

1467 The Service-Template directory contains one or more Service Template files (file extension `.ste`). These
1468 Service Template files contain definitions related to the cloud application of the CSAR. One of these
1469 Service Template files is distinguished as *entry Service Template*, i.e. it contains the definition of the
1470 structure and behavior of the cloud application, while the other Service Template files contain definitions
1471 that are referenced by the entry Service Template.

1472 8.2 TOSCA Meta File

1473 The TOSCA meta file includes metadata that allows interpreting the various artifacts within the CSAR
1474 properly. The `TOSCA.meta` file is contained in the `TOSCA-Metadata` directory of the CSAR.

1475 A TOSCA meta file consists of name/value pairs. The name-part of a name/value pair is followed by a
1476 colon, followed by a blank, followed by the value-part of the name/value pair. The name MUST not
1477 contain a colon. Values that represent binary data must be base64 encoded. Values that extend beyond
1478 one line can be spread over multiple lines if each subsequent line starts with at least one space. Such
1479 spaces are then collapsed when the value string is read.

```
1480 <name>: <value>
```

1481 Each name/value pair is in a separate line. A list of related name/value pairs, i.e. a list of consecutive
1482 name/value pairs describing a particular file in a CSAR, is called a *block*. Blocks are separated by an
1483 empty line. The first block, called *block_0*, is metadata about the CSAR itself. All other blocks represent
1484 metadata of files in the CSAR.

1485 The structure of `block_0` in the TOSCA meta file is as follows:

```
1486 TOSCA-Meta-File-Version: digit.digit  
1487 CSAR-Version: digit.digit  
1488 Created-By: string  
1489 Entry-Service-Template: filename
```

1490 The name/value pairs are as follows:

- 1491 • `TOSCA-Meta-File-Version`: This is the version number of the TOSCA meta file format.
1492 The value must be “1.0” in the current version of the TOSCA specification.
- 1493 • `CSAR-Version`: This is the version number of the CSAR specification. The value must be “1.0”
1494 in the current version of the TOSCA specification.
- 1495 • `Created-By`: The person or vendor, respectively, who created the CSAR.
- 1496 • `Entry-Service-Template`: The service template from the Service-Template directory of
1497 the CSAR that is the entry point for the overall cloud application.

1498 Note, that a CSAR may contain multiple Service Template files. One reason for this is
1499 completeness, e.g. a Service Template containing Node Types referred to by the entry Service
1500 Template might be included in the Service-Template directory to avoid importing it from external
1501 locations.

1502 The first line of a block (other than block_0) must be a name/value pair that has the name “Name” and the
1503 value of which is the path-name of the file described. The second line must be a name/value pair that has
1504 the name “Content-Type” describing the type of the file described; the format is that of a MIME type with
1505 type/subtype structure. The other name/value pairs that consecutively follow are file-type specific.

```
1506 Name: <path-name_1>  
1507 Content-Type: type_1/subtype_1  
1508 <name_11>: <value_11>  
1509 <name_12>: <value_12>  
1510 ...  
1511 <name_1n>: <value_1n>  
1512 ...  
1513 ...  
1514 ...  
1515 Name: <path-name_k>  
1516 Content-Type: type_k/subtype_k  
1517 <name_k1>: <value_k1>  
1518 <name_k2>: <value_k2>  
1519 ...  
1520 <name_km>: <value_km>
```

1521 The name/value pairs are as follows:

- 1522 • Name: The pathname or pathname pattern of the file(s) or resources described within the actual
1523 CSAR.

1524 Note, that the file located at this location may basically contain a reference to an external file.
1525 Such a reference is given by a URI that is of one of the URL schemes “file”, “http”, or “https”.

- 1526 • Content-Type: The type of the file described. This type is a MIME type complying with the
1527 type/subtype structure. Vendor defined subtypes should start as usual with the string “vnd.”.
1528

1529 Note that later directives override earlier directives. This allows for specifying global default directives that
1530 can be specialized by later directorives in the TOSCA meta file.

1531 8.3 Example

1532 Figure 5 depicts a sample Service Template of an application, named `Payroll.ste`. The application is
1533 a payroll application written in Java that must be deployed on a proper application server. The Service
1534 Template of the application defines the Node Template `Payroll Application`, the Node Template
1535 `Application Server`, as well as the Relationship Template `deployed_on`. The `Payroll`
1536 `Application` is associated with an EAR file (named `Payroll.ear`) which is provided as
1537 corresponding Deployment Artifact of the `Payroll Application` Node Template. An Amazon
1538 Machine Image (AMI) is the Deployment Artifact of the `Application Server` Node Template; this
1539 Deployment Artifact is a reference to the image in the Amazon EC2 environment. The Implementation
1540 Artifacts of some operations of the Node Templates are provided too; for example, the `start` operation
1541 of the `Payroll Application` is implemented by a Java API supported by the `payrolladm.jar`
1542 file, the `installApp` operation of the `Application Server` is realized by the Python script
1543 `wsadmin.py`, while the `runInstances` operation is a REST API available at Amazon for running
1544 instances of an AMI. Note, that the `runInstances` operation is not related to a particular
1545 implementation artifact because it is available as an Amazon Web Service
1546 (<https://ec2.amazonaws.com/?Action=RunInstances>); but the details of this REST API are specified with
1547 the operation of the `Application Server` Node Type.

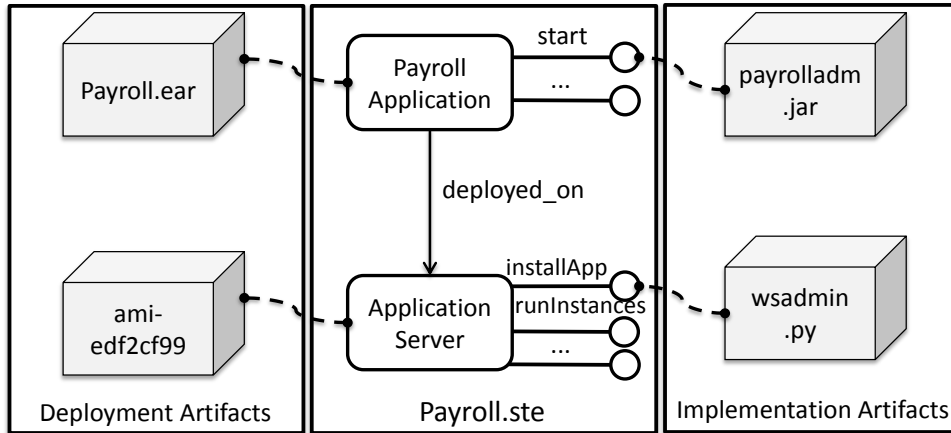


Figure 5: Sample Service Template

The corresponding Node Types and Relationship Types have been defined in the `PayrollTypes.ste` document, which is imported by the `Payroll` Service Template. The following listing provides some of the details:

```

1548 <ServiceTemplate id="Payroll"
1549     targetNamespace="http://www.example.com/ste"
1550     xmlns:pay="http://www.example.com/ste/Types">
1551
1552   <Import namespace="http://www.example.com/ste/Types"
1553     location="http://www.example.com/ste/Types/PayrollTypes.ste"
1554     importType=" http://docs.oasis-open.org/ns/tosca/2011/12"/>
1555
1556   <Types>
1557     ...
1558   </Types>
1559
1560   <TopologyTemplate ID="PayrollTemplate">
1561
1562     <NodeTemplate id="Payroll Application"
1563       nodeType="pay:ApplicationNodeType">
1564       ...
1565
1566       <DeploymentArtifacts>
1567         <DeploymentArtifact name="PayrollEAR"
1568           type="http://www.example.com/
1569             ns/tosca/2011/12/
1570             DeploymentArtifactTypes/CSARref">
1571           EARs/Payroll.ear
1572         </DeploymentArtifact>
1573       </DeploymentArtifacts>
1574
1575       <ImplementationArtifacts>
1576         <ImplementationArtifact operationName="start"
1577           type="http://www.example.com/
1578             ns/tosca/2011/12/
1579             ImplementationArtifactTypes/CSARref">
1580           JARs/payrolladm.jar
1581         </ImplementationArtifact>
1582       </ImplementationArtifacts>
1583
1584     </NodeTemplate>
1585
1586   </TopologyTemplate>
1587
1588 </ServiceTemplate>
1589
1590

