Topology and Orchestration Specification for Cloud Applications Version 1.0

Committee Specification Draft 04

30 August 2012

Specification URIs

This version:

http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd04/TOSCA-v1.0-csd04.doc (Authoritative)
http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd04/TOSCA-v1.0-csd04.html
http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd04/TOSCA-v1.0-csd04.pdf

Previous 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

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@vnomic.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/csd04/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. 30 August 2012. OASIS Committee Specification Draft 04. http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd04/TOSCA-v1.0-csd04.html.

Notices

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

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

2 Language Design

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

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

2.1 Dependencies on Other Specifications

TOSCA utilizes the following specifications:

- 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, *Key words for use in RFCs to Indicate Requirement Levels*, 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/

[XMLSpec] XML Specification, W3C Recommendation, February 1998, http://www.w3.org/TR/1998/REC-xml-19980210

[XPATH 1.0] XML Path Language (XPath) Version 1.0, W3C Recommendation, November 1999, http://www.w3.org/TR/1999/REC-xpath-19991116

[UNCEFACT XMLNDR] UN/CEFACT XML Naming and Design Rules Technical Specification, Version 3.0, http://www.unece.org/fileadmin/DAM/cefact/xml/UNCEFACT+XML+NDR+V3p0.pdf

2.4 Non-Normative References

2.5 Typographical Conventions

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

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

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

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

2.6 Namespaces

This specification uses a number of namespace prefixes throughout; they are listed in Table 1: Prefixes and namespaces used in this specification

. Note that the choice of any namespace prefix is arbitrary and not semantically significant (see [XML Namespaces]). Furthermore, the namespace http://docs.oasis-open.org/tosca/ns/2011/12 is assumed to be the default namespace, i.e. the corresponding namespace name ste is omitted in this specification to improve readability.

Prefix	Namespace
tosca	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

Table 1: Prefixes and namespaces used in this specification

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

2.7 Language Extensibility

The TOSCA extensibility mechanism allows:

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

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

2.8 ServiceTemplate Document

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

2.8.1 XML Syntax

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

```
<ServiceTemplate id="xs:ID"
                 name="xs:string"?
                 targetNamespace="xs:anyURI"
                 substitutableNodeType="xs:QName"?>

  <Extensions>
    <Extension namespace="xs:anyURI"
               mustUnderstand="yes|no"?/> +
  </Extensions> ?

  <Import namespace="xs:anyURI"?
          location="xs:anyURI"?
          importType="xs:anyURI"/> *

  <Tags>
    <Tag name="xs:string" value="xs:string"/> +
  </Tags> ?
```

```
<BoundaryDefinitions>
  <Properties>
    XML fragment
    <PropertyMappings>
      <PropertyMapping serviceTemplatePropertyRef="xs:string"
                       targetObjectRef="xs:IDREF"
                       targetPropertyRef="xs:IDREF"/> +
      </PropertyMappings/> ?
  </Properties> ?

  <PropertyConstraints>
    <PropertyConstraint property="xs:string"
                        constraintType="xs:anyURI"> +
      constraint ?
    </PropertyConstraint>
  </PropertyConstraints> ?

  <Requirements>
    <Requirement name="xs:string" ref="xs:IDREF"/> +
  </Requirements> ?

  <Capabilities>
    <Capability name="xs:string" ref="xs:IDREF"/> +
  </Capabilities> ?

  <Policies>
    <Policy name="xs:string" type="xs:anyURI">
      policy specific content ?
    </Policy> +
  </Policies> ?

</BoundaryDefinitions> ?

<Types>
  <xs:schema .../> *
</Types> ?

(
 <TopologyTemplate>
   ...
 </TopologyTemplate>
|
 <TopologyTemplateReference reference="xs:QName"/>
)?

<ArtifactTemplates> ... </ArtifactTemplates> ?

<NodeTypes> ... </NodeTypes> ?

<RequirementTypes> ... </RequirementTypes> ?

<CapabilityTypes> ... </CapabilityTypes> ?

<RelationshipTypes> ... </RelationshipTypes> ?

<ArtifactTypes> ... </ArtifactTypes> ?
```

```
  <Plans> ... </Plans> ?

</ServiceTemplate>
```

2.8.2 Properties

The `ServiceTemplate` element has the following properties:

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

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

- `name`: This OPTIONAL attribute specifies the name of the Service Template.

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

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

- `substitutableNodeType`: This OPTIONAL attribute specifies the Node Type this Service Template can substitute, meaning that if another Service Template contains a Node Template of the specified Node Type, this Node Template can be substituted by an instance of this Service Template that then provides the functionality of the substituted node. See section 3.5 for more details.

- `Extensions`: This element specifies namespaces of TOSCA extension attributes and extension elements. The element is OPTIONAL.
  If present, the `Extensions` element MUST include at least one `Extension` element. The `Extension` element is used to specify a namespace of TOSCA extension attributes and extension elements, and indicates whether they are mandatory or optional.
  The attribute `mustUnderstand` is used to specify whether the extension must be understood by a compliant implementation. If the `mustUnderstand` attribute has value “yes” (which is the default value for this attribute) the extension is mandatory. Otherwise, the extension is optional. If a TOSCA implementation does not support one or more of the extensions with `mustUnderstand`="yes", then the Service Template MUST be rejected. Optional extensions MAY be ignored. It is not necessary to declare optional extensions.
  The same extension URI MAY be declared multiple times in the `Extensions` element. If an extension URI is identified as mandatory in one `Extension` element and optional in another, then the mandatory semantics have precedence and MUST be enforced. The extension declarations in an `Extensions` element MUST be treated as an unordered set.

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

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

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

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

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

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

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

- `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by the author to describe the Service Template. Each tag is defined separate, nested `Tag` element. The `Tag` element has the following properties:
  - `name`: This attribute specifies the name of the tag.
  - `value`: This attribute specifies the value of the tag.

    Note: The name/value pairs defined in tags have no normative interpretation.
- `BoundaryDefinitions`: This OPTIONAL element specifies the properties the Service Template exposes beyond its boundaries, i.e. properties that can be observed from outside the Service Template. The `BoundaryDefinitions` element has the following properties.
  - `Properties`: This OPTIONAL element specifies global properties of the Service Template in the form of an XML fragment contained in the body of the `Properties` element. Those properties MAY be mapped to properties of components within the Service Template to make them visible to the outside.
    The `Properties` element has the following properties:
    - `PropertyMappings`: This OPTIONAL element specifies mappings of one or more of the Service Template's properties to properties of components within the Service Template (e.g. Node Templates, Relationship Templates, etc.). Each property mapping is defined by a separated, nested `PropertyMapping` element. The `PropertyMapping` element has the following properties:

- `serviceTemplatePropertyRef`:This attribute identifies a property of the Service Template by means of an XPath expression to be evaluated on the XML fragment defining the Service Template's properties.
- `targetObjectRef`: This attribute specifies the object that provides the property to which the respective Service Template property is mapped. The referenced target object MUST be one of Node Template, Requirement of a Node Template, Capability of a Node Template, or Relationship Template.
- `targetObjectPropertyRef`: This attribute identifies a property of the target object by means of an XPath expression to be evaluated on the XML fragment defining the target object's properties.

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

  Note: If a Service Template property is mapped to a property of a component within the Service Template, reading the Service Template property corresponds to reading the mapped property, and writing the Service Template property corresponds to writing the mapped property.

- `PropertyConstraints`: This OPTIONAL element specifies constraints on one or more of the Service Template's properties. Each constraint is specified by means of a separate nested `PropertyConstraint` element.
  The `PropertyConstraint` element has the following properties:
  - `property`: This attribute identifies a property by means of an XPath expression to be evaluated on the XML fragment defining the Service Template's properties.

    Note: If the property affected by the property constraint is mapped to a property of a component within the Service Template, the property constraint SHOULD be compatible with any property constraint defined for the mapped property.
  - `constraintType`: This attribute specifies the type of constraint by means of a URI, which defines both the semantic meaning of the constraint as well as the format of the content.
  - The body of the `PropertyConstraint` element provides the actual constraint.
    Note: The body MAY be empty in case the `constraintType` URI already specifies the constraint appropriately. For example, a "read-only" constraint could be expressed solely by the `constraintType` URI.
- `Requirements`: This OPTIONAL element specifies Requirements exposed by the Service Template. Those Requirements correspond to Requirements of Node Templates within the Service Template that are propagated beyond the boundaries of the Service Template. Each Requirement is defined by a separate, nested `Requirement` element.
  The `Requirement` element has the following properties:
  - `name`: This OPTIONAL attribute allows for specifying a name of the Requirement other than that specified by the referenced Requirement of a Node Template.
  - `ref`: This attribute references a `Requirement` element of a Node Template within the Service Template.
- `Capabilities`: This OPTIONAL element specifies Capabilities exposed by the Service Template. Those Capabilities correspond to Capabilities of Node Templates within the Service Template that are propagated beyond the boundaries of the Service

Template. Each Capability is defined by a separate, nested `Capability` element. The `Capability` element has the following properties:

- `name`: This OPTIONAL attribute allows for specifying a name of the Capability other than that specified by the referenced Capability of a Node Template.
- `ref`: This attribute references a `Capability` element of a Node Template within the Service Template.

- `Policies`: This OPTIONAL element specifies global policies of the Service Template related to a particular management aspect. Each policy is defined by a separate, nested `Policy` element.
  The `Policy` element has the following properties:
  - `type`: This attribute specifies the kind of policy (e.g. management practice) supported by the Service Template.
  - `name`: This attribute defines the name of the policy. The name value MUST be unique within the containing `Policies` element.

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

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

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

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

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

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

- `ArtifactTemplates`: This element specifies templates describing artifacts referenced by other parts of the Service Template. For example, the installable artifact for an application server node might be defined as an artifact template.

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

- `RequirementTypes`: This element specifies types of requirements that can be exposed by Node Types of a Service Template.

- `CapabilityTypes`: This element specifies types of capabilities that can be exposed by Node Types of a Service Template.
- `RelationshipTypes`: This element specifies the types of relationships, i.e. the kind of links between Node Templates within a Service Template, and their properties.
- `ArtifactTypes`: This element specifies the types of artifacts used within the Service Template. Artifact types might be, for example, application modules such as .war files or .ear files, operating system packages like RPMs, or virtual machine images like .ova files.
- `Plans`: This element specifies the operational behavior of the service. Each `Plan` contained in the `Plans` element specifies how to create, terminate or manage the service.

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

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

```
01 <documentation source="xs:anyURI"? xml:lang="xs:language"?>
   ...
</documentation>
```

Example of use of a documentation:

```
<ServiceTemplate id="myService" name="My Service" ...>

  <documentation xml:lang="EN">
    This is a simple example of the usage of the documentation
    element as nested under a ServiceTemplate element.
  </documentation>

</ServiceTemplate>
```

3 Core Concepts and Usage Pattern

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

3.1 Core Concepts

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.

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.

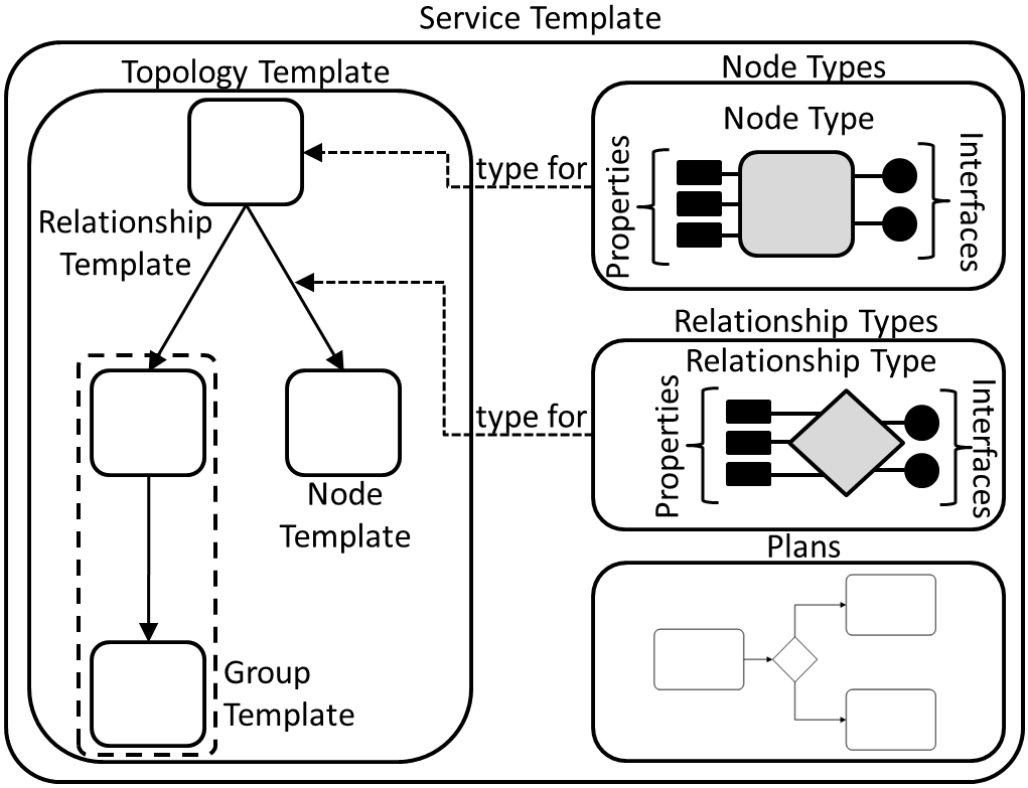

Figure 1: Structural Elements of a Service Template and their Relations

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 address, and an operation for shutting down an instance of this application server. A constraint in the

Node Template can specify a range of IP addresses available when making a concrete application server available.

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

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

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

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

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

3.2 Use Cases

The specification supports at least the following major use cases.

3.2.1 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.

Making a concrete instance of a Topology Template can be done by running a corresponding Plan (so-called instantiating management plan, a.k.a. build plan). This build plan could be provided by the service developer who also creates the Service Template. The build plan can be adapted to the concrete environment of a particular service provider. Other management plans useful in various states of the

whole lifecycle of a service could be specified as part of a Service Template. Similar to build plans such management plans can be adapted to the concrete environment of a particular service provider.

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

3.2.2 Portability of Service Templates

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

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

3.2.3 Service Composition

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

3.2.4 Relation to Virtual Images

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

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

3.3 Service Templates and Artifacts

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

TOSCA distinguishes two kinds of artifacts: *implementation artifacts* and *deployment artifacts*. An implementation artifact represents the executable of an operation of a node type, and a deployment artifact represents the executable for materializing instances of a node. For example, a REST operation

to store an image may have an implementation artifact that is a WAR file. The node type this REST operation is associated with may have the image itself as a deployment artifact.

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

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

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

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

3.4 Requirements and Capabilities

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

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

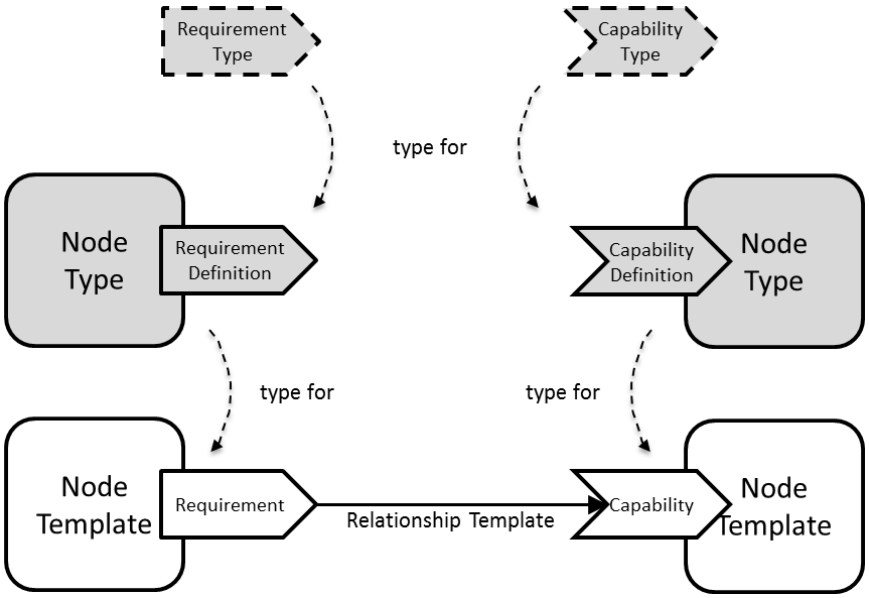

Figure 2: Requirements and Capabilities

Node Templates which have corresponding Node Types with Requirement Definitions or Capability Definitions will include representations of the respective *Requirements* and *Capabilities* with content

specific to the respective Node Template. For example, while Requirement Types just represent Requirement metadata, the Requirement represented in a Node Template can provide concrete values for properties defined in the Requirement Type. In addition, Requirements and Capabilities of Node Templates in a Topology Template can optionally be connected via Relationship Templates to indicate that a specific requirement of one node is fulfilled by a specific capability provided by another node.

Requirements can be matched in two ways as briefly indicated above: (1) requirements of a Node Template can be matched by capabilities of another Node Template in the same Service Template by connecting the respective requirement-capability-pairs via Relationship Templates; (2) requirements of a Node Template can be matched by the general hosting environment (or the TOSCA container), for example by allocating required resources for a Node Template during instantiation.

3.5 Composition of Service Templates

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

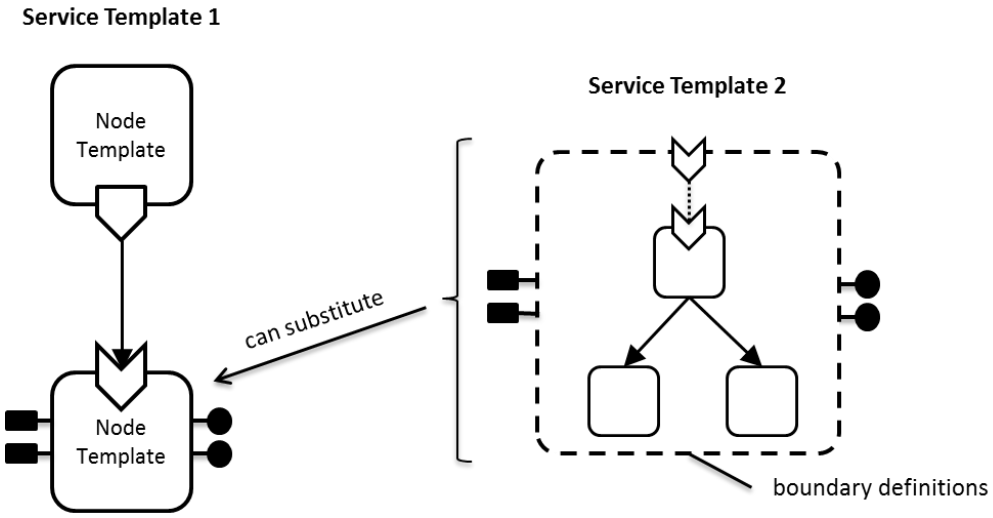

Figure 3: Service Template Composition

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

3.6 Archive Format for Cloud Applications

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

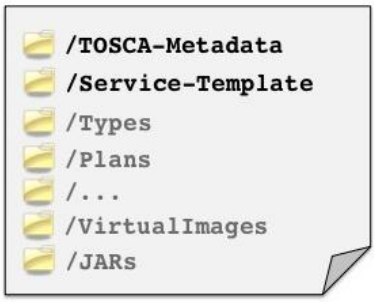

Figure 4: Structure of the CSAR

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

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

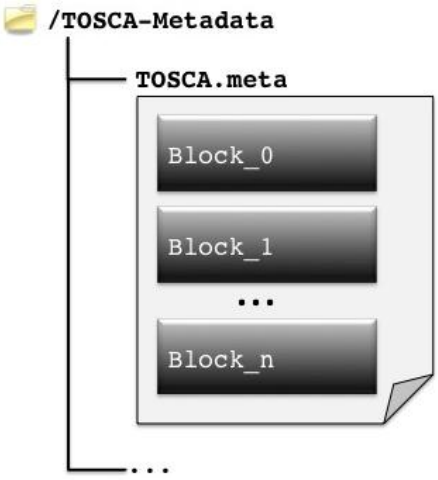

Figure 5: Structure of the TOSCA Meta File

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

627
628 Figure 6: Providing Metadata for Artifacts
629

4 Node Types

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

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

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

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

4.1 XML Syntax

The following pseudo schema defines the XML syntax of NodeTypes:

```
01 <NodeTypes targetNamespace="xs:anyURI"?>
  <NodeType name="xs:NCName" targetNamespace="xs:anyURI"?
            abstract="yes|no"? final="yes|no"?>

    <DerivedFrom typeRef="xs:QName"/> ?

    <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?

    <RequirementDefinitions>
      <RequirementDefinition name="xs:string"
                             requirementType="xs:QName"
                             lowerBound="xs:integer"?
                             upperBound="xs:integer | xs:string"?>
        <Constraints>
          <Constraint constraintType="xs:anyURI">
            constraint type specific content
          </Constraint> +
        </Constraints> ?
      </RequirementDefinition> +
    </RequirementDefinitions> ?

    <CapabilityDefinitions>
      <CapabilityDefinition name="xs:string"
                            capabilityType="xs:QName"
                            lowerBound="xs:integer"?
                            upperBound="xs:integer | xs:string"?>
        <Constraints>
          <Constraint constraintType="xs:anyURI">
            constraint type specific content
          </Constraint> +
        </Constraints> ?
      </CapabilityDefinition> +
    </CapabilityDefinitions>
```

```
    <InstanceStates>
      <InstanceState state="xs:anyURI"> +
    </InstanceStates> ?

    <Interfaces>
      <Interface name="xs:NCName | xs:anyURI">
        <Operation name="xs:NCName">
          (
            <WSDL portType="xs:QName" operation="xs:NCName"/>
           |
            <REST method="GET | PUT | POST | DELETE"
                  abs_path="xs:anyURI"? absoluteURI="xs:anyURI"?
                  requestBody="xs:QName"? responseBody="xs:QName"?>

              <Parameters>
                <Parameter name="xs:string" required="yes|no"/> +
              </Parameters> ?

              <Headers>
                <Header name="xs:string" required="yes|no"/> +
              </Headers> ?

            </REST>
           |
            <ScriptOperation>
              <InputParameters>
                <InputParamter name="xs:string" type="xs:string"
                               required="yes|no"/> +
              </InputParameters> ?
              <OutputParameters>
                <OutputParamter name="xs:string" type="xs:string"
                                required="yes|no"/> +
              </OutputParameters> ?
            </ScriptOperation>
          )
        </Operation> *

        <ImplementationArtifacts>
          <ImplementationArtifact operationName="xs:string"?
                                  artifactType="xs:QName"
                                  artifactRef="xs:QName"?>
            <RequiredContainerFeatures>
              <RequiredContainerFeature feature="xs:anyURI"/> +
            </RequiredContainerFeatures> ?
            artifact specific content
          <ImplementationArtifact> +
        </ImplementationArtifacts> ?
      </Interface> +
    </Interfaces> ?

    <Policies>
      <Policy name="xs:string" type="xs:anyURI">
        policy specific content ?
      </Policy> +
    </Policies> ?

    <DeploymentArtifacts>
      <DeploymentArtifact name="xs:string" artifactType="xs:QName"
```



```
                          artifactRef="xs:QName"?> +
        artifact specific content
      </DeploymentArtifact> +
    </DeploymentArtifacts> ?

  </NodeType> +
</NodeTypes> ?
```

4.2 Properties

The `NodeTypes` element allows for specifying a target namespace to which all contained Node Type definitions will be added by means of its `targetNamespace` attribute. Nested Node Type definitions MAY override this target namespace definition by means of their own `targetNamespace` attributes (see below). If no `targetNamspace` is specified, contained definitions are added to the Service Template document's target namespace.

Each Node Type is defined by a separate, nested `NodeType` element.
The `NodeType` element has the following properties:

- `name`: This attribute specifies the name or identifier of the Node Type, which MUST be unique within the target namespace.
- `targetNamespace`: This OPTIONAL attribute specifies the target namespace to which the definition of the Node Type will be added. If not specified, the Node Type definition will be added to the target namespace defined at the `NodeTypes` element or to the target namespace of the Service Template document containing the Node Type definition.
- `abstract`: This OPTIONAL attribute specifies that no instances can be created from Node Templates that use this Node Type as their type. If a Node Type includes and Requirement Definition or Capability Definition of an abstract Requirement Type or Capability Type, respectively, the Node Type MUST be declared as abstract as well.

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

  Note: an abstract Node Type MUST NOT be declared as final.
- `final`: This OPTIONAL attribute specifies that no other Node Types MUST be derived from the specific Node Type.

  Note: a final Node Type MUST NOT be declared as abstract.
- `DerivedFrom`: This is an OPTIONAL reference to another Node Type from which this Node Type derives. Conflicting definitions are resolved by the rule that local new definitions always override derived definitions. See section 4.3 Derivation Rules for details.
  The `DerivedFrom` element has the following properties:
  - `typeRef`: The QName specifies the Node Type from which this Node Type derives its definitions.
- `PropertiesDefinition`: This element specifies the structure of the observable properties of the Node Type, such as its configuration and state, by means of XML schema.

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

- The element attribute provides the QName of an XML element defining the structure of the Node Type Properties.
- The type attribute provides the QName of an XML (complex) type defining the structure of the Node Type Properties.

- RequirementDefinitions: This OPTIONAL element specifies the requirements that the Node Type exposes (see section 3.4 for details). Each requirement is defined in a nested RequirementDefinition element.
  The RequirementDefinition element has the following properties:
  - name: This attribute specifies the name of the defined requirement and MUST be unique within the RequirementsDefinitions of the current Node Type.