```

```

1591 <NodeTemplate id="Application Server"
1592         nodeType="pay:ApplicationServerNodeType">
1593     ...
1594
1595     <DeploymentArtifacts>
1596         <DeploymentArtifact name="ApplicationServerImage"
1597             type="http://www.example.com/
1598                 ns/tosca/2011/12/
1599                 DeploymentArtifactTypes/AMIref">
1600             ami-edf2cf99
1601         </DeploymentArtifact>
1602     </DeploymentArtifacts>
1603
1604     <ImplementationArtifacts>
1605         <ImplementationArtifact operationName="installApp"
1606             type="http://www.example.com/
1607                 ns/tosca/2011/12/
1608                 ImplementationArtifactTypes/CSARref">
1609             Python/wsadmin.py
1610         </ImplementationArtifact>
1611     </ImplementationArtifacts>
1612
1613 </NodeTemplate>
1614
1615 <RelationshipTemplate id="deployed_on"
1616     relationshipType="pay:deployed_on">
1617     <SourceElement id="Payroll Application"/>
1618     <TargetElement id="Application Server"/>
1619 </RelationshipTemplate>
1620
1621 </TopologyTemplate>
1622
1623 </ServiceTemplate>

```

1624

1625 The Payroll Application Node Template specifies the deployment artifact PayrollEAR. It is a
1626 reference to the CSAR containing the Payroll.ste file, which is indicated by the .../CSARref type
1627 of the DeploymentArtifact element. The type specific content is a path expression in the directory
1628 structure of the CSAR: it points to the Payroll.ear file in the EARs directory of the CSAR (see Figure
1629 6 for the structure of the corresponding CSAR).

1630 The Payroll Application Node Template also contains an ImplementationArtifact
1631 element. This element contains information about the implementation of the start operation by pointing
1632 to the payrolladm.jar file in the JARs directory of the CSAR.

1633 The Application Server Node Template has a DeploymentArtifact called
1634 ApplicationServerImage that is a reference to an AMI (Amazon Machine Image), indicated by an
1635 .../AMIref type. It provides a Python script (the wsadmin.py file in the Python directory of the
1636 CSAR) as implementation of the install operation; the type of the implementation artifact is again a
1637 CSAR reference.

1638 The corresponding CSAR has the following structure (see Figure 6): The TOSCA.meta file is contained
1639 in the TOSCA-Metadata directory. The Payroll.ste file itself is contained in the Service-
1640 Template directory. Also, the PayrollTypes.ste file is in this directory. The content of the other
1641 directories has been sketched before.



Figure 6: Structure of CSAR Sample

1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667

The TOSCA.meta file is as follows:

```

TOSCA-Meta-Version: 1.0
CSAR-Version: 1.0
Created-By: Frank
Entry-Service-Template: /Service-Template/Payroll.ste

Name: Service-Template/Payroll.ste
Content-Type: application/vnd.oasis.service_template

Name: Service-Template/PayrollTypes.ste
Content-Type: application/vnd.oasis.service_template

Name: Plans/AddUser.bpmn
Content-Type: application/vnd.oasis.bpmn

Name: EARs/Payroll.ear
Content-Type: application/vnd.oasis.ear

Name: JARs/Payrolladm.jar
Content-Type: application/vnd.oasis.jar

Name: Python/wsadmin.py
Content-Type: application/vnd.oasis.py

```

1668 **9 Security Considerations**

1669 TOSCA does not mandate the use of any specific mechanism or technology for client authentication.
1670 However, a client **MUST** provide a principal or the principal **MUST** be obtainable by the infrastructure.

1671 **10 Conformance**

1672 **This section is to be done.**

1673

Appendix A. Portability and Interoperability Considerations

1674

1675 This section illustrates the portability and interoperability aspects addressed by Service Templates:

1676 Portability - The ability to take Service Templates created in one vendor's environment and use them in
1677 another vendor's environment.

1678 Interoperability - The capability for multiple components (e.g. a task of a plan and the definition of a
1679 topology node) to interact using well-defined messages and protocols. This enables combining
1680 components from different vendors allowing seamless management of services.

1681 Portability demands support of TOSCA elements.

1682

Appendix B. Acknowledgements

1683 The following individuals have participated in the creation of this specification and are gratefully
1684 acknowledged.

1685 **Participants:**

Aaron Zhang	Huawei Technologies Co., Ltd.
Adolf Hohl	NetApp
Afkham Azeez	WSO2
Alex Heneveld	Cloudsoft Corporation Limited
Allen Bannon	SAP AG
Anthony Rutkowski	Yaana Technologies, LLC
Arvind Srinivasan	IBM
Bryan Haynie	VCE
Celso Rodriguez	ASG Software Solutions
Chandrasekhar Sundaresh	CA Technologies
Charith Wickramarachchi	WSO2
Colin Hopkinson	3M HIS
Dale Moberg	Axway Software
Debojyoti Dutta	Cisco Systems
Dee Schur	OASIS
Denis Nothern	CenturyLink
Denis Weerasiri	WSO2
Derek Palma	Vnomic
Dhiraj Pathak	PricewaterhouseCoopers LLP:
Diane Mueller	ActiveState Software, Inc.
Doug Davis	IBM
Duncan Johnston-Watt	Cloudsoft Corporation Limited
Efraim Moscovich	CA Technologies
Frank Leymann	IBM
Gerd Breiter	IBM
James Thomason	Gale Technologies
Jan Ignatius	Nokia Siemens Networks GmbH & Co. KG
Jim Marino	Individual
John Wilmes	Progress Software
Joseph Malek	VCE
Kevin Poulter	SAP AG
Koert Struijk	CA Technologies
Lee Thompson	Morphlabs, Inc.
Marvin Waschke	CA Technologies
Mascot Yu	Huawei Technologies Co., Ltd.
Matthew Dovey	JISC Executive, University of Bristol
Matthew Rutkowski	IBM
Michael Schuster	SAP AG
Mike Edwards	IBM
Naveen Joy	Cisco Systems
Nikki Heron	rPath, Inc.
Pascal Vitoux	ASG Software Solutions
Paul Fremantle	WSO2
Paul Lipton	CA Technologies

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

1686

1687

Appendix C. Complete TOSCA Grammar

1689 **Note:** The following is a pseudo EBNF grammar notation meant for documentation purposes only. The
1690 grammar is not intended for machine processing.

```

1691 1 <ServiceTemplate id="ID"
1692 2     name="string"?
1693 3     targetNamespace="anyURI">
1694 4
1695 5     <Extensions>?
1696 6         <Extension namespace="anyURI"
1697 7             mustUnderstand="yes|no"?/>>+
1698 8     </Extensions>
1699 9
1700 10    <Import namespace="anyURI"?
1701 11        location="anyURI"?
1702 12        importType="anyURI"/>*
1703 13
1704 14    <Types>?
1705 15        <xs:schema .../>*
1706 16    </Types>
1707 17
1708 18    (
1709 19    <TopologyTemplateReference reference="QName"/>
1710 20    |
1711 21    <TopologyTemplate id="ID"
1712 22        name="string"?>
1713 23
1714 24        (
1715 25        <NodeTemplate id="ID"
1716 26            name="string"?
1717 27            nodeType="QName"
1718 28            minInstances="int"?
1719 29            maxInstances="int|string"?>
1720 30
1721 31            <PropertyDefaults>?
1722 32                XML fragment
1723 33            </PropertyDefaults>
1724 34
1725 35            <PropertyConstraints>?
1726 36
1727 37                <PropertyConstraint property="string"
1728 38                    constraintType="anyURI">+
1729 39                    constraint?
1730 40                </PropertyConstraint>
1731 41
1732 42            </PropertyConstraints>
1733 43
1734 44            <Policies>?
1735 45                <Policy name="string" type="anyURI">+
1736 46                    policy specific content
1737 47                </Policy>
1738 48            </Policies>
1739 49
1740 50            <EnvironmentConstraints>?
1741 51            <EnvironmentConstraint constraintType="anyURI">+

```

```

1742 52         constraint type specific content?
1743 53     </EnvironmentConstraint>
1744 54 </EnvironmentConstraints>
1745 55
1746 56 <DeploymentArtifacts>?
1747 57     <DeploymentArtifact name="string" type="anyURI">+
1748 58         artifact specific content
1749 59     </DeploymentArtifact>
1750 60 </DeploymentArtifacts>
1751 61
1752 62 <ImplementationArtifacts>?
1753 63     <ImplementationArtifact operationName="string"
1754 64         type="anyURI">+
1755 65         <RequiredContainerCapabilities>?
1756 66             <RequiredContainerCapability capability="anyURI"/>+
1757 67         </RequiredContainerCapabilities>
1758 68         artifact specific content
1759 69     <ImplementationArtifact>
1760 70 </ImplementationArtifacts>
1761 71
1762 72 </NodeTemplate>
1763 73 |
1764 74 <RelationshipTemplate id="ID"
1765 75     name="string"?
1766 76     relationshipType="QName">+
1767 77
1768 78     <SourceElement id="IDREF"/>
1769 79
1770 80     ( <TargetElement id="IDREF"/>
1771 81     |
1772 82     <TargetElementReference id="QName"/>
1773 83     )
1774 84
1775 85     <PropertyDefaults>?
1776 86         XML fragment
1777 87     </PropertyDefaults>
1778 88
1779 89     <PropertyConstraints>?
1780 90
1781 91         <PropertyConstraint property="string"
1782 92             constraintType="anyURI">+
1783 93             constraint?
1784 94         </PropertyConstraint>
1785 95
1786 96     </PropertyConstraints>
1787 97
1788 98     <RelationshipConstraints>?
1789 99
1790 100         <RelationshipConstraint constraintType="anyURI">+
1791 101             constraint?
1792 102         </RelationshipConstraint>
1793 103
1794 104     </RelationshipConstraints>
1795 105
1796 106 </RelationshipTemplate>
1797 107 |
1798 108 <GroupTemplate id="ID"
1799 109     name="string"?

```

```

1800 110             minInstances="int"?
1801 111             maxInstances="int|string"?>
1802 112
1803 113         (
1804 114             <NodeTemplate ... />
1805 115         |
1806 116             <RelationshipTemplate ... />
1807 117         |
1808 118             <GroupTemplate ... />
1809 119         )+
1810 120
1811 121         <Policies>?
1812 122             <Policy name="string" type="anyURI">+
1813 123                 policy specific content
1814 124             </Policy>
1815 125         </Policies>
1816 126
1817 127         </GroupTemplate>
1818 128     )+
1819 129
1820 130 </TopologyTemplate>
1821 131 )?
1822 132
1823 133 <NodeTypes>?
1824 134
1825 135     <NodeType id="ID"
1826 136         name="string"?
1827 137         abstract="yes|no"?
1828 138         final="yes|no"?>+
1829 139
1830 140     <NodeTypeProperties element="QName"?
1831 141         type="QName"?/>?
1832 142
1833 143     <DerivedFrom nodeTypeRef="QName"/>?
1834 144
1835 145     <InstanceStates>?
1836 146         <InstanceState state="anyURI">+
1837 147     </InstanceStates>
1838 148
1839 149     <Interfaces>?
1840 150
1841 151         <Interface name="NCName | anyURI">+
1842 152
1843 153             <Operation name="NCName">*
1844 154
1845 155             (
1846 156                 <WSDL portType="QName"
1847 157                     operation="NCName"/>
1848 158             |
1849 159                 <REST method="GET | PUT | POST | DELETE"
1850 160                     abs_path="anyURI"?
1851 161                     absoluteURI="anyURI"?
1852 162                     requestBody="QName"?
1853 163                     responseBody="QName"?>
1854 164
1855 165             <Parameters>?
1856 166                 <Parameter name="string" required="yes|no"/>+
1857 167             </Parameters>

```

```

1858 168
1859 169         <Headers>?
1860 170             <Header name="string" required="yes|no"/>+
1861 171         </Headers>
1862 172
1863 173         </REST>
1864 174     |
1865 175         <ScriptOperation>
1866 176
1867 177             <InputParameters>?
1868 178
1869 179                 <InputParamter name="string"
1870 180                     type="string"
1871 181                     required="yes|no"/>+
1872 182
1873 183             </InputParameters>
1874 184
1875 185             <OutputParameters>?
1876 186
1877 187                 <OutputParamter name="string"
1878 188                     type="string"
1879 189                     required="yes|no"/>+
1880 190
1881 191             </OutputParameters>
1882 192
1883 193         </ScriptOperation>
1884 194     )
1885 195
1886 196 </Operation>
1887 197
1888 198 <ImplementationArtifacts>?
1889 199
1890 200     <ImplementationArtifact operationName="string"?
1891 201         type="anyURI">+
1892 202
1893 203     <RequiredContainerCapabilities>?
1894 204         <RequiredContainerCapability capability="anyURI"/>+
1895 205     </RequiredContainerCapabilities>
1896 206
1897 207         artifact specific content
1898 208
1899 209     <ImplementationArtifact>
1900 210
1901 211 </ImplementationArtifacts>
1902 212
1903 213 </Interface>
1904 214
1905 215 </Interfaces>
1906 216
1907 217 <DeploymentArtifacts>?
1908 218     <DeploymentArtifact name="string" type="anyURI">+
1909 219         artifact specific content
1910 220     </DeploymentArtifact>
1911 221 </DeploymentArtifacts>
1912 222
1913 223
1914 224 <Policies>?
1915 225

```

```

1916 226     <Policy name="string" type="anyURI">+
1917 227         policy specific content
1918 228     </Policy>
1919 229
1920 230     </Policies>
1921 231
1922 232 </NodeType>
1923 233
1924 234 </NodeTypes>
1925 235
1926 236 <RelationshipTypes>?
1927 237
1928 238     <RelationshipType id="ID"
1929 239         name="string"?
1930 240         abstract="yes|no"?
1931 241         final="yes|no"?
1932 242         cascadingDeletion="yes|no"?>+
1933 243
1934 244     <RelationshipTypeProperties element="QName"?
1935 245         type="QName"?/>?
1936 246
1937 247     <InstanceStates>?
1938 248         <InstanceState state="anyURI">+
1939 249     </InstanceStates>
1940 250
1941 251     <SourceInterfaces>?
1942 252         <Interface name="NCName | anyURI">+
1943 253         ...
1944 254     </Interface>
1945 255 </SourceInterfaces>
1946 256
1947 257     <TargetInterfaces>?
1948 258         <Interface name="NCName | anyURI">+
1949 259         ...
1950 260     </Interface>
1951 261 </TargetInterfaces>
1952 262
1953 263     <ValidSource typeRef="QName"/>?
1954 264
1955 265     <ValidTarget typeRef="QName"/>?
1956 266
1957 267 </RelationshipType>
1958 268
1959 269 </RelationshipTypes>
1960 270
1961 271 <Plans>?
1962 272
1963 273     <Plan id="ID"
1964 274         name="string"?
1965 275         planType="anyURI"
1966 276         languageUsed="anyURI">+
1967 277
1968 278     <PreCondition expressionLanguage="anyURI">?
1969 279         condition
1970 280 </PreCondition>
1971 281
1972 282     ( <PlanModel>
1973 283         actual plan