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

- CapabilityDefinitions: This OPTIONAL element specifies the capabilities that the Node Type exposes (see section 3.4 for details). Each capability is defined in a nested CapabilityDefinition element.
  The CapabilityDefinition element has the following properties:
  - name: This attribute specifies the name of the defined capability and MUST be unique within the CapabilityDefinitions of the current Node Type.

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

- `capabilityType`: This attribute identifies by QName the Capability Type of capability that is being defined by the current `CapabilityDefinition`.
- `lowerBound`: This OPTIONAL attribute specifies the lower boundary of requiring nodes that the defined capability can serve. The default value for this attribute is one. A value of zero is invalid, since this would mean that the capability cannot actually satisfy any requiring nodes.
- `upperBound`: This OPTIONAL attribute specifies the upper boundary of client requirements the defined capability can serve. The default value for this attribute is one. A value of "`unbounded`" indicates that there is no upper boundary.
- `Constraints`: This OPTIONAL element contains a list of `Constraint` elements that specify additional constraints on the capability definition.
  The nested `Constraint` element has the following properties:
  - `constraintType`: This attribute specifies the type of constraint. According to this type, the body of the `Constraint` element will contain type specific content.

- `InstanceStates`: This OPTIONAL element lists the set of states an instance of this Node Type can occupy at runtime. Those states are defined in nested `InstanceState` elements.
  The `InstanceState` element has the following nested properties:
  - `state`: This attribute specifies a URI that identifies a potential state.
- `Interfaces`: This element contains the definitions of the operations that can be performed on (instances of) this Node Type. Such operation definitions are given in the form of nested `Interface` elements.
  The `Interface` element has the following properties:
  - `name`: The name of the interface. This name is either a URI or it is an NCName that MUST be unique in the scope of the Node Type being defined.
  - `Operation`: This element defines an operation available to manage particular aspects of the Node Type.

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

    The `Operation` element has the following properties:
    - `name`: This attribute defines the name of the operation and MUST be unique within the containing `Interface` of the Node Type.

- `WSDL`: The operation is implemented by means of Web Service technology. The port type and operation of the Web Service are specified in the corresponding attributes of the `WSDL` element.
  The `WSDL` element has the following properties:
  - `portType`: This is the QName of the port type that contains the definition of the operation defined as part of the interface. Note that the corresponding namespace MUST be imported.
  - `operation`: This attribute specifies the name of an operation of the port type to become part of the interface.
- `REST`: The operation is implemented as a REST API.
  The `REST` element has the following properties:
  - `method`: The HTTP method to be used for building the REST request. If no method is explicitly specified, GET is assumed as default.
  - `abs_path`: The absolute path of the URI that represents the target resource of the request.

    Note, that the proper network location of the URI MUST be set as value of the Host header field of the request when using `abs_path` instead of `absoluteURI`.
  - `absoluteURI`: The absolute URI of the resource.

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

- `ScriptOperation`: The operation is implemented by scripts.
  The `ScriptOperation` element has the following properties:
  - `InputParameters`: This OPTIONAL property contains one or more nested `InputParameter` elements. Each such element specifies three attributes: the `name` of the parameter, its `type`, and whether it must be available as input (`required` attribute with a value of "yes", which is the default) or not (value "no"). Note that the types of the parameters specified for an operation MUST comply with the type systems of the languages of implementations.
  - `OutputParameters`: This OPTIONAL property contains one or more nested `OutputParameter` elements. Each such element specifies three attributes: the `name` of the parameter, its `type`, and whether it must be available as output (`required` attribute with a value of "yes", which is the default) or not (value "no"). Note that the types of the parameters specified for an operation MUST comply with the type systems of the languages of implementations.

- `ImplementationArtifacts`: This element specifies a set of implementation artifacts for operations in an interface.
  The `ImplementationArtifacts` element has the following properties:

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

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

  The `ImplementationArtifact` element has the following properties:
  - `operationName`: This OPTIONAL attribute specifies the name of the operation that is implemented by the actual implementation artifact. If not specified, the implementation artifact is assumed to provide the implementation for all operations defined within the containing interface. For example, a .WAR file could provide the implementation for all REST operations of an interface.
  - `artifactType`: This attribute specifies the type of this artifact. The QName value of this attribute SHOULD correspond to the QName of an `ArtifactType` defined in the same Service Template or in an imported Service Template document.

    The `artifactType` attribute gives an indication on the artifact type specific content of the `ImplementationArtifact` element body. The attribute further indicates the type of Artifact Template referenced by the Implementation Artifact via the `artifactRef` attribute.
  - `artifactRef`: This attribute contains a QName that identifies an Artifact Template to be used as implementation artifact. This

Artifact Template could be defined in the same Service Template document or in a separate, imported document.
The type of Artifact Template referenced by the `artifactRef` attribute MUST be the same type or a sub-type of the type specified in the `artifactType` attribute.

- `RequiredContainerFeatures`: An implementation of an operation might depend on certain features of the environment it is executed in, such as specific (potentially proprietary) APIs of the TOSCA container. For example, an implementation to deploy a virtual machine based on an image could require access to some API provided by a public cloud, while another implementation could require an API of a vendor-specific virtual image library.

  Thus, the contents of the `RequiredContainerFeatures` element provide "hints" to the TOSCA container about which implementation artifact to select for an operation in a case where multiple alternatives are provided.

  Each such dependency is explicitly declared by a separate `RequiredContainerFeature` element. The `feature` attribute of this element is a URI that denotes the corresponding feature of the environment.

- `Policies`: The nested list of elements provides information related to a particular management aspect like billing or monitornig. Those management aspects are defined in the form of `Policy` elements.
  The `Policy` element has the following properties:

- 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.

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

- `DeploymentArtifacts`: This element specifies deployment artifacts relevant for the Node Type. A deployment artifact is an entity that – if specified – is needed for creating an instance of the corresponding Node Type. For example, a virtual image could be a deployment artifact of a JEE server. Each deployment artifact is defined by a nested `DeploymentArtifact` element.
  The `DeploymentArtifact` element has the following properties:

- `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.
- `artifactType`: This attribute specifies the type of this artifact. The QName value of this attribute SHOULD correspond to the QName of an `ArtifactType` defined in the same Service Template or in an imported Service Template document.

  The `artifactType` attribute gives an indication on the artifact type specific content of the `DeploymentArtifact` element body. The attribute further indicates the type of Artifact Template referenced by the Deployment Artifact via the `artifactRef` attribute.
- `artifactRef`: This attribute contains a QName that identifies an Artifact Template to be used as deployment artifact. This Artifact Template could be defined in the same Service Template document or in a separate, imported document.
  The type of Artifact Template referenced by the `artifactRef` attribute MUST be the same type or a sub-type of the type specified in the `artifactType` attribute.
- The body of this element contains the Artifact Type specific content that specifies context information for the use of the artifact.

  For example, if OVF package is referenced as deployment artifact, the body might contain an XML fragment that defines a mapping between service template data and elements of the respective OVF envelope.

4.3 Derivation Rules

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

- 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.
- Requirements and capabilities: The set of requirements or capabilities of the Node Type under definition consists of the set union of requirements or capabilities defined by the Node Type derived from and the requirements or capabilities defined by the Node Type under definition.

  In cases where the Node Type under definition defines a requirement or capability with a certain name where the Node Type derived from already contains a respective definition with the same name, the definition in the Node Type under definition overrides the definition of the Node Type derived from. In such a case, the requirement definition or capability definition, respectively, MUST reference a Requirement Type or Capability Type that is derived from the one in the corresponding requirement definition or capability definition of the Node Type derived from.
- 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.
- 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.
  Two interfaces of the same name will be combined into a single, derived interface with the same name. The set of operations of the derived interface consists of the set union of operations defined by both interfaces. If an operation defined by the Node Type under definition has the

same name as an operation of the Node Type derived from, the former operation substitutes the latter one.

- 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.
  If an implementation artifact defined by the Node Type under definition has the same operation name and type as an implementation artifact of the Node Type derived from, the former implementation artifact substitutes the latter one.
- Deployment Artifacts: The set of deployment artifacts of the Node Type under definition consists of the set union of the deployment artifacts defined by the Nodes Type derived from and the deployment artifacts defined by the Node Type under definition. A deployment artifact defined by the Node Type under definition substitutes a deployment artifact with the same name and type of the Node Type derived from.
- Policies: The set of policies of the Node Type under definition consists of the set union of the policies defined by the Nodes Type derived from and the policies defined by the Node Type under definition. A policy defined by the Node Type under definition substitutes a policy with the same name and type of the Node Type derived from.

4.4 Example

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

```
01 <ServiceTemplate id="myService" name="My Service"
                 targetNamespace="http://www.example.com/sample">
  <NodeTypes>
    <NodeType name="Project">

      <documentation xml:lang="EN">
        A reusable definition of a node type supporting
        the creation of new projects.
      </documentation>

      <PropertiesDefinition element="ProjectProperties"/>

      <InstanceStates>
        <InstanceState state="www.example.com/active"/>
        <InstanceState state="www.example.com/onHalt"/>
      </InstanceStates>

      <Interfaces>
        <Interface name="ProjectInterface">
          <Operation name="CreateProject">
            <ScriptOperation>
              <InputParameters>
                <InputParamter name="ProjectName"
                               type="string"/>
                <InputParamter name="Owner"
                               type="string"/>
                <InputParamter name="AccountID"
                               type="string"/>
              </InputParameters>
```

```
            </ScriptOperation>
          </Operation>
          <ImplementationArtifacts>
            <ImplementationArtifact operationName="CreateProject"
              type="http://www.example.com/ScriptArtifact/
              PhythonReference">
              scripts/phython/createProject.py
            </ImplementationArtifact>
          </ImplementationArtifacts>
        </Interface>
      </Interfaces>
    </NodeType>
  </NodeTypes>
</ServiceTemplate>
```

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

5 Requirement Types

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

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

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

5.1 XML Syntax

The following pseudo schema defines the XML syntax of RequirementTypes:

```
<RequirementTypes targetNamespace="xs:anyURI"?>

  <RequirementType name="xs:NCName"
                   targetNamespace="xs:anyURI"?
                   abstract="yes|no"?
                   final="yes|no"?
                   requiredCapabilityType="xs:QName"?>

    <DerivedFrom typeRef="xs:QName"/> ?

    <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?

  </RequirementType> +

</RequirementTypes> ?
```

5.2 Properties

The `RequirementTypes` element allows for specifying a target namespace to which all contained Requirement Type definitions will be added by means of its `targetNamespace` attribute. Nested Requirement Type definitions MAY override this target namespace definition by means of their own `targetNamespace` attributes (see below). If no `targetNamspace` is specified, contained definitions are added to the Service Template document's target namespace.

Each Requirement Type is defined by a separate, nested `RequirementType` element.
The `RequirementType` element has the following properties:

- `name`: This attribute specifies the name or identifier of the Requirement Type, which MUST be unique within the target namespace.
- `targetNamespace`: This OPTIONAL attribute specifies the target namespace to which the definition of the Requirement Type will be added. If not specified, the Requirement Type definition will be added to the target namespace defined at the `RequirementTypes` element or to the target namespace of the Service Template document containing the Requirement Type definition.

- `abstract`: This OPTIONAL attribute specifies that no instances can be created from Node Templates of a Node Type that defines a requirement of that abstract Requirement Type.

  As a consequence, a Node Type with a Requirement Definition of an abstract Requirement Type MUST be declared as abstract as well and a derived Node Type that defines a requirement of a type derived from the abstract Requirement Type has to be defined. For example, an abstract Node Type “Application” might be defined having a requirement of the abstract type “Container”. A derived Node Type “Web Application” can then be defined with a more concrete requirement of type “Web Application Container” which can then be used for defining Node Templates that can be instantiated during the creation of a service according to a Service Template.

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

- `final`: This OPTIONAL attribute specifies that other Requirement Types MUST NOT be derived from the specific Requirement Type.

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

- `requiredCapabilityType`; This OPTIONAL attribute specifies the type of capability required to match the defined Requirement Type. The QName value of this attribute refers to the QName of a `CapabilityType` element defined in the same Service Template document or in a separate, imported Service Template document.

  Note: The following basic match-making for Requirements and Capabilities MUST be supported by each TOSCA implementation. Each Requirement is defined by a Requirement Definition, which in turn refers to a Requirement Type that specifies the required Capability Type by means of its `requiredCapabilityType` attribute. The value of this attribute is used for basic type-based match-making: a Capability matches a Requirement if the Requirement’s Requirement Type has a `requiredCapabilityType` value that corresponds to the Capability Type of the Capability or one of its super-types.
  Any domain-specific match-making semantics (e.g. based on constraints or properties) has to be defined in the cause of specifying the corresponding Requirement Types and Capability Types.

- `DerivedFrom`: This is an OPTIONAL reference to another Requirement Type from which this Requirement Type derives. See section 5.3 Derivation Rules for details.
  The `DerivedFrom` element has the following properties:

  - `typeRef`: The QName specifies the Requirement Type from which this Requirement Type derives its definitions and semantics.