```

```
1974 284     </PlanModel>
1975 285     |
1976 286     <PlanModelReference reference="anyURI"/>
1977 287     )
1978 288
1979 289     </Plan>
1980 290
1981 291     </Plans>
1982 292
1983 293 </ServiceTemplate>
```


1984

Appendix D. TOSCA Schema

1985 TOSCA-v1.0.xsd:

```
1986 1 <?xml version="1.0" encoding="UTF-8"?>
1987 2 <xs:schema targetNamespace="http://docs.oasis-open.org/tosca/ns/2011/12"
1988 3   elementFormDefault="qualified" attributeFormDefault="unqualified"
1989 4   xmlns="http://docs.oasis-open.org/tosca/ns/2011/12"
1990 5   xmlns:xs="http://www.w3.org/2001/XMLSchema">
1991 6
1992 7   <xs:import namespace="http://www.w3.org/XML/1998/namespace"
1993 8     schemaLocation="http://www.w3.org/2001/xml.xsd"/>
1994 9
1995 10  <xs:element name="documentation" type="tDocumentation"/>
1996 11  <xs:complexType name="tDocumentation" mixed="true">
1997 12    <xs:sequence>
1998 13      <xs:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
1999 14    </xs:sequence>
2000 15    <xs:attribute name="source" type="xs:anyURI"/>
2001 16    <xs:attribute ref="xml:lang"/>
2002 17  </xs:complexType>
2003 18
2004 19  <xs:complexType name="tExtensibleElements">
2005 20    <xs:sequence>
2006 21      <xs:element ref="documentation" minOccurs="0" maxOccurs="unbounded"/>
2007 22      <xs:any namespace="##other" processContents="lax" minOccurs="0"
2008 23        maxOccurs="unbounded"/>
2009 24    </xs:sequence>
2010 25    <xs:anyAttribute namespace="##other" processContents="lax"/>
2011 26  </xs:complexType>
2012 27
2013 28  <xs:complexType name="tImport">
2014 29    <xs:complexContent>
2015 30      <xs:extension base="tExtensibleElements">
2016 31        <xs:attribute name="namespace" type="xs:anyURI"/>
2017 32        <xs:attribute name="location" type="xs:anyURI"/>
2018 33        <xs:attribute name="importType" type="importedURI" use="required"/>
2019 34      </xs:extension>
2020 35    </xs:complexContent>
2021 36  </xs:complexType>
2022 37
2023 38  <xs:element name="ServiceTemplate">
2024 39    <xs:complexType>
2025 40      <xs:complexContent>
2026 41        <xs:extension base="tServiceTemplate"/>
2027 42      </xs:complexContent>
2028 43    </xs:complexType>
2029 44  </xs:element>
2030 45
2031 46  <xs:complexType name="tServiceTemplate">
2032 47    <xs:complexContent>
2033 48      <xs:extension base="tExtensibleElements">
2034 49        <xs:sequence>
2035 50          <xs:element name="Import" type="tImport" minOccurs="0"
2036 51            maxOccurs="unbounded"/>
2037 52          <xs:element name="Types" minOccurs="0"/>
```

```

2038 53     <xs:complexType>
2039 54     <xs:sequence>
2040 55         <xs:any namespace="##other" processContents="lax" minOccurs="0"
2041 56             maxOccurs="unbounded"/>
2042 57     </xs:sequence>
2043 58 </xs:complexType>
2044 59 </xs:element>
2045 60 <xs:element name="Extensions" minOccurs="0">
2046 61     <xs:complexType>
2047 62         <xs:sequence>
2048 63             <xs:element name="Extension" type="tExtension"
2049 64                 maxOccurs="unbounded"/>
2050 65         </xs:sequence>
2051 66     </xs:complexType>
2052 67 </xs:element>
2053 68 <xs:choice minOccurs="0">
2054 69     <xs:element name="TopologyTemplateReference">
2055 70         <xs:complexType>
2056 71             <xs:attribute name="reference" type="xs:QName"/>
2057 72         </xs:complexType>
2058 73     </xs:element>
2059 74     <xs:element name="TopologyTemplate" type="tTopologyTemplate"/>
2060 75 </xs:choice>
2061 76 <xs:element name="NodeTypes" type="tNodeTypes" minOccurs="0"/>
2062 77 <xs:element name="RelationshipTypes" type="tRelationshipTypes"
2063 78     minOccurs="0"/>
2064 79 <xs:element name="Plans" type="tPlans" minOccurs="0"/>
2065 80 </xs:sequence>
2066 81 <xs:attribute name="id" type="xs:ID" use="required"/>
2067 82 <xs:attribute name="name" type="xs:string" use="optional"/>
2068 83 <xs:attribute name="targetNamespace" type="xs:anyURI"/>
2069 84 </xs:extension>
2070 85 </xs:complexContent>
2071 86 </xs:complexType>
2072 87
2073 88 <xs:complexType name="tDeploymentArtifact">
2074 89     <xs:complexContent>
2075 90         <xs:extension base="tExtensibleElements">
2076 91             <xs:attribute name="name" type="xs:string" use="required"/>
2077 92             <xs:attribute name="type" type="xs:anyURI" use="required"/>
2078 93         </xs:extension>
2079 94     </xs:complexContent>
2080 95 </xs:complexType>
2081 96
2082 97 <xs:element name="NodeTemplate" type="tNodeTemplate"/>
2083 98 <xs:complexType name="tNodeTemplate">
2084 99     <xs:complexContent>
2085 100         <xs:extension base="tExtensibleElements">
2086 101             <xs:sequence>
2087 102                 <xs:element name="PropertyDefaults" minOccurs="0">
2088 103                     <xs:complexType>
2089 104                         <xs:sequence>
2090 105                             <xs:any namespace="##other" processContents="lax"/>
2091 106                         </xs:sequence>
2092 107                     </xs:complexType>
2093 108                 </xs:element>
2094 109                 <xs:element name="PropertyConstraints" minOccurs="0">
2095 110                     <xs:complexType>

```

```

2096 111     <xs:sequence>
2097 112         <xs:element name="PropertyConstraint"
2098 113             type="tPropertyConstraint" maxOccurs="unbounded"/>
2099 114     </xs:sequence>
2100 115 </xs:complexType>
2101 116 </xs:element>
2102 117 <xs:element name="Policies" minOccurs="0">
2103 118     <xs:complexType>
2104 119         <xs:sequence>
2105 120             <xs:element name="Policy" type="tPolicy"
2106 121                 maxOccurs="unbounded"/>
2107 122         </xs:sequence>
2108 123     </xs:complexType>
2109 124 </xs:element>
2110 125 <xs:element name="EnvironmentConstraints" minOccurs="0">
2111 126     <xs:complexType>
2112 127         <xs:sequence>
2113 128             <xs:element name="EnvironmentConstraint"
2114 129                 type="tEnvironmentConstraint" maxOccurs="unbounded"/>
2115 130         </xs:sequence>
2116 131     </xs:complexType>
2117 132 </xs:element>
2118 133 <xs:element name="DeploymentArtifacts" minOccurs="0">
2119 134     <xs:complexType>
2120 135         <xs:sequence>
2121 136             <xs:element name="DeploymentArtifact"
2122 137                 type="tDeploymentArtifact" maxOccurs="unbounded"/>
2123 138         </xs:sequence>
2124 139     </xs:complexType>
2125 140 </xs:element>
2126 141 <xs:element name="ImplementationArtifacts" minOccurs="0">
2127 142     <xs:complexType>
2128 143         <xs:sequence>
2129 144             <xs:element name="ImplementationArtifact"
2130 145                 type="tImplementationArtifact" maxOccurs="unbounded"/>
2131 146         </xs:sequence>
2132 147     </xs:complexType>
2133 148 </xs:element>
2134 149 </xs:sequence>
2135 150 <xs:attribute name="id" type="xs:ID" use="required"/>
2136 151 <xs:attribute name="name" type="xs:string" use="optional"/>
2137 152 <xs:attribute name="nodeType" type="xs:QName" use="required"/>
2138 153 <xs:attribute name="minInstances" type="xs:int" use="optional"
2139 154     default="1"/>
2140 155 <xs:attribute name="maxInstances" use="optional" default="1">
2141 156     <xs:simpleType>
2142 157         <xs:union>
2143 158             <xs:simpleType>
2144 159                 <xs:restriction base="xs:nonNegativeInteger">
2145 160                     <xs:pattern value="([1-9]+[0-9]*)"/>
2146 161                 </xs:restriction>
2147 162             </xs:simpleType>
2148 163             <xs:simpleType>
2149 164                 <xs:restriction base="xs:string">
2150 165                     <xs:enumeration value="unbounded"/>
2151 166                 </xs:restriction>
2152 167             </xs:simpleType>
2153 168         </xs:union>