- `PropertiesDefinition`: This element specifies the structure of the observable properties of the Requirement Type, such as its configuration and state, by means of XML schema.
  The `PropertiesDefinition` element has one but not both of the following properties:

  - The `element` attribute provides the QName of an XML element defining the structure of the Requirement Type Properties.

- The `type` attribute provides the QName of an XML (complex) type defining the structure of the Requirement Type Properties.

5.3 Derivation Rules

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

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

5.4 Example

The following example defines the Requirement Type "DatabaseClientEndpoint" that expresses the requirement of a client for a database connection. It is defined in a Service Template "MyRequirements" within the target namespace "http://www.example.com/SampleRequirements". Thus, by importing the corresponding namespace into another Service Template, the "DatabaseClientEndpoint" Requirement Type is available for use in the other Service Template.

```
01 <ServiceTemplate id="MyRequirements" name="My Requirements"
  targetNamespace="http://www.example.com/SampleRequirements"
  xmlns:br="http://www.example.com/BaseRequirementTypes"
  xmlns:mrp="http://www.example.com/SampleRequirementProperties>

  <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
    namespace="http://www.example.com/BaseRequirementTypes"/>

  <Import importType="http://www.w3.org/2001/XMLSchema"
    namespace="http://www.example.com/SampleRequirementProperties"/>

  <RequirementTypes>

    <RequirementType name="DatabaseClientEndpoint">
      <DerivedFrom typeRef="br:ClientEndpoint"/>
      <PropertiesDefinition
        element="mrp:DatabaseClientEndpointProperties"/>
    </RequirementType>

  </RequirementTypes>

</ServiceTemplate>
```

The Requirement Type "DatabaseClientEndpoint" defined in the example above is derived from another generic "ClientEndpoint" Requirement Type defined in a separate file by means of the `DerivedFrom` element. The definitions in that separate Service Template file are imported by means of the first `Import` element and the namespace of those imported definitions is assigned the prefix "`br`" in the current file.

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

6 Capability Types

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

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

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

6.1 XML Syntax

The following pseudo schema defines the XML syntax of CapabilityTypes:

```
<CapabilityTypes targetNamespace="xs:anyURI"?>
  <CapabilityType name="xs:NCName"
                  targetNamespace="xs:anyURI"?
                  abstract="yes|no"?
                  final="yes|no"?>
    <DerivedFrom typeRef="xs:QName"/> ?
    <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
  </CapabilityType> +
</CapabilityTypes> ?
```

6.2 Properties

The `CapabilityTypes` element allows for specifying a target namespace to which all contained Capability Type definitions will be added by means of its `targetNamespace` attribute. Nested Capability Type definitions MAY override this target namespace definition by means of their own `targetNamespace` attributes (see below). If no `targetNamspace` is specified, contained definitions are added to the Service Template document’s target namespace.

Each Capability Type is defined by a separate, nested `CapabilityType` element.
The `CapabilityType` element has the following properties:

- `name`: This attribute specifies the name or identifier of the Capability Type, which MUST be unique within the target namespace.
- `targetNamespace`: This OPTIONAL attribute specifies the target namespace to which the definition of the Capability Type will be added. If not specified, the Capability Type definition will be added to the target namespace defined at the `CapabilityTypes` element or to the target namespace of the Service Template document containing the Capability Type definition.
- `abstract`: This OPTIONAL attribute specifies that no instances can be created from Node Templates of a Node Type that defines a capability of that abstract Capability Type.

  As a consequence, a Node Type with a Capability Definition of an abstract Capability Type MUST be declared as abstract as well and a derived Node Type that defines a capability of a type derived from the abstract Capability Type has to be defined. For

example, an abstract Node Type “Server” might be defined having a capability of the abstract type “Container”. A derived Node Type “Web Server” can then be defined with a more concrete capability of type “Web Application Container” which can then be used for defining Node Templates that can be instantiated during the creation of a service according to a Service Template.

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

- `final`: This OPTIONAL attribute specifies that other Capability Types MUST NOT be derived from the specific Capability Type.

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

- `DerivedFrom`: This is an OPTIONAL reference to another Capability Type from which this Capability Type derives. See section 6.3 Derivation Rules for details.
  The `DerivedFrom` element has the following properties:
  - `typeRef`: The QName specifies the Capability Type from which this Capability Type derives its definitions and semantics.
- `PropertiesDefinition`: This element specifies the structure of the observable properties of the Capability Type, such as its configuration and state, by means of XML schema.
  The `PropertiesDefinition` element has one but not both of the following properties:
  - The `element` attribute provides the QName of an XML element defining the structure of the Capability Type Properties.
  - The `type` attribute provides the QName of an XML (complex) type defining the structure of the Capability Type Properties.

6.3 Derivation Rules

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

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

6.4 Example

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

```
01 <ServiceTemplate id="MyCapabilities" name="My Capabilities"
  targetNamespace="http://www.example.com/SampleCapabilities"
  xmlns:bc="http://www.example.com/BaseCapabilityTypes"
  xmlns:mcp="http://www.example.com/SampleCapabilityProperties>

  <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
          namespace="http://www.example.com/BaseCapabilityTypes"/>

  <Import importType="http://www.w3.org/2001/XMLSchema"
          namespace="http://www.example.com/SampleCapabilityProperties"/>
```



```
  <CapabilityTypes>

    <CapabilityType name="DatabaseServerEndpoint">
      <DerivedFrom typeRef="bc:ServerEndpoint"/>
      <PropertiesDefinition
        element="mcp:DatabaseServerEndpointProperties"/>
    </RequirementType>

  </RequirementTypes>

</ServiceTemplate>
```

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

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

7 Relationship Types

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

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

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

7.1 XML Syntax

The following pseudo schema defines the XML syntax of RelationshipTypes:

```
<RelationshipTypes targetNamespace="xs:anyURI"?>

  <RelationshipType name="xs:NCName"
                    targetNamespace="xs:anyURI"?
                    abstract="yes|no"?
                    final="yes|no"?> +

    <DerivedFrom typeRef="xs:QName"/> ?

    <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?

    <InstanceStates>
      <InstanceState state="xs:anyURI"> +
    </InstanceStates> ?

    <SourceInterfaces>
      <Interface name="xs:NCName | xs:anyURI">
        ...
      </Interface> +
    </SourceInterfaces> ?

    <TargetInterfaces>
      <Interface name="xs:NCName | xs:anyURI">
        ...
      </Interface> +
    </TargetInterfaces> ?

    <ValidSource typeRef="xs:QName"/> ?

    <ValidTarget typeRef="xs:QName"/> ?

  </RelationshipType>

</RelationshipTypes>
```

7.2 Properties

The `RelationshipTypes` element allows for specifying a target namespace to which all contained Relationship Type definitions will be added by means of its `targetNamespace` attribute. Nested Relationship Type definitions MAY override this target namespace definition by means of their own `targetNamespace` attributes (see below). If no `targetNamspace` is specified, contained definitions are added to the Service Template document's target namespace.

Each Relationship Type is defined by a separate, nested `RelationshipType` element.
The `RelationshipType` element has the following properties:

- `name`: This attribute specifies the name or identifier of the Relationship Type, which MUST be unique within the target namespace.
- `targetNamespace`: This OPTIONAL attribute specifies the target namespace to which the definition of the Relationship Type will be added. If not specified, the Relationship Type definition will be added to the target namespace defined at the `RelationshipTypes` element or to the target namespace of the Service Template document containing the Relationship Type definition.
- `abstract`: This OPTIONAL attribute specifies that no instances can be created from Relationship Templates that use this Relationship Type as their type.

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

  Note: an abstract Relationship Type MUST NOT be declared as final.
- `final`: This OPTIONAL attribute specifies that other Relationship Types MUST NOT be derived from the specific Relationship Type.

  Note: a final Relationship Type MUST NOT be declared as abstract.
- `DerivedFrom`: This is an OPTIONAL reference to another Relationship Type from which this Relationship Type is derived. Conflicting definitions are resolved by the rule that local new definitions always override derived definitions. See section 7.3 Derivation Rules for details.
  The `DerivedFrom` element has the following properties:
  - `typeRef`: The QName specifies the Relationship Type from which this Relationship Type derives its definitions.
- `PropertiesDefinition`: This element specifies the structure of the observable properties of the Relationship Type, such as its configuration and state, by means of XML schema.
  The `PropertiesDefinition` element has one but not both of the following properties:
  - The `element` attribute provides the QName of an XML element defining the structure of the Relationship Type Properties.
  - The `type` attribute provides the QName of an XML (complex) type defining the structure of the Relationship Type Properties.

- `InstanceStates`: This OPTIONAL element lists the set of states an instance of this Relationship Type can occupy at runtime. Those states are defined in nested `InstanceState` elements.
  The `InstanceState` element has the following nested properties:
  - `state`: This attribute specifies a URI that identifies a potential state.
- `SourceInterfaces`: This OPTIONAL element contains definitions of manageability interfaces that can be performed on the source of a relationship of this Relationship Type to actually establish the relationship between the source and the target in the deployed service.
  Those interface definitions are contained in nested `Interface` elements, the content of which is that described for Node Type interfaces (see section 4.2).
- `TargetInterfaces`: This OPTIONAL element contains definitions of manageability interfaces that can be performed on the target of a relationship of this Relationship Type to actually establish the relationship between the source and the target in the deployed service.
  Those interface definitions are contained in nested `Interface` elements, the content of which is that described for Node Type interfaces (see section 4.2).
- `ValidSource`: This OPTIONAL element specifies the type of object that is allowed as a valid origin for relationships defined using the Relationship Type under definition. If not specified, any Node Type is allowed to be the origin of the relationship.
  The `ValidSource` element has the following properties:
  - `typeRef`: This attribute specifies the QName of a Node Type or Requirement Type that is allowed as a valid source for relationships defined using the Relationship Type under definition. Node Types or Requirements Types derived from the specified Node Type or Requirement Type, respectively, MUST also be accepted as valid relationship source.

    Note: If `ValidSource` specifies a Node Type, the `ValidTarget` element (if present) of the Relationship Type under definition MUST also specify a Node Type.
    If `ValidSource` specifies a Requirement Type, the `ValidTarget` element (if present) of the Relationship Type under definition MUST specify a Capability Type. This Capability Type MUST match the requirement defined in `ValidSource`, i.e. it MUST be of the type (or a sub-type of) the capability specified in the `requiredCapabilityType` attribute of the respective `RequirementType` definition.
- `ValidTarget`: This OPTIONAL element specifies the type of object that is allowed as a valid target for relationships defined using the Relationship Type under definition. If not specified, any Node Type is allowed to be the origin of the relationship.
  The `ValidTarget` element has the following properties:
  - `typeRef`: This attribute specifies the QName of a Node Type or Capability Type that is allowed as a valid target for relationships defined using the Relationship Type under definition. Node Types or Capability Types derived from the specified Node Type or Capability Type, respectively, MUST also be accepted as valid targets of relationships.

    Note: If `ValidTarget` specifies a Node Type, the `ValidSource` element (if

present) of the Relationship Type under definition MUST also specify a Node Type.

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

7.3 Derivation Rules

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

- Relationship Type Properties: It is assumed that the XML element (or type) representing the Relationship Type properties of the Relationship Type under definition extends the XML element (or type) of the Relationship Type properties of the Relationship Type referenced in the `DerivedFrom` element.
- Instance States: The resulting set of instance states of the Relationship Type under definition consists of the set union of the instances states defined by the Relationship Type derived from and the instance states explicitly defined by the Relationship Type under definition. Instance states with the same state attribute will be combined into a single instance state of the same state.
- Valid source and target: An object specified as a valid source or target, respectively, of the Relationship Type under definition MUST be of a subtype defined as valid source or target, respectively, of the Relationship Type derived from.

  If the Relationship Type derived from has no valid source or target defined, the types of object being defined in the `ValidSource` or `ValidTarget` elements of the Relationship Type under definition are not restricted.

  If the Relationship Type under definition has no source or target defined, only the types of objects defined as source or target of the Relationship Type derived from are valid origins or destinations of the Relationship Type under definition.

7.4 Example

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

```
01 <RelationshipTypes>

  <RelationshipType name="processDeployedOn">

    <RelationshipTypeProperties element="ProcessDeployedOnProperties"/>

    <InstanceStates>
      <InstanceState state="www.example.com/successfullyDeployed"/>
      <InstanceState state="www.example.com/failed"/>
    </InstanceStates>
```

```
  </RelationshipType>

</RelationshipTypes>
```

8 Artifact Types

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

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

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

8.1 XML Syntax

The following pseudo schema defines the XML syntax of ArtifactTypes:

```
<ArtifactTypes targetNamespace="xs:anyURI"?>

  <ArtifactType name="xs:NCName"
                targetNamespace="xs:anyURI"?
                abstract="yes|no"?
                final="yes|no"?>

    <DerivedFrom typeRef="xs:QName"/> ?

    <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?

  </ArtifactType> +

</ArtifactTypes> ?
```

8.2 Properties

The `ArtifactTypes` element allows for specifying a target namespace to which all contained Artifact Type definitions will be added by means of its `targetNamespace` attribute. Nested Artifact Type definitions MAY override this target namespace definition by means of their own `targetNamespace` attributes (see below). If no `targetNamspace` is specified, contained definitions are added to the Service Template document's target namespace.

Each Artifact Type is defined by a separate, nested `ArtifactType` element.
The `ArtifactType` element has the following properties:

- `name`: This attribute specifies the name or identifier of the Artifact Type, which MUST be unique within the target namespace.

- `targetNamespace`: This OPTIONAL attribute specifies the target namespace to which the definition of the Artifact Type will be added. If not specified, the Artifact Type definition will be added to the target namespace defined at the `ArtifactTypes` element or to the target namespace of the Service Template document containing the Artifact Type definition.
- `abstract`: This OPTIONAL attribute specifies that no instances can be created from Artifact Templates of that abstract Artifact Type, i.e. the respective artifacts cannot be used directly as deployment or implementation artifact in any context.

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

  Note: an abstract Artifact Type MUST NOT be declared as final.
- `final`: This OPTIONAL attribute specifies that other Artifact Types MUST NOT be derived from the specific Artifact Type.

  Note: a final Artifact Type MUST NOT be declared as abstract.
- `DerivedFrom`: This is an OPTIONAL reference to another Artifact Type from which this Artifact Type derives. See section 8.3 Derivation Rules for details.
  The `DerivedFrom` element has the following properties:
  - `typeRef`: The QName specifies the Artifact Type from which this Artifact Type derives its definitions and semantics.
- `PropertiesDefinition`: This element specifies the structure of the observable properties of the Artifact Type, such as its configuration and state, by means of XML schema.
  The `PropertiesDefinition` element has one but not both of the following properties:
  - The `element` attribute provides the QName of an XML element defining the structure of the Artifact Type Properties.
  - The `type` attribute provides the QName of an XML (complex) type defining the structure of the Artifact Type Properties.

8.3 Derivation Rules

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

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

8.4 Example

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

```
01 <ServiceTemplate id="MyArtifacts" name="My Artifacts"
```



```
  targetNamespace="http://www.example.com/SampleArtifacts"
  xmlns:ba="http://www.example.com/BaseArtifactTypes"
  xmlns:map="http://www.example.com/SampleArtifactProperties>

  <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
          namespace="http://www.example.com/BaseArtifactTypes"/>

  <Import importType="http://www.w3.org/2001/XMLSchema"
          namespace="http://www.example.com/SampleArtifactProperties"/>

  <ArtifactTypes>

    <ArtifactType name="RPMPackage">
      <DerivedFrom typeRef="ba:OSPackage"/>
      <PropertiesDefinition element="map:RPMPackageProperties"/>
    </ArtifactType>

  </ArtifactTypes>

</ServiceTemplate>
```

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

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

9 Artifact Templates

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

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

Furthermore, an Artifact Template typically provides one or more references to the actual artifact itself that can be contained as a file in the CSAR (see section 3.6 and section 12) containing the overall Service Template or that can be available at a remote location such as an FTP server.

9.1 XML Syntax

The following pseudo schema defines the XML syntax of ArtifactTemplates:

```
<ArtifactTemplates>

  <ArtifactTemplate id="xs:ID" name="xs:string"? type="xs:QName">

    <Properties>
      XML fragment
    </Properties> ?

    <PropertyConstraints>
      <PropertyConstraint property="xs:string"
                          constraintType="xs:anyURI"> +
        constraint?
      </PropertyConstraint>
    </PropertyConstraints> ?

    <ArifactReferences>
      <ArtifactReference reference="xs:anyURI">
        (
         <Include pattern="xs:string"/>
        |
         <Exclude pattern="xs:string"/>
        )*
      </ArtifactReference> +
    </ArtifactReferences> ?

  </ArtifactTemplate> +

</ArtifactTemplates> ?
```

9.2 Properties

The `ArtifactTemplates` element contains a list of artifact template definitions, each contained in a separate `ArtifactTemplate` element.

The `ArtifactTemplate` element has the following properties:

- `id`: This attribute specifies the identifier of the Artifact Template. The identifier of the Artifact Template MUST be unique within the target namespace.
- `name`: This OPTIONAL attribute specifies the name of the Artifact Template.
- `type`: The QName value of this attribute refers to the Artifact Type providing the type of the Artifact Template.

  Note: If the Artifact Type referenced by the `type` attribute of an Artifact Template is declared as abstract, no instances of the specific Artifact Template can be created, i.e. the artifact cannot be used directly as deployment or implementation artifact. Instead, a substitution of the Artifact Template with one having a specialized, derived Artifact Type has to be done at the latest during the instantiation time of a Service Template.
- `Properties`: This OPTIONAL element specifies the invariant properties of the Artifact Template, i.e. those properties that will be commonly used across different contexts in which the Artifact Template is used.

  The initial values are specified by providing an instance document of the XML schema of the corresponding Artifact Type Properties. This instance document considers the inheritance structure deduced by the `DerivedFrom` property of the Artifact Type referenced by the type attribute of the Artifact Template.
- `PropertyConstraints`: This OPTIONAL element specifies constraints on the use of one or more of the Artifact Type Properties of the Artifact Type providing the property definitions for the Artifact Template. Each constraint is specified by means of a separate nested `PropertyConstraint` element.
  The `PropertyConstraint` element has the following properties:
  - `property`: The string value of this property is an XPath expression pointing to the property within the Artifact Type Properties document that is constrained within the context of the Artifact Template. More than one constraint MUST NOT be defined for each property.
  - `constraintType`: The constraint type is specified by means of a URI, which defines both the semantic meaning of the constraint as well as the format of the content.

    For example, a constraint type of http://www.example.com/PropertyConstraints/unique could denote that the reference property of the Artifact Template under definition has to be unique within a certain scope. The constraint type specific content of the respective `PropertyConstraint` element could then define the actual scope in which uniqueness has to be ensured in more detail.
- `ArtifactReferences`: This OPTIONAL element contains one or more references to the actual artifact proper, each represented by a separate `ArtifactReference` element.
  The `ArtifactReference` element has the following properties:
  - `reference`: This attribute contains a URI pointing to an actual artifact. If this URI is a relative URI, it is interpreted relative to the root directory of the CSAR containing the Service Template (see also sections 3.6 and 12).
  - `Include`: This OPTIONAL element can be used to define a pattern of files that shall be included in the artifact reference in case the reference points to a complete directory.
    The `Include` element has the following properties:
    - `pattern`: This attribute contains a pattern definition for files that shall be included in the overall artifact reference. For example, a pattern of "`*.py`" would include all python scripts contained in a directory.

- `Exclude`: This OPTIONAL element can be used to define a pattern of files that shall be excluded from the artifact reference in case the reference points to a complete directory. The `Exclude` element has the following properties:
  - `pattern`: This attribute contains a pattern definition for files that shall be excluded in the overall artifact reference. For example, a pattern of "`*.sh`" would exclude all bash scripts contained in a directory.

9.3 Example

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

```
01 <ServiceTemplate id="MyArtifacts" name="My Artifacts"
  targetNamespace="http://www.example.com/SampleArtifacts"
  xmlns:ba="http://www.example.com/BaseArtifactTypes">

  <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
          namespace="http://www.example.com/BaseArtifactTypes"/>

  <ArtifactTemplates>

    <ArtifactTemplate id="MyInstallable"
                      name="My installable"
                      type="ba:ZipFile">
      <ArtifactReferences>
        <ArtifactReference reference="files/MyInstallable.zip"/>
      </ArtifactReferences>
    </ArtifactTemplate>

  </ArtifactTemplates>

</ServiceTemplate>
```

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

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

10 Topology Template

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

10.1 XML Syntax

The following pseudo schema defines the XML syntax of TopologyTemplate:

```
<TopologyTemplate id="xs:ID" name="xs:string"?>
  (
    <NodeTemplate id="xs:ID" name="xs:string"? type="xs:QName"
                  minInstances="xs:integer"?
                  maxInstances="xs:integer | xs:string"?>
      <Properties>
        XML fragment
      </Properties> ?

      <PropertyConstraints>
        <PropertyConstraint property="xs:string"
                            constraintType="xs:anyURI">
           constraint ?
        </PropertyConstraint> +
      </PropertyConstraints> ?

      <Requirements>
        <Requirement id="xs:ID" name="xs:string" type="xs:QName"> +
          <Properties>
            XML fragment
          <Properties> ?
          <PropertyConstraints>
            <PropertyConstraint property="xs:string"
                                constraintType="xs:anyURI"> +
              constraint ?
            </PropertyConstraint>
          </PropertyConstraints> ?
        </Requirement>
      </Requirements> ?

      <Capabilities>
        <Capability id="xs:ID" name="xs:string" type="xs:QName"> +
          <Properties>
            XML fragment
          <Properties> ?
          <PropertyConstraints>
            <PropertyConstraint property="xs:string"
                                constraintType="xs:anyURI">
              constraint ?
            </PropertyConstraint> +
          </PropertyConstraints> ?
        </Capability>
      </Capabilities> ?
```

```
      <Policies>
        <Policy name="xs:string" type="xs:anyURI">
          policy specific content ?
        </Policy> +
      </Policies> ?

      <DeploymentArtifacts>
        <DeploymentArtifact name="xs:string" artifactType="xs:QName"
                            artifactRef="xs:QName"?>
           artifact specific content
        </DeploymentArtifact> +
      </DeploymentArtifacts> ?

      <ImplementationArtifacts>
        <ImplementationArtifact operationName="xs:string"?
                                artifactType="xs:QName"
                                artifactRef="xs:QName"?>
          <RequiredContainerFeatures>
            <RequiredContainerFeature feature="xs:anyURI"/> +
          </RequiredContainerFeatures> ?
            artifact specific content
        <ImplementationArtifact> +
      </ImplementationArtifacts> ?
    </NodeTemplate>
   |
    <RelationshipTemplate id="xs:ID" name="xs:string"? type="xs:QName">
      <Properties>
        XML fragment
      </Properties> ?

      <PropertyConstraints>
        <PropertyConstraint property="xs:string"
                            constraintType="xs:anyURI">
          constraint ?
        </PropertyConstraint> +
      </PropertyConstraints> ?

      <SourceElement ref="xs:IDREF"/>
      <TargetElement ref="xs:IDREF"? externalRef="xs:QName"?/>

      <RelationshipConstraints>
        <RelationshipConstraint constraintType="xs:anyURI">
          constraint ?
        </RelationshipConstraint> +
      </RelationshipConstraints> ?

    </RelationshipTemplate>
   |
    <GroupTemplate id="xs:ID" name="xs:string"? minInstances="xs:integer"?
                   maxInstances="xs:integer | xs:string"?>
      (
        <NodeTemplate ... />
      |
        <RelationshipTemplate ... />
      |
     <GroupTemplate ... />
    )+
```



```
    <Policies>
      <Policy name="xs:string" type="xs:anyURI">
        policy specific content ?
      </Policy> +
    </Policies> ?
  </GroupTemplate>
  ) +
</TopologyTemplate>
```

10.2 Properties

The `TopologyTemplate` element has the following properties:

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

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

The `NodeTemplate` element has the following properties:

- `id`: This attribute specifies the identifier of the Node Template. The identifier of the Node Template MUST be unique within the target namespace.
- `name`: This OPTIONAL attribute specifies the name of the Node Template.
- `type`: The QName value of this attribute refers to the Node Type providing the type of the Node Template.

  Note: If the Node Type referenced by the `type` attribute of a Node Template is declared as abstract, no instances of the specific Node Template can be created. Instead, a substitution of the Node Template with one having a specialized, derived Node Type has to be done at the latest during the instantiation time of the Node Template.
- `minInstances`: This integer attribute specifies the minimun number of instances to be created when instantiating the Node Template. The default value of this attribute is 1..The value of `minInstances` MUST NOT be less than 0.
- `maxInstances`: This attribute specifies the maximum number of instances that can be created when instantiating the Node Template. The default value of this attribute is 1. If the string is set to “`unbounded`”, an unbouded number of instances can be created. The value of `maxInstances` MUST be 1 or greater and MUST NOT be less than the value specified for `minInstances`.

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

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

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

- `PropertyConstraints`: Specifies constraints on the use of one or more of the Node Type Properties of the Node Type providing the property definitions for the Node Template. Each constraint is specified by means of a separate nested `PropertyConstraint` element. The `PropertyConstraint` element has the following properties:

  - `property`: The string value of this property is an XPath expression pointing to the property within the Node Type Properties document that is constrained within the context of the Node Template. More than one constraint MUST NOT be defined for each property.

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

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

- `Requirements`: This element contains a list of requirements for the Node Template, according to the list of requirement definitions of the Node Type specified in the `type` attribute of the Node Template. Each requirement is specified in a separate nested `Requirement` element. The `Requirement` Element has the following properties:

  - `id`: This attribute specifies the identifier of the Requirement. The identifier of the Requirement MUST be unique within the target namespace.