```

```

2154 169     </xs:simpleType>
2155 170     </xs:attribute>
2156 171     </xs:extension>
2157 172     </xs:complexContent>
2158 173 </xs:complexType>
2159 174
2160 175 <xs:complexType name="tPropertyConstraint">
2161 176   <xs:sequence>
2162 177     <xs:any namespace="##other" processContents="lax" minOccurs="0"/>
2163 178   </xs:sequence>
2164 179   <xs:attribute name="property" type="xs:string" use="required"/>
2165 180   <xs:attribute name="constraintType" type="xs:anyURI" use="required"/>
2166 181 </xs:complexType>
2167 182
2168 183 <xs:element name="TopologyTemplate" type="tTopologyTemplate"/>
2169 184 <xs:complexType name="tTopologyTemplate">
2170 185   <xs:complexContent>
2171 186     <xs:extension base="tTopologyElementCollection"/>
2172 187   </xs:complexContent>
2173 188 </xs:complexType>
2174 189
2175 190 <xs:element name="GroupTemplate" type="tGroupTemplate"/>
2176 191 <xs:complexType name="tGroupTemplate">
2177 192   <xs:complexContent>
2178 193     <xs:extension base="tTopologyElementCollection">
2179 194       <xs:sequence>
2180 195         <xs:element name="Policies" minOccurs="0">
2181 196           <xs:complexType>
2182 197             <xs:sequence>
2183 198               <xs:element name="Policy" type="tPolicy"
2184 199                 maxOccurs="unbounded"/>
2185 200             </xs:sequence>
2186 201           </xs:complexType>
2187 202         </xs:element>
2188 203       </xs:sequence>
2189 204       <xs:attribute name="minInstances" type="xs:int" use="optional"
2190 205         default="1"/>
2191 206       <xs:attribute name="maxInstances" use="optional" default="1">
2192 207         <xs:simpleType>
2193 208           <xs:union>
2194 209             <xs:simpleType>
2195 210               <xs:restriction base="xs:nonNegativeInteger">
2196 211                 <xs:pattern value="([1-9]+[0-9]*)"/>
2197 212               </xs:restriction>
2198 213             </xs:simpleType>
2199 214             <xs:simpleType>
2200 215               <xs:restriction base="xs:string">
2201 216                 <xs:enumeration value="unbounded"/>
2202 217               </xs:restriction>
2203 218             </xs:simpleType>
2204 219           </xs:union>
2205 220         </xs:simpleType>
2206 221       </xs:attribute>
2207 222     </xs:extension>
2208 223   </xs:complexContent>
2209 224 </xs:complexType>
2210 225
2211 226 <xs:complexType name="tTopologyElementCollection">

```

```

2212 227     <xs:complexContent>
2213 228     <xs:extension base="tExtensibleElements">
2214 229     <xs:choice maxOccurs="unbounded">
2215 230     <xs:element name="NodeTemplate" type="tNodeTemplate"/>
2216 231     <xs:element name="RelationshipTemplate"
2217 232     type="tRelationshipTemplate"/>
2218 233     <xs:element name="GroupTemplate" type="tGroupTemplate"/>
2219 234     </xs:choice>
2220 235     <xs:attribute name="id" type="xs:ID" use="required"/>
2221 236     <xs:attribute name="name" type="xs:string" use="optional"/>
2222 237     <xs:attribute name="targetNamespace" type="xs:anyURI"/>
2223 238     </xs:extension>
2224 239 </xs:complexContent>
2225 240 </xs:complexType>
2226 241
2227 242 <xs:element name="RelationshipTypes" type="tRelationshipTypes"/>
2228 243 <xs:complexType name="tRelationshipTypes">
2229 244 <xs:sequence>
2230 245 <xs:element name="RelationshipType" type="tRelationshipType"
2231 246 maxOccurs="unbounded"/>
2232 247 </xs:sequence>
2233 248 <xs:attribute name="targetNamespace" type="xs:anyURI"/>
2234 249 </xs:complexType>
2235 250
2236 251 <xs:element name="RelationshipType" type="tRelationshipType"/>
2237 252 <xs:complexType name="tRelationshipType">
2238 253 <xs:complexContent>
2239 254 <xs:extension base="tExtensibleElements">
2240 255 <xs:sequence>
2241 256 <xs:element name="RelationshipTypeProperties" minOccurs="0">
2242 257 <xs:complexType>
2243 258 <xs:attribute name="element" type="xs:QName"/>
2244 259 <xs:attribute name="type" type="xs:QName"/>
2245 260 </xs:complexType>
2246 261 </xs:element>
2247 262 <xs:element name="DerivedFrom" minOccurs="0">
2248 263 <xs:complexType>
2249 264 <xs:attribute name="relationshipTypeRef" type="xs:QName"
2250 265 use="required"/>
2251 266 </xs:complexType>
2252 267 </xs:element>
2253 268 <xs:element name="InstanceStates"
2254 269 type="tTopologyElementInstanceStates" minOccurs="0"/>
2255 270 <xs:element name="SourceInterfaces" minOccurs="0">
2256 271 <xs:complexType>
2257 272 <xs:sequence>
2258 273 <xs:element name="Interface" type="tInterface"
2259 274 maxOccurs="unbounded"/>
2260 275 </xs:sequence>
2261 276 </xs:complexType>
2262 277 </xs:element>
2263 278 <xs:element name="TargetInterfaces" minOccurs="0">
2264 279 <xs:complexType>
2265 280 <xs:sequence>
2266 281 <xs:element name="Interface" type="tInterface"
2267 282 maxOccurs="unbounded"/>
2268 283 </xs:sequence>
2269 284 </xs:complexType>

```

```

2270 285     </xs:element>
2271 286     <xs:element name="ValidSource" minOccurs="0">
2272 287       <xs:complexType>
2273 288         <xs:attribute name="typeRef" type="xs:QName" use="required"/>
2274 289       </xs:complexType>
2275 290     </xs:element>
2276 291     <xs:element name="ValidTarget" minOccurs="0">
2277 292       <xs:complexType>
2278 293         <xs:attribute name="typeRef" type="xs:QName" use="required"/>
2279 294       </xs:complexType>
2280 295     </xs:element>
2281 296   </xs:sequence>
2282 297   <xs:attribute name="id" type="xs:ID" use="required"/>
2283 298   <xs:attribute name="name" type="xs:string" use="optional"/>
2284 299   <xs:attribute name="cascadingDeletion" type="tBoolean"
2285 300     use="optional" default="no"/>
2286 301   <xs:attribute name="targetNamespace" type="xs:anyURI"/>
2287 302   <xs:attribute name="abstract" type="tBoolean" default="no"/>
2288 303   <xs:attribute name="final" type="tBoolean" default="no"/>
2289 304 </xs:extension>
2290 305 </xs:complexContent>
2291 306 </xs:complexType>
2292 307
2293 308 <xs:element name="RelationshipTemplate" type="tRelationshipTemplate"/>
2294 309 <xs:complexType name="tRelationshipTemplate">
2295 310   <xs:complexContent>
2296 311     <xs:extension base="tExtensibleElements">
2297 312       <xs:sequence>
2298 313         <xs:element name="SourceElement">
2299 314           <xs:complexType>
2300 315             <xs:attribute name="id" type="xs:IDREF" use="required"/>
2301 316           </xs:complexType>
2302 317         </xs:element>
2303 318         <xs:choice>
2304 319           <xs:element name="TargetElement">
2305 320             <xs:complexType>
2306 321               <xs:attribute name="id" type="xs:IDREF" use="required"/>
2307 322             </xs:complexType>
2308 323           </xs:element>
2309 324           <xs:element name="TargetElementReference">
2310 325             <xs:complexType>
2311 326               <xs:attribute name="id" type="xs:QName" use="required"/>
2312 327             </xs:complexType>
2313 328           </xs:element>
2314 329         </xs:choice>
2315 330       <xs:element name="PropertyDefaults" minOccurs="0">
2316 331         <xs:complexType>
2317 332           <xs:sequence>
2318 333             <xs:any namespace="##other" processContents="lax"/>
2319 334           </xs:sequence>
2320 335         </xs:complexType>
2321 336       </xs:element>
2322 337     <xs:element name="PropertyConstraints" minOccurs="0">
2323 338       <xs:complexType>
2324 339         <xs:sequence>
2325 340           <xs:element name="PropertyConstraint"
2326 341             type="tPropertyConstraint" maxOccurs="unbounded"/>
2327 342         </xs:sequence>

```

```

2328 343     </xs:complexType>
2329 344 </xs:element>
2330 345 <xs:element name="RelationshipConstraints" minOccurs="0">
2331 346   <xs:complexType>
2332 347     <xs:sequence>
2333 348       <xs:element name="RelationshipConstraint"
2334 349         minOccurs="unbounded">
2335 350         <xs:complexType>
2336 351           <xs:sequence>
2337 352             <xs:any namespace="##other" processContents="lax"
2338 353               minOccurs="0"/>
2339 354           </xs:sequence>
2340 355           <xs:attribute name="constraintType" type="xs:anyURI"
2341 356             use="required"/>
2342 357         </xs:complexType>
2343 358       </xs:element>
2344 359     </xs:sequence>
2345 360   </xs:complexType>
2346 361 </xs:element>
2347 362 </xs:sequence>
2348 363 <xs:attribute name="id" type="xs:ID" use="required"/>
2349 364 <xs:attribute name="name" type="xs:string" use="optional"/>
2350 365 <xs:attribute name="relationshipType" type="xs:QName"
2351 366   use="required"/>
2352 367 </xs:extension>
2353 368 </xs:complexContent>
2354 369 </xs:complexType>
2355 370
2356 371 <xs:element name="NodeTypes" type="tNodeTypes"/>
2357 372 <xs:complexType name="tNodeTypes">
2358 373   <xs:sequence>
2359 374     <xs:element name="NodeType" type="tNodeType" minOccurs="unbounded"/>
2360 375   </xs:sequence>
2361 376   <xs:attribute name="targetNamespace" type="xs:anyURI"/>
2362 377 </xs:complexType>
2363 378
2364 379 <xs:element name="NodeType" type="tNodeType"/>
2365 380 <xs:complexType name="tNodeType">
2366 381   <xs:complexContent>
2367 382     <xs:extension base="tExtensibleElements">
2368 383       <xs:sequence>
2369 384         <xs:element name="NodeTypeProperties" minOccurs="0">
2370 385           <xs:complexType>
2371 386             <xs:attribute name="element" type="xs:QName"/>
2372 387             <xs:attribute name="type" type="xs:QName"/>
2373 388           </xs:complexType>
2374 389         </xs:element>
2375 390         <xs:element name="DerivedFrom" minOccurs="0">
2376 391           <xs:complexType>
2377 392             <xs:attribute name="nodeTypeRef" type="xs:QName"
2378 393               use="required"/>
2379 394           </xs:complexType>
2380 395         </xs:element>
2381 396         <xs:element name="InstanceStates"
2382 397           type="tTopologyElementInstanceStates" minOccurs="0"/>
2383 398         <xs:element name="Interfaces" minOccurs="0">
2384 399           <xs:complexType>
2385 400             <xs:sequence>

```

```

2386 401         <xs:element name="Interface" type="tInterface"
2387 402             maxOccurs="unbounded"/>
2388 403     </xs:sequence>
2389 404 </xs:complexType>
2390 405 </xs:element>
2391 406 <xs:element name="Policies" minOccurs="0">
2392 407     <xs:complexType>
2393 408         <xs:sequence>
2394 409             <xs:element name="Policy" type="tPolicy"
2395 410                 maxOccurs="unbounded"/>
2396 411         </xs:sequence>
2397 412     </xs:complexType>
2398 413 </xs:element>
2399 414 <xs:element name="DeploymentArtifacts" minOccurs="0">
2400 415     <xs:complexType>
2401 416         <xs:sequence>
2402 417             <xs:element name="DeploymentArtifact"
2403 418                 type="tDeploymentArtifact" maxOccurs="unbounded"/>
2404 419         </xs:sequence>
2405 420     </xs:complexType>
2406 421 </xs:element>
2407 422 </xs:sequence>
2408 423 <xs:attribute name="id" type="xs:ID" use="required"/>
2409 424 <xs:attribute name="name" type="xs:string" use="optional"/>
2410 425 <xs:attribute name="targetNamespace" type="xs:anyURI"/>
2411 426 <xs:attribute name="abstract" type="tBoolean" default="no"/>
2412 427 <xs:attribute name="final" type="tBoolean" default="no"/>
2413 428 </xs:extension>
2414 429 </xs:complexContent>
2415 430 </xs:complexType>
2416 431
2417 432 <xs:element name="Plans" type="tPlans"/>
2418 433 <xs:complexType name="tPlans">
2419 434     <xs:sequence>
2420 435         <xs:element name="Plan" type="tPlan" maxOccurs="unbounded"/>
2421 436     </xs:sequence>
2422 437     <xs:attribute name="targetNamespace" type="xs:anyURI"/>
2423 438 </xs:complexType>
2424 439
2425 440 <xs:element name="Plan" type="tPlan"/>
2426 441 <xs:complexType name="tPlan">
2427 442     <xs:complexContent>
2428 443         <xs:extension base="tExtensibleElements">
2429 444             <xs:sequence>
2430 445                 <xs:element name="Precondition" type="tCondition" minOccurs="0"/>
2431 446             <xs:choice>
2432 447                 <xs:element name="PlanModel">
2433 448                     <xs:complexType>
2434 449                         <xs:sequence>
2435 450                             <xs:any namespace="##other" processContents="lax"/>
2436 451                         </xs:sequence>
2437 452                     </xs:complexType>
2438 453                 </xs:element>
2439 454                 <xs:element name="PlanModelReference">
2440 455                     <xs:complexType>
2441 456                         <xs:attribute name="reference" type="xs:anyURI"
2442 457                             use="required"/>
2443 458                     </xs:complexType>