  - `name`: This attribute specifies the name of the Requirement. The `name` and `type` of the Requirement MUST match the `name` and `type` of a Requirement Definition in the Node Type specified in the `type` attribute of the Node Template.

  - `type`: The QName value of this attribute refers to the Requirement Type definition of the Requirement. This Requirement Type denotes the semantics and well as potential properties of the Requirement.

  - `Properties`: This element specifies initial values for one or more of the Requirement Properties according to the Requirement Type providing the property definitions. Properties are provided in the form of an XML fragment. The same rules as outlined for the `Properties` element of the Node Template apply.

  - `PropertyConstraints`: This element specifies constraints on the use of one or more of the Properties of the Requirement Type providing the property definitions for the

Requirement. Each constraint is specified by means of a separate nested `PropertyConstraint` element. The same rules as outlined for the `PropertyConstraints` element of the Node Template apply.

- `Capabilities`: This element contains a list of capabilities for the Node Template, according to the list of capability definitions of the Node Type specified in the `type` attribute of the Node Template. Each capability is specified in a separate nested `Capability` element.
  The `Capability` Element has the following properties:
  - `id`: This attribute specifies the identifier of the Capability. The identifier of the Capability MUST be unique within the target namespace.
  - `name`: This attribute specifies the name of the Capability. The `name` and `type` of the Capability MUST match the `name` and `type` of a Capability Definition in the Node Type specified in the `type` attribute of the Node Template.
  - `type`: The QName value of this attribute refers to the Capability Type definition of the Capability. This Capability Type denotes the semantics and well as potential properties of the Capability.
  - `Properties`: This element specifies initial values for one or more of the Capability Properties according to the Capability Type providing the property definitions. Properties are provided in the form of an XML fragment. The same rules as outlined for the `Properties` element of the Node Template apply.
  - `PropertyConstraints`: This element specifies constraints on the use of one or more of the Properties of the Capability Type providing the property definitions for the Capability. Each constraint is specified by means of a separate nested `PropertyConstraint` element. The same rules as outlined for the `PropertyConstraints` element of the Node Template apply.
- `Policies`: Specifies policies of the Node Template. Each policy is specified by means of a separate nested `Policy` element.

  Note that a policy specified in the Node Template overrides any policy of the same name and type that might be specified with the Node Type of this Node Template. Any policies of the Node Type that are not overridden are combined with the policies of the Node Template.

  The nested `Policy` elements have the following properties:
  - `type`: This attribute specifies the kind of policy (e.g. management practice) supported by an instance of the Node Template containing this element.
  - `name`: This attribute defines the name of the policy. The name MUST be unique within a given Node Template containing the `Policy` element.
- `DeploymentArtifacts`: This element specifies the deployment artifacts relevant for the Node Template under definition. Its nested `DeploymentArtifact` elements specify details about individual deployment artifacts.
  The `DeploymentArtifact` element has the following properties:
  - `name`: This attribute specifies the name of the artifact. Uniqueness of the name within the scope of the encompassing Node Template SHOULD be guaranteed by the definition.
  - `artifactType`: This attribute specifies the type of this artifact. The QName value of this attribute SHOULD correspond to the QName of an `ArtifactType` defined in the same Service Template or in an imported Service Template document.

The `artifactType` attribute gives an indication on the artifact type specific content of the `DeploymentArtifact` element body. The attribute further indicates the type of Artifact Template referenced by the Deployment Artifact via the `artifactRef` attribute.

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

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

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

- `ImplementationArtifacts`: This element contains a list of implementation artifacts that provide the implementation for operations of the Node Template. For example, a servlet might be the implementation artifact for a REST API. Each such implementation artifact is defined by a separate nested Implementation Artifact element.
  The `ImplementationArtifact` element has the following properties:
  - `operationName`: This OPTIONAL attribute specifies the name of the operation that is implemented by the implementation artifact.
  - `artifactType`: This attribute specifies the type of this artifact. The QName value of this attribute SHOULD correspond to the QName of an `ArtifactType` defined in the same Service Template or in an imported Service Template document.

    The `artifactType` attribute gives an indication on the artifact type specific content of the `ImplementationArtifact` element body. The attribute further indicates the type of Artifact Template referenced by the Deployment Artifact via the `artifactRef` attribute.
  - `artifactRef`: This attribute contains a QName that identifies an Artifact Template to be used as implementation artifact. This Artifact Template could be defined in the same Service Template document or in a separate, imported document.
    The type of Artifact Template referenced by the `artifactRef` attribute MUST be the same type or a sub-type of the type specified in the `artifactType` attribute.

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

    Otherwise, the implementation artifacts of the Node Type given as value of the `type` attribute of the Node Template under definition and the implementation artifacts defined with the Node Template are combined.
  - `RequiredContainerFeatures`: An implementation of an operation might depend on certain features of the environment it is executed in, such as specific (potentially proprietary) APIs of the TOSCA container. For example, an implementation to deploy a virtual machine based on an image could require access to some API provided by a public cloud, while another implementation could require an API of a vendor-specific virtual image library.

Thus, the contents of the RequiredContainerFeatures element provide “hints” to the TOSCA container about which implementation artifact to select for an operation in a case where multiple alternatives are provided.

Each such dependency is explicitly declared by a separate RequiredContainerFeature element. The feature attribute of this element is a URI that denotes the corresponding feature of the environment.

The RelationshipTemplate element has the following properties:

- id: This attribute specifies the identifier of the Relationship Template. The identifier of the Relationship Template MUST be unique within the target namespace.
- name: This OPTIONAL attribute specifies the name of the Relationship Template.
- type: The QName value of this property refers to the Relationship Type providing the type of the Relationship Template.

  Note: If the Relationship Type referenced by the type 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 Template with one having a specialized, derived Relationship Type has to be done at the latest during the instantiation time of the Relationship Template.
- Properties: 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.

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

  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 Properties element. Once the defined Relationship Template has been instantiated, any XML representation of the Relationship Type properties MUST validate according to the associated XML schema definition.
- PropertyConstraints: Specifies constraints on the use of one or more of the Relationship Type Properties of the Relationship Type providing the property definitions for the Relationship Template. Each constraint is specified by means of a separate nested PropertyConstraint element.

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

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

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

- `SourceElement`: This element specifies the origin of the relationship represented by the current Relationship Template.
  The `SourceElement` element has the following properties:
  - `ref`: This attribute references by ID a Node Template, Requirement of a Node Template, or a Group Template within the same Service Template document that is the source of the Relationship Template.

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

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

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

- `TargetElement`: This element specifies the target of the relationship represented by the current Relationship Template.
  The TargetElement element has the following properties:
  - `ref`: This attribute references by ID a Node Template, Capability of a Node Template, or a Group Template within the same Service Template document that is the target of the Relationship Template.

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

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

    In case a Capability Type is defined a valid target in the Relationship Type definition, the `ref` attribute MUST reference a Capability of the corresponding Capability Type within a Node Template.
  - `externalRef`: This attribute refers by QName to an imported Node Template, Capability of a Node Template, or Group Template defined in another Service Template document as the target of the Relationship Template.

    The same constraints on the type of references object according to a `ValidTarget` definition on the corresponding Relationship Type as outlined for the `ref` attribute above apply.

Either `ref` or `externalRef` MUST be specified but not both.

- `RelationshipConstraints`: This element specifies a list of constraints on the use of the relationship in separate nested `RelationshipConstraint` elements.
  The `RelationshipConstraint` element has the following properties:
  - `constraintType`: This attribute specifies the type of relationship constraint by means of a URI. Depending on the type, the body of the `RelationshipConstraint` element might contain type specific content that further details the actual constraint.

The `GroupTemplate` element has the following properties:

- `id`: This attribute specifies the identifier of the Group Template. The identifier of the Group Template MUST be unique within the target namespace.
- `name`: This OPTIONAL attribute specifies the name of the Group Template.
- `minInstances`: This integer attribute specifies the minimun number of instances to be created when instantiating the Group Template. The default value of this attribute is 1. The value of `minInstances` MUST NOT be less than 0.
- `maxInstances`: This attribute specifies the maximum number of instances that can be created when instantiating the Group Template. The default value of this attribute is 1. If the string is set to “unbounded”, an unbouded number of instances can be created. The value of `maxInstances` MUST be 1 or greater and MUST NOT be less than the value specified for `minInstances`.
- `NodeTemplate`: This is a node template contained within, or grouped by the Group Template.
- `RelationshipTemplate`: This is a relationship template contained within, or grouped by the Group.
- `GroupTemplate`: This is a Group Template of a nested group contained within, or grouped by the Group Template.
- `Policies`: Specifies policies of the Group Template. Each policy is specified by means of a separate nested `Policy` element. This element contains the actual policy specific content of the policy.

10.3 Example

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

```
01 <ServiceTemplate id="myService"
                    name="My Service"
                    targetNamespace="http://www.example.com/sample"
```



```
                xmlns:abc="http://www.example.com/sample">

  <Import namespace="http://www.example.com/sample"
          importType="http://docs.oasis-open.org/tosca/ns/2011/12"/>

  <TopologyTemplate id="SampleApplication">

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

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

    <RelationshipTemplate id="MyDeploymentRelationship"
                          type="deployedOn">
      <SourceElement ref="MyApplication"/>
      <TargetElement ref="MyAppServer"/>
    </RelationshipTemplate>

  </TopologyTemplate>

</ServiceTemplate>
```

11 Plans

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

11.1 XML Syntax

The following pseudo schema defines the XML syntax of Plans:

```
<Plans>
  <Plan id="xs:ID"
        name="xs:string"?
        planType="xs:anyURI"
        languageUsed="xs:anyURI">

     <PreCondition expressionLanguage="xs:anyURI">
       condition
     </PreCondition> ?

    ( <PlanModel>
        actual plan
      </PlanModel>
     |
      <PlanModelReference reference="xs:anyURI"/>
    )

  </Plan> +
</Plans> ?
```

11.2 Properties

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

- `id`: This attribute specifies the identifier of the Plan. The identifier of the Plan MUST be unique within the target namespace.
- `name`: This OPTIONAL attribute specifies the name of the Plan.
- `planType`: The value of the attribute specifies the type of the plan as an indication on what the effect of executing the plan on a service will have. The plan type is specified by means of a URI, allowing for an extensibility mechanism for authors of service templates to define new plan types over time.
  The following plan types are defined as part of the TOSCA specification.
  - http://docs.oasis-open.org/tosca/ns/2011/12/PlanTypes/BuildPlan - This URI defines the *build plan* plan type for plans used to initially create a new instance of a service from a Service Template.

- http://docs.oasis-open.org/tosca/ns/2011/12/PlanTypes/TerminationPlan - This URI defines the *termination plan* plan type for plans used to terminate the existence of a service instance.

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

- `languageUsed`: This attribute denotes the process modeling language (or metamodel) used to specify the plan. For example, "http://www.omg.org/spec/BPMN/2.0/" would specify that BPMN 2.0 has been used to model the plan.
- `PreCondition`: This OPTIONAL element specifies a condition that needs to be satisfied in order for the plan to be executed. The `expressionLanguage` attribute of this element specifies the expression language the nested condition is provided in.

  Typically, the precondition will be an expression in the instance state attribute of some of the node templates or relationship templates of the topology template. It will be evaluated based on the actual values of the corresponding attributes at the time the plan is requested to be executed. Note, that any other kind of pre-condition is allowed.
- `PlanModel`: This property contains the actual model content.
- `PlanModelReference`: This property points to the model content. Its reference attribute contains a URI of the model of the plan.

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

11.3 Use of Process Modeling Languages

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

11.4 Example

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

```
01 <Plans>

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

     <PreCondition expressionLanguage="www.example.com/text"> ?
         Run only if funding is available
     </PreCondition>

    <PlanModel>
      <process name="DeployNewApplication" id="p1">
```

```
        <documentation>This process deploys a new instance of the
          sample application.
        </documentation>

        <task id="t1" name="CreateAccount"/>

        <task id="t2" name="AcquireNetworkAddresses"
              isSequential="false"
              loopDataInput="t2Input.LoopCounter"/>
          <documentation>Assumption: t2 gets data of type "input"
             as input and this data has a field names "LoopCounter"
             that contains the actual multiplicity of the task.
          </documentation>

        <task id="t3" name="DeployApplicationServer"
              isSequential="false"
              loopDataInput="t3Input.LoopCounter"/>

        <task id="t4" name="DeployApplication"
              isSequential="false"
              loopDataInput="t4Input.LoopCounter"/>

        <sequenceFlow id="s1" targetRef="t2" sourceRef="t1"/>
        <sequenceFlow id="s2" targetRef="t3" sourceRef="t2"/>
        <sequenceFlow id="s3" targetRef="t4" sourceRef="t3"/>
      </process>
    </PlanModel>
  </Plan>

  <Plan id="RemoveApplication"
        planType="http://docs.oasis-
         open.org/tosca/ns/2011/12/PlanTypes/TerminationPlan"
        languageUsed=
         "http://docs.oasis-open.org/wsbpel/2.0/process/executable">
    <PlanModelReference reference="prj:RemoveApp"/>
  </Plan>

</Plans>
```

12 Cloud Service Archive (CSAR)

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

12.1 Overall Structure of a CSAR

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

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

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

12.2 TOSCA Meta File

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

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

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

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

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

```
TOSCA-Meta-File-Version: digit.digit
CSAR-Version: digit.digit
Created-By: string
01 Entry-Service-Template: filename
```

The name/value pairs are as follows:

- `TOSCA-Meta-File-Version`: This is the version number of the TOSCA meta file format. The value must be “1.0” in the current version of the TOSCA specification.
- `CSAR-Version`: This is the version number of the CSAR specification. The value must be “1.0” in the current version of the TOSCA specification.
- `Created-By`: The person or vendor, respectively, who created the CSAR.
- `Entry-Service-Template`: The service template from the Service-Template directory of the CSAR that is the entry point for the overall cloud application.
  Note, that a CSAR may contain multiple Service Template files. One reason for this is

completeness, e.g. a Service Template containing Node Types referred to by the entry Service Template might be included in the Service-Template directory to avoid importing it from external locations.

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

```
Name: <path-name_1>
Content-Type: type_1/subtype_1
<name_11>: <value_11>
<name_12>: <value_12>
...
<name_1n>: <value_1n>
01
02 ...
03
04 Name: <path-name_k>
05 Content-Type: type_k/subtype_k
06 <name_k1>: <value_k1>
07 <name_k2>: <value_k2>
08 ...
09 <name_km>: <value_km>
```

The name/value pairs are as follows:

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

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

12.3 Example

Figure 7 depicts a sample Service Template of an application, named `Payroll.ste`. The application is a payroll application written in Java that must be deployed on a proper application server. The Service Template of the application defines the Node Template `Payroll Application`, the Node Template `Application Server`, as well as the Relationship Template `deployed_on`. The `Payroll Application` is associated with an EAR file (named `Payroll.ear`) which is provided as corresponding Deployment Artifact of the `Payroll Application` Node Template. An Amazon Machine Image (AMI) is the Deployment Artifact of the `Application Server` Node Template; this Deployment Artifact is a reference to the image in the Amazon EC2 environment. The Implementation Artifacts of some operations of the Node Templates are provided too; for example, the `start` operation of the `Payroll Application` is implemented by a Java API supported by the `payrolladm.jar` file, the `installApp` operation of the `Application Server` is realized by the Python script `wsadmin.py`, while the `runInstances` operation is a REST API available at Amazon for running instances of an AMI. Note, that the `runInstances` operation is not related to a particular implementation artifact because it is available as an Amazon Web Service (https://ec2.amazonaws.com/?Action=RunInstances); but the details of this REST API are specified with the operation of the `Application Server` Node Type.

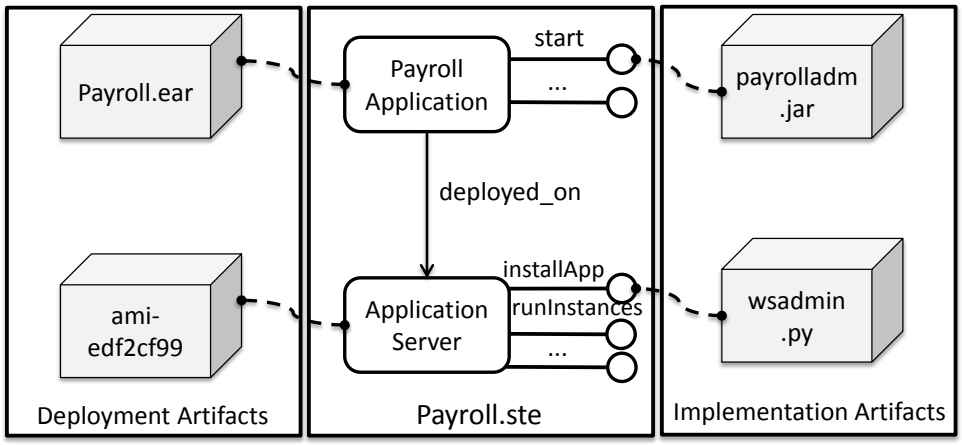

Figure 7: Sample Service Template

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:

```
01 <ServiceTemplate id="Payroll"
                 targetNamespace="http://www.example.com/ste"
                 xmlns:pay="http://www.example.com/ste/Types">

  <Import namespace="http://www.example.com/ste/Types"
          location="http://www.example.com/ste/Types/PayrollTypes.ste"
          importType=" http://docs.oasis-open.org/ns/tosca/2011/12"/>

  <Types>
     ...
  </Types>

  <TopologyTemplate ID="PayrollTemplate">

    <NodeTemplate id="Payroll Application"
                  type="pay:ApplicationNodeType">
       ...

       <DeploymentArtifacts>
         <DeploymentArtifact name="PayrollEAR"
                             type="http://www.example.com/
                                   ns/tosca/2011/12/
                                   DeploymentArtifactTypes/CSARref">
                 EARs/Payroll.ear
         </DeploymentArtifact>
       </DeploymentArtifacts>

      <ImplementationArtifacts>
        <ImplementationArtifact operationName="start"
                                type="http://www.example.com/
                                      ns/tosca/2011/12/
                                     ImplementationArtifactTypes/CSARref">
                JARs/payrolladm.jar
        <ImplementationArtifact>
      </ImplementationArtifacts>

    </NodeTemplate>
```

```
    <NodeTemplate id="Application Server"
                  type="pay:ApplicationServerNodeType">
       ...

       <DeploymentArtifacts>
         <DeploymentArtifact name="ApplicationServerImage"
                             type="http://www.example.com/
                                   ns/tosca/2011/12/
                                   DeploymentArtifactTypes/AMIref">
                  ami-edf2cf99
         </DeploymentArtifact>
       </DeploymentArtifacts>

      <ImplementationArtifacts>
        <ImplementationArtifact operationName="installApp"
                                type="http://www.example.com/
                                      ns/tosca/2011/12/
                                     ImplementationArtifactTypes/CSARref">
                  Python/wsadmin.py
        <ImplementationArtifact>
      </ImplementationArtifacts>

    </NodeTemplate>

    <RelationshipTemplate id="deployed_on"
                          type="pay:deployed_on">
        <SourceElement ref="Payroll Application"/>
        <TargetElement ref="Application Server"/>
    </RelationshipTemplate>

  </TopologyTemplate>

</ServiceTemplate>
```

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

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

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

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

Figure 8: Structure of CSAR Sample

The `TOSCA.meta` file is as follows:

```
TOSCA-Meta-Version: 1.0
CSAR-Version: 1.0
01 Created-By: Frank
02 Entry-Service-Template: /Service-Template/Payroll.ste
03
04 Name: Service-Template/Payroll.ste
05 Content-Type: application/vnd.oasis.service_template
06
07 Name: Service-Template/PayrollTypes.ste
08 Content-Type: application/vnd.oasis.service_template
09
10 Name: Plans/AddUser.bpmn
11 Content-Type: application/vnd.oasis.bpmn
12
13 Name: EARs/Payroll.ear
14 Content-Type: application/vnd.oasis.ear
15
16 Name: JARs/Payrolladm.jar
17 Content-Type: application/vnd.oasis.jar
18
19 Name: Python/wsadmin.py
20 Content-Type: application/vnd.oasis.py
```

13 Security Considerations

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

14 Conformance

This section is to be done.

Appendix A. Portability and Interoperability Considerations

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

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

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

Portability demands support of TOSCA elements.

Appendix B. Acknowledgements

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

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.

2664

2665

Appendix C. Complete TOSCA Grammar

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

```
01 <ServiceTemplate id="xs:ID"
                    name="xs:string"?
                    targetNamespace="xs:anyURI">

  <Extensions>
    <Extension namespace="xs:anyURI"
               mustUnderstand="yes|no"?/> +
  </Extensions> ?

  <Import namespace="xs:anyURI"?
          location="xs:anyURI"?
          importType="xs:anyURI"/> *

  <Tags>
    <Tag name="xs:string" value="xs:string"> +
  </Tags> ?

  <BoundaryDefinitions>
    <Properties>
      XML fragment
      <PropertyMappings>
        <PropertyMapping serviceTemplatePropertyRef="xs:string"
                         targetObjectRef="xs:IDREF"
                         targetPropertyRef="xs:IDREF"/> +
        </PropertyMappings/> ?
    </Properties> ?

    <PropertyConstraints>
      <PropertyConstraint property="xs:string"
                          constraintType="xs:anyURI"> +
        constraint ?
      </PropertyConstraint>
    </PropertyConstraints> ?

    <Requirements>
      <Requirement name="xs:string" ref="xs:IDREF"/> +
    </Requirements> ?

    <Capabilities>
      <Capability name="xs:string" ref="xs:IDREF"/> +
    </Capabilities> ?

    <Policies>
      <Policy name="xs:string" type="xs:anyURI">
        policy specific content ?
      </Policy> +
    </Policies> ?

  </BoundaryDefinitions> ?

  <Types>
```

```
    <xs:schema .../> *
  </Types> ?

 (
  <TopologyTemplateReference reference="xs:QName"/>
  |
  <TopologyTemplate id="xs:ID"
                    name="xs:string"?>

    (
     <NodeTemplate id="xs:ID"
                   name="xs:string"?
                   type="xs:QName"
                   minInstances="xs:integer"?
                   maxInstances="xs:integer | xs:string"?>

       <Properties>
         XML fragment
       </Properties> ?

       <PropertyConstraints>
         <PropertyConstraint property="xs:string"
                             constraintType="xs:anyURI"> +
           constraint ?
         </PropertyConstraint>
       </PropertyConstraints> ?

       <Requirements>
         <Requirement id="xs:ID"
                      name="xs:string"
                      type="xs:QName"> +

           <Properties>
             XML fragment
           <Properties> ?

           <PropertyConstraints>
             <PropertyConstraint property="xs:string"
                                 constraintType="xs:anyURI">
               constraint ?
             </PropertyConstraint> +
           </PropertyConstraints> ?

         </Requirement>
       </Requirements> ?

       <Capabilities>
         <Capability id="xs:ID"
                     name="xs:string"
                     type="xs:QName"> +

           <Properties>
             XML fragment
           <Properties> ?

           <PropertyConstraints>
             <PropertyConstraint property="xs:string"
                                 constraintType="xs:anyURI">
```

```
              constraint ?
            </PropertyConstraint> +
          </PropertyConstraints> ?

        </Capability>
      </Capabilities> ?

      <Policies>
        <Policy name="xs:string" type="xs:anyURI">
          policy specific content ?
        </Policy> +
      </Policies> ?

      <DeploymentArtifacts>
        <DeploymentArtifact name="xs:string"
                            artifactType="xs:QName"
                            artifactRef="xs:QName"?>
           artifact specific content
        </DeploymentArtifact> +
      </DeploymentArtifacts> ?

      <ImplementationArtifacts>
        <ImplementationArtifact operationName="xs:string"
                                artifactType="xs:QName"
                                artifactRef="xs:QName"?>
          <RequiredContainerFeatures>
            <RequiredContainerFeature feature="xs:anyURI"/> +
          </RequiredContainerFeatures> ?
          artifact specific content
        <ImplementationArtifact> +
      </ImplementationArtifacts> ?

    </NodeTemplate>
    |
    <RelationshipTemplate id="xs:ID"
                          name="xs:string"?
                          type="xs:QName">

      <SourceElement ref="xs:IDREF"/>

      <TargetElement ref="xs:IDREF"?
                     externalRef="xs:IDREF"?/>

      <PropertyDefaults>
        XML fragment
      </PropertyDefaults> ?

      <PropertyConstraints>
        <PropertyConstraint property="xs:string"
                            constraintType="xs:anyURI">
          constraint ?
        </PropertyConstraint> +
      </PropertyConstraints> ?

      <RelationshipConstraints>
        <RelationshipConstraint constraintType="xs:anyURI">
           constraint ?
        </RelationshipConstraint> +
```



```
      </RelationshipConstraints> ?

    </RelationshipTemplate> +
   |
    <GroupTemplate id="xs:ID"
                   name="xs:string"?
                   minInstances="xs:integer"?
                   maxInstances="xs:integer | xs:string"?>

      (
       <NodeTemplate ... />
      |
       <RelationshipTemplate ... />
      |
       <GroupTemplate ... />
      )+

      <Policies>
        <Policy name="xs:string" type="xs:anyURI">
           policy specific content ?
        </Policy> +
      </Policies> ?

    </GroupTemplate>
   ) +

 </TopologyTemplate>
) ?

 <ArtifactTemplates>
   <ArtifactTemplate id="xs:ID"
                     name="xs:string"?
                     type="xs:QName"> +

     <Properties>
       XML fragment
     </Properties> ?

     <PropertyConstraints>
       <PropertyConstraint property="xs:string"
                           constraintType="xs:anyURI">
         constraint ?
       </PropertyConstraint> +
     </PropertyConstraints> ?

     <ArifactReferences>
       <ArtifactReference reference="xs:anyURI">
         (
          <Include pattern="xs:string"/>
         |
          <Exclude pattern="xs:string"/>
         )*
       </ArtifactReference> +
     </ArtifactReferences> ?

   </ArtifactTemplate>
 </ArtifactTemplates> ?
```

```
<NodeTypes targetNamespace="xs:anyURI"?>

 <NodeType name="xs:NCName"
           targetNamespace="xs:anyURI"?
           abstract="yes|no"?
           final="yes|no"?>

  <DerivedFrom typeRef="xs:QName"/> ?

  <PropertiesDefinition element="xs:QName"?
                        type="xs:QName"?/> ?

  <RequirementDefinitions>
    <RequirementDefinition name="xs:string"
                           requirementType="xs:QName"
                           lowerBound="xs:integer"?
                           upperBound="xs:integer | xs:string"?>
      <Constraints>
        <Constraint constraintType="xs:anyURI">
          constraint type specific content
        </Constraint> +
      </Constraints> ?
    </RequirementDefinition> +
  </RequirementDefinitions> ?

  <CapabilityDefinitions>
    <CapabilityDefinition name="xs:string"
                          capabilityType="xs:QName"
                          lowerBound="xs:integer"?
                          upperBound="xs:integer | xs:string"?>
      <Constraints>
        <Constraint constraintType="xs:anyURI">
          constraint type specific content
        </Constraint> +
      </Constraints> ?
    </CapabilityDefinition> +
  </CapabilityDefinitions> ?

  <InstanceStates>
    <InstanceState state="xs:anyURI"> +
  </InstanceStates> ?

  <Interfaces>
    <Interface name="xs:NCName | xs:anyURI">

      <Operation name="xs:NCName">

        (
          <WSDL portType="xs:QName"
                operation="xs:NCName"/>
         |
          <REST method="GET | PUT | POST | DELETE"
                abs_path="xs:anyURI"?
                absoluteURI="xs:anyURI"?
                requestBody="xs:QName"?
                responseBody="xs:QName"?>

            <Parameters>
```



```
              <Parameter name="xs:string" required="yes|no"/> +
            </Parameters> ?

            <Headers>
              <Header name="xs:string" required="yes|no"/> +
            </Headers> ?

          </REST>
         |
          <ScriptOperation>
            <InputParameters>
              <InputParamter name="xs:string"
                             type="xs:string"
                             required="yes|no"/> +

            </InputParameters> ?

            <OutputParameters>
              <OutputParamter name="xs:string"
                              type="xs:string"
                              required="yes|no"/> +
            </OutputParameters> ?
          </ScriptOperation>
        )

      </Operation> +

      <ImplementationArtifacts>
        <ImplementationArtifact operationName="xs:string"?
                                artifactType="xs:QName"
                                artifactRef="xs:QName"?> +

          <RequiredContainerFeatures>
            <RequiredContainerFeature feature="xs:anyURI"/> +
          </RequiredContainerFeatures> ?
          artifact specific content
        <ImplementationArtifact>
      </ImplementationArtifacts> ?
    </Interface> +
  </Interfaces> ?

  <DeploymentArtifacts>
    <DeploymentArtifact name="xs:string"
                        artifactType="xs:QName"
                        artifactRef="xs:QName"?> +
      artifact specific content
    </DeploymentArtifact>
  </DeploymentArtifacts> ?

  <Policies>
    <Policy name="xs:string" type="xs:anyURI">
       policy specific content ?
    </Policy> +
  </Policies> ?

 </NodeType> +

</NodeTypes> ?
```

```
<RequirementTypes targetNamespace="xs:anyURI"?>
  <RequirementType name="xs:NCName"
                   targetNamespace="xs:anyURI"?
                   abstract="yes|no"?
                   final="yes|no"?
                   requiredCapabilityType="xs:QName"?> +
    <DerivedFrom typeRef="xs:QName"/> ?
    <PropertiesDefinition element="xs:QName"?
                          type="xs:QName"?/> ?
  </RequirementType>
</RequirementTypes> ?

<CapabilityTypes targetNamespace="xs:anyURI"?>
  <CapabilityType name="xs:NCName"
                  targetNamespace="xs:anyURI"?
                  abstract="yes|no"?
                  final="yes|no"?> +
    <DerivedFrom typeRef="xs:QName"/> ?
    <PropertiesDefinition element="xs:QName"?
                          type="xs:QName"?/> ?
  </CapabilityType>
</CapabilityTypes> ?

<RelationshipTypes targetNamespace="xs:anyURI"?>
  <RelationshipType name="xs:NCName"
                    targetNamespace="xs:anyURI"?
                    abstract="yes|no"?
                    final="yes|no"?> +
    <DerivedFrom typeRef="xs:QName"/> ?
    <PropertiesDefinition element="xs:QName"?
                          type="xs:QName"?/> ?
    <InstanceStates>
      <InstanceState state="xs:anyURI"> +
    </InstanceStates> ?
    <SourceInterfaces>
      <Interface name="xs:NCName | xs:anyURI"> +
        ...
      </Interface>
    </SourceInterfaces> ?
    <TargetInterfaces>
      <Interface name="xs:NCName | xs:anyURI">
        ...
      </Interface> +
    </TargetInterfaces> ?
    <ValidSource typeRef="xs:QName"/> ?
    <ValidTarget typeRef="xs:QName"/> ?
  </RelationshipType> ?
</RelationshipTypes>

<ArtifactTypes targetNamespace="xs:anyURI"?> ?
  <ArtifactType name="xs:NCName"
                targetNamespace="xs:anyURI"?
                abstract="yes|no"?
                final="yes|no"?>
    <DerivedFrom typeRef="xs:QName"/> ?
    <PropertiesDefinition element="xs:QName"?
                          type="xs:QName"?/> ?
```



```
    </ArtifactType> +
  </ArtifactTypes> ?

  <Plans>
    <Plan id="xs:ID"
          name="xs:string"?
          planType="xs:anyURI"
          languageUsed="xs:anyURI"> +

      <PreCondition expressionLanguage="xs:anyURI">
        condition
      </PreCondition> ?

      ( <PlanModel>
          actual plan
        </PlanModel>
      |
        <PlanModelReference reference="xs:anyURI"/>
      )

    </Plan>
  </Plans> ?
</ServiceTemplate>
```

Appendix D. TOSCA Schema

TOSCA-v1.0.xsd:

```
01 <?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://docs.oasis-open.org/tosca/ns/2011/12"
 elementFormDefault="qualified" attributeFormDefault="unqualified"
 xmlns="http://docs.oasis-open.org/tosca/ns/2011/12"
 xmlns:xs="http://www.w3.org/2001/XMLSchema">

 <xs:import namespace="http://www.w3.org/XML/1998/namespace"
  schemaLocation="http://www.w3.org/2001/xml.xsd"/>

 <xs:element name="documentation" type="tDocumentation"/>
 <xs:complexType name="tDocumentation" mixed="true">
  <xs:sequence>
   <xs:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="source" type="xs:anyURI"/>
  <xs:attribute ref="xml:lang"/>
 </xs:complexType>

 <xs:complexType name="tExtensibleElements">
  <xs:sequence>
   <xs:element ref="documentation" minOccurs="0" maxOccurs="unbounded"/>
   <xs:any namespace="##other" processContents="lax" minOccurs="0"
    maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:anyAttribute namespace="##other" processContents="lax"/>
 </xs:complexType>

 <xs:complexType name="tImport">
  <xs:complexContent>
   <xs:extension base="tExtensibleElements">
    <xs:attribute name="namespace" type="xs:anyURI"/>
    <xs:attribute name="location" type="xs:anyURI"/>
    <xs:attribute name="importType" type="importedURI" use="required"/>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>

 <xs:element name="ServiceTemplate">
  <xs:complexType>
   <xs:complexContent>
    <xs:extension base="tServiceTemplate"/>
   </xs:complexContent>
  </xs:complexType>
 </xs:element>

 <xs:complexType name="tServiceTemplate">
  <xs:complexContent>
   <xs:extension base="tExtensibleElements">
    <xs:sequence>
     <xs:element name="Import" type="tImport" minOccurs="0"
      maxOccurs="unbounded"/>
     <xs:element name="Tags" type="tTags" minOccurs="0" maxOccurs="1"/>
```

```
      <xs:element name="BoundaryDefinitions" type="tBoundaryDefinitions"
       minOccurs="0"/>
      <xs:element name="Types" minOccurs="0">
       <xs:complexType>
        <xs:sequence>
         <xs:any namespace="##other" processContents="lax" minOccurs="0"
          maxOccurs="unbounded"/>
        </xs:sequence>
       </xs:complexType>
      </xs:element>
      <xs:element name="Extensions" minOccurs="0">
       <xs:complexType>
        <xs:sequence>
         <xs:element name="Extension" type="tExtension"
          maxOccurs="unbounded"/>
        </xs:sequence>
       </xs:complexType>
      </xs:element>
      <xs:choice minOccurs="0">
       <xs:element name="TopologyTemplateReference">
        <xs:complexType>
         <xs:attribute name="reference" type="xs:QName"/>
        </xs:complexType>
       </xs:element>
       <xs:element name="TopologyTemplate" type="tTopologyTemplate"/>
      </xs:choice>
      <xs:element name="ArtifactTemplates" minOccurs="0">
       <xs:complexType>
        <xs:sequence>
         <xs:element name="ArtifactTemplate" type="tArtifactTemplate"
          maxOccurs="unbounded"/>
        </xs:sequence>
       </xs:complexType>
      </xs:element>
      <xs:element name="NodeTypes" type="tNodeTypes" minOccurs="0"/>
      <xs:element name="RequirementTypes" type="tRequirementTypes"
       minOccurs="0"/>
      <xs:element name="CapabilityTypes" type="tCapabilityTypes"
       minOccurs="0"/>
      <xs:element name="RelationshipTypes" type="tRelationshipTypes"
       minOccurs="0"/>
      <xs:element name="ArtifactTypes" type="tArtifactTypes"
       minOccurs="0"/>
      <xs:element name="Plans" type="tPlans" minOccurs="0"/>
     </xs:sequence>
     <xs:attribute name="id" type="xs:ID" use="required"/>
     <xs:attribute name="name" type="xs:string" use="optional"/>
     <xs:attribute name="targetNamespace" type="xs:anyURI"/>
     <xs:attribute name="substitutableNodeType" type="xs:QName"
      use="optional"/>
    </xs:extension>
   </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="tTags">
   <xs:sequence>
    <xs:element name="Tag" type="tTag" minOccurs="1"
     maxOccurs="unbounded"/>
```



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

<xs:complexType name="tTag">
 <xs:attribute name="name" type="xs:string" use="required"/>
 <xs:attribute name="value" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="tBoundaryDefinitions">
 <xs:sequence>
  <xs:element name="Properties" minOccurs="0">
   <xs:complexType>
    <xs:sequence>
     <xs:any namespace="##other"/>
     <xs:element name="PropertyMappings" minOccurs="0">
      <xs:complexType>
       <xs:sequence>
        <xs:element name="PropertyMapping" type="tPropertyMapping"/>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <xs:element name="PropertyConstraints" minOccurs="0">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="PropertyConstraint" type="tPropertyConstraint"
      maxOccurs="unbounded"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <xs:element name="Requirements" minOccurs="0">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="Requirement" type="tRequirementRef"
      maxOccurs="unbounded"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <xs:element name="Capabilities" minOccurs="0">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="Capability" type="tCapabilityRef"
      maxOccurs="unbounded"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <xs:element name="Policies" minOccurs="0">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="Policy" type="tPolicy" maxOccurs="unbounded"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
</xs:complexType>
```

```
<xs:complexType name="tPropertyMapping">
 <xs:attribute name="serviceTemplatePropertyRef" type="xs:string"
  use="required"/>
 <xs:attribute name="targetObjectRef" type="xs:IDREF" use="required"/>
 <xs:attribute name="targetPropertyRef" type="xs:string"
  use="required"/>
</xs:complexType>

<xs:complexType name="tRequirementRef">
 <xs:attribute name="name" type="xs:string" use="optional"/>
 <xs:attribute name="ref" type="xs:IDREF" use="required"/>
</xs:complexType>

<xs:complexType name="tCapabilityRef">
 <xs:attribute name="name" type="xs:string" use="optional"/>
 <xs:attribute name="ref" type="xs:IDREF" use="required"/>
</xs:complexType>

<xs:element name="EntityType" type="tEntityType" abstract="true"/>
<xs:complexType name="tEntityType" abstract="true">
 <xs:complexContent>
  <xs:extension base="tExtensibleElements">
   <xs:sequence>
    <xs:element name="DerivedFrom" minOccurs="0">
     <xs:complexType>
      <xs:attribute name="typeRef" type="xs:QName" use="required"/>
     </xs:complexType