```



```

2444 459     </xs:element>
2445 460     </xs:choice>
2446 461     </xs:sequence>
2447 462     <xs:attribute name="id" type="xs:ID" use="required"/>
2448 463     <xs:attribute name="name" type="xs:string" use="optional"/>
2449 464     <xs:attribute name="planType" type="xs:anyURI" use="required"/>
2450 465     <xs:attribute name="languageUsed" type="xs:anyURI" use="required"/>
2451 466     </xs:extension>
2452 467     </xs:complexContent>
2453 468 </xs:complexType>
2454 469
2455 470 <xs:complexType name="tPolicy">
2456 471   <xs:complexContent>
2457 472     <xs:extension base="tExtensibleElements">
2458 473       <xs:attribute name="name" type="xs:string" use="required"/>
2459 474       <xs:attribute name="type" type="xs:anyURI" use="required"/>
2460 475     </xs:extension>
2461 476   </xs:complexContent>
2462 477 </xs:complexType>
2463 478
2464 479 <xs:complexType name="tEnvironmentConstraint">
2465 480   <xs:sequence>
2466 481     <xs:any namespace="##other" processContents="lax"/>
2467 482   </xs:sequence>
2468 483   <xs:attribute name="constraintType" type="xs:anyURI" use="required"/>
2469 484 </xs:complexType>
2470 485
2471 486 <xs:complexType name="tExtensions">
2472 487   <xs:complexContent>
2473 488     <xs:extension base="tExtensibleElements">
2474 489       <xs:sequence>
2475 490         <xs:element name="Extension" type="tExtension"
2476 491           maxOccurs="unbounded"/>
2477 492       </xs:sequence>
2478 493     </xs:extension>
2479 494   </xs:complexContent>
2480 495 </xs:complexType>
2481 496
2482 497 <xs:complexType name="tExtension">
2483 498   <xs:complexContent>
2484 499     <xs:extension base="tExtensibleElements">
2485 500       <xs:attribute name="namespace" type="xs:anyURI" use="required"/>
2486 501       <xs:attribute name="mustUnderstand" type="tBoolean" use="optional"
2487 502         default="yes"/>
2488 503     </xs:extension>
2489 504   </xs:complexContent>
2490 505 </xs:complexType>
2491 506
2492 507 <xs:complexType name="tParameter">
2493 508   <xs:attribute name="name" type="xs:string" use="required"/>
2494 509   <xs:attribute name="type" type="xs:string" use="required"/>
2495 510   <xs:attribute name="required" type="tBoolean" use="optional"
2496 511     default="yes"/>
2497 512 </xs:complexType>
2498 513
2499 514 <xs:complexType name="tInterface">
2500 515   <xs:sequence>
2501 516     <xs:element name="Operation" type="tOperation" minOccurs="0"

```

```

2502 517     maxOccurs="unbounded"/>
2503 518     <xs:element name="ImplementationArtifacts" minOccurs="0">
2504 519         <xs:complexType>
2505 520             <xs:sequence>
2506 521                 <xs:element name="ImplementationArtifact"
2507 522                     type="tImplementationArtifact" maxOccurs="unbounded"/>
2508 523             </xs:sequence>
2509 524         </xs:complexType>
2510 525     </xs:element>
2511 526 </xs:sequence>
2512 527     <xs:attribute name="name" type="xs:anyURI" use="required"/>
2513 528 </xs:complexType>
2514 529
2515 530 <xs:complexType name="tWSDL">
2516 531     <xs:attribute name="portType" type="xs:QName" use="required"/>
2517 532     <xs:attribute name="operation" type="xs:NCName" use="required"/>
2518 533 </xs:complexType>
2519 534
2520 535 <xs:complexType name="tOperation">
2521 536     <xs:complexContent>
2522 537         <xs:extension base="tExtensibleElements">
2523 538             <xs:choice>
2524 539                 <xs:element name="WSDL" type="tWSDL"/>
2525 540                 <xs:element name="REST" type="tREST"/>
2526 541                 <xs:element name="ScriptOperation" type="tScriptOperation"/>
2527 542             </xs:choice>
2528 543             <xs:attribute name="name" type="xs:NCName" use="required"/>
2529 544         </xs:extension>
2530 545     </xs:complexContent>
2531 546 </xs:complexType>
2532 547
2533 548 <xs:complexType name="tREST">
2534 549     <xs:sequence>
2535 550         <xs:element name="Parameters" minOccurs="0">
2536 551             <xs:complexType>
2537 552                 <xs:sequence>
2538 553                     <xs:element name="Parameter" maxOccurs="unbounded">
2539 554                         <xs:complexType>
2540 555                             <xs:attribute name="name" type="xs:string" use="required"/>
2541 556                             <xs:attribute name="required" type="tBoolean" use="optional"
2542 557                                 default="yes"/>
2543 558                         </xs:complexType>
2544 559                     </xs:element>
2545 560                 </xs:sequence>
2546 561             </xs:complexType>
2547 562         </xs:element>
2548 563     <xs:element name="Headers" minOccurs="0">
2549 564         <xs:complexType>
2550 565             <xs:sequence>
2551 566                 <xs:element name="Header" maxOccurs="unbounded">
2552 567                     <xs:complexType>
2553 568                         <xs:attribute name="name" type="xs:string" use="required"/>
2554 569                         <xs:attribute name="required" type="tBoolean" use="optional"
2555 570                             default="yes"/>
2556 571                     </xs:complexType>
2557 572                 </xs:element>
2558 573             </xs:sequence>
2559 574         </xs:complexType>

```

```

2560 575     </xs:element>
2561 576     </xs:sequence>
2562 577     <xs:attribute name="method" default="GET">
2563 578       <xs:simpleType>
2564 579         <xs:restriction base="xs:string">
2565 580           <xs:enumeration value="GET"/>
2566 581           <xs:enumeration value="PUT"/>
2567 582           <xs:enumeration value="POST"/>
2568 583           <xs:enumeration value="DELETE"/>
2569 584         </xs:restriction>
2570 585       </xs:simpleType>
2571 586     </xs:attribute>
2572 587     <xs:attribute name="abs_path" type="xs:anyURI" use="optional"/>
2573 588     <xs:attribute name="absoluteURI" type="xs:anyURI" use="optional"/>
2574 589     <xs:attribute name="requestBody" type="xs:QName" use="optional"/>
2575 590     <xs:attribute name="responseBody" type="xs:QName" use="optional"/>
2576 591   </xs:complexType>
2577 592
2578 593   <xs:complexType name="tScriptOperation">
2579 594     <xs:sequence>
2580 595       <xs:element name="InputParameters" minOccurs="0">
2581 596         <xs:complexType>
2582 597           <xs:sequence>
2583 598             <xs:element name="InputParameter" type="tParameter"
2584 599               maxOccurs="unbounded"/>
2585 600           </xs:sequence>
2586 601         </xs:complexType>
2587 602       </xs:element>
2588 603       <xs:element name="OutputParameters" minOccurs="0">
2589 604         <xs:complexType>
2590 605           <xs:sequence>
2591 606             <xs:element name="OutputParameter" type="tParameter"
2592 607               maxOccurs="unbounded"/>
2593 608           </xs:sequence>
2594 609         </xs:complexType>
2595 610       </xs:element>
2596 611     </xs:sequence>
2597 612   </xs:complexType>
2598 613   <xs:complexType name="tImplementationArtifact">
2599 614     <xs:complexContent>
2600 615       <xs:extension base="tExtensibleElements">
2601 616         <xs:sequence>
2602 617           <xs:element name="RequiredContainerCapabilities" minOccurs="0">
2603 618             <xs:complexType>
2604 619               <xs:sequence>
2605 620                 <xs:element name="RequiredContainerCapability"
2606 621                   maxOccurs="unbounded">
2607 622                   <xs:complexType>
2608 623                     <xs:attribute name="capability" type="xs:anyURI"
2609 624                       use="required"/>
2610 625                   </xs:complexType>
2611 626                 </xs:element>
2612 627               </xs:sequence>
2613 628             </xs:complexType>
2614 629           </xs:element>
2615 630         </xs:sequence>
2616 631       <xs:attribute name="operationName" type="xs:string"
2617 632         use="optional"/>

```

```

2618 633     <xs:attribute name="type" type="xs:anyURI" use="required"/>
2619 634     </xs:extension>
2620 635     </xs:complexContent>
2621 636 </xs:complexType>
2622 637
2623 638 <xs:complexType name="tCondition">
2624 639   <xs:sequence>
2625 640     <xs:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
2626 641   </xs:sequence>
2627 642   <xs:attribute name="expressionLanguage" type="xs:anyURI"
2628 643     use="required"/>
2629 644 </xs:complexType>
2630 645
2631 646 <xs:complexType name="tTopologyElementInstanceStates">
2632 647   <xs:sequence>
2633 648     <xs:element name="InstanceState" maxOccurs="unbounded">
2634 649       <xs:complexType>
2635 650         <xs:attribute name="state" type="xs:anyURI" use="required"/>
2636 651       </xs:complexType>
2637 652     </xs:element>
2638 653   </xs:sequence>
2639 654 </xs:complexType>
2640 655
2641 656 <xs:simpleType name="tBoolean">
2642 657   <xs:restriction base="xs:string">
2643 658     <xs:enumeration value="yes"/>
2644 659     <xs:enumeration value="no"/>
2645 660   </xs:restriction>
2646 661 </xs:simpleType>
2647 662
2648 663 <xs:simpleType name="importedURI">
2649 664   <xs:restriction base="xs:anyURI"/>
2650 665 </xs:simpleType>
2651 666
2652 667 </xs:schema>

```

2653

Appendix E. Sample

2654

This appendix contains the full sample used in this specification.

2655

E.1 Sample Service Topology Definition

2656

```
<ServiceTemplate name="myService"
  targetNamespace="http://www.ibm.com/sample">
  <Types>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
      elementFormDefault="qualified"
      attributeFormDefault="unqualified">
      <xs:element name="ApplicationProperties">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Owner" type="xs:string"/>
            <xs:element name="InstanceName" type="xs:string"/>
            <xs:element name="AccountID" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="AppServerProperties">
        <xs:complexType>
          <xs:sequence>
            <element name="HostName" type="string"/>
            <element name="IPAddress" type="string"/>
            <element name="HeapSize" type="positiveInteger"/>
            <element name="SoapPort" type="positiveInteger"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </Types>

  <TopologyTemplate id="SampleApplication">

    <NodeTemplate id="MyApplication"
      name="My Application"
      nodeType="abc:Application">
      <PropertyDefaults>
        <ApplicationProperties>
          <Owner>Frank</Owner>
          <InstanceName>Thomas' favorite application</InstanceName>
        </ApplicationProperties>
      </PropertyDefaults>
    </NodeTemplate/>

    <NodeTemplate id="MyAppServer"
      name="My Application Server"
      nodeType="abc:ApplicationServer"
      minInstances="0"
      maxInstances="unbounded"/>

    <RelationshipTemplate id="MyDeploymentRelationship"
```

2704

```

2705         relationshipType="deployedOn">
2706     <SourceElement id="MyApplication"/>
2707     <TargetElement id="MyAppServer"/>
2708 </RelationshipTemplate>
2709
2710 </TopologyTemplate>
2711
2712 <NodeTypes>
2713     <NodeType name="Application">
2714         <documentation xml:lang="EN">
2715             A reusable definition of a node type representing an
2716             application that can be deployed on application servers.
2717         </documentation>
2718         <NodeTypeProperties element="ApplicationProperties"/>
2719         <InstanceStates>
2720             <InstanceState state="http://www.example.com/started"/>
2721             <InstanceState state="http://www.example.com/stopped"/>
2722         </InstanceStates>
2723         <Interfaces>
2724             <Interface name="DeploymentInterface">
2725                 <Operation name="DeployApplication">
2726                     <ScriptOperation>
2727                         <InputParameters>
2728                             <InputParamter name="InstanceName"
2729                                 type="string"/>
2730                             <InputParamter name="AppServerHostname"
2731                                 type="string"/>
2732                             <InputParamter name="ContextRoot"
2733                                 type="string"/>
2734                         </InputParameters>
2735                     </ScriptOperation>
2736                 </Operation>
2737                 <ImplementationArtifacts>
2738                     <ImplementationArtifact operationName="DeployApplication"
2739                         type="http://www.example.com/ScriptArtifact/PhythonReference">
2740                         scripts/phython/deployApplication.py
2741                     </ImplementationArtifact>
2742                 </ImplementationArtifacts>
2743             </Interface>
2744         </Interfaces>
2745     </NodeType>
2746     <NodeType name="ApplicationServer">
2747         targetNamespace="http://www.ibm.com/sample">
2748     <NodeTypeProperties element="AppServerProperties"/>
2749     <Interfaces>
2750         <Interface name="MyAppServerInterface">
2751             <Operation name="AcquireNetworkAddress">
2752                 <WSDL portType="my:NetworkPT"
2753                     operation="AcquireNetworkAddress"/>
2754             </Operation>
2755             <Operation name="DeployApplicationServer">
2756                 <WSDL portType="my:AppServerPT"
2757                     operation="DeployApplicationServer"/>
2758             </Operation>
2759             <ImplementationArtifacts>
2760                 <ImplementationArtifact operationName="AcquireNetworkAddress"
2761                     type="http://www.example.com/MyJeeArtifact/EarRef">
2762                     artifacts/jee/MyEAR.ear

```

```

2763         </ImplementationArtifact>
2764         <ImplementationArtifact operationName="DeployApplicationServer"
2765             type="http://www.example.com/MyJeeArtifact/EarRef">
2766             artifacts/jee/AppServerManagement.ear
2767         </ImplementationArtifact>
2768     </ImplementationArtifacts>
2769 </Interface>
2770 </Interfaces>
2771 </NodeType>
2772 </NodeTypes>
2773
2774 <RelationshipTypes>
2775     <documentation xml:lang="EN">
2776         A reusable definition of relation that expresses deployment of
2777         an artifact on a hosting environment.
2778     </documentation>
2779     <RelationshipType name="deployedOn">
2780     </RelationshipType>
2781 </RelationshipTypes>
2782
2783 <Plans>
2784     <Plan id="DeployApplication"
2785         name="Sample Application Build Plan"
2786         planType="http://docs.oasis-
2787             open.org/tosca/ns/2011/12/PlanTypes/BuildPlan"
2788         languageUsed="http://www.omg.org/spec/BPMN/2.0/">
2789
2790         <PreCondition expressionLanguage="www.example.com/text"?
2791             Run only if funding is available
2792         </PreCondition>
2793
2794     <PlanModel>
2795         <process name="DeployNewApplication" id="p1">
2796             <documentation>This process deploys a new instance of the
2797             sample application.
2798             </documentation>
2799
2800             <task id="t1" name="CreateAccount"/>
2801
2802             <task id="t2" name="AcquireNetworkAddresses"
2803                 isSequential="false"
2804                 loopDataInput="t2Input.LoopCounter"/>
2805                 <documentation>Assumption: t2 gets data of type "input"
2806                 as input and this data has a field names "LoopCounter"
2807                 that contains the actual multiplicity of the task.
2808             </documentation>
2809
2810             <task id="t3" name="DeployApplicationServer"
2811                 isSequential="false"
2812                 loopDataInput="t3Input.LoopCounter"/>
2813
2814             <task id="t4" name="DeployApplication"
2815                 isSequential="false"
2816                 loopDataInput="t4Input.LoopCounter"/>
2817
2818             <sequenceFlow id="s1" targetRef="t2" sourceRef="t1"/>
2819             <sequenceFlow id="s2" targetRef="t3" sourceRef="t2"/>
2820             <sequenceFlow id="s3" targetRef="t4" sourceRef="t3"/>

```

```
2821     </process>
2822   </PlanModel>
2823 </Plan>
2824
2825   <Plan id="RemoveApplication"
2826     planType="http://docs.oasis-
2827     open.org/tosca/ns/2011/12/PlanTypes/TerminationPlan"
2828     languageUsed="http://docs.oasis-
2829     open.org/wsbpel/2.0/process/executable">
2830     <PlanModelReference reference="prj:RemoveApp"/>
2831   </Plan>
2832 </Plans>
2833
2834 </ServiceTemplate>
```


Appendix F. Revision History

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

wd-09	2012-07-16	Frank Leyman, Thomas Spatzier, Tobias Kunze	Changes for JIRA Issue TOSCA-17: Specifies the format of a CSAR file; Explained CSAR concept in the corresponding section.
-------	------------	---	---

2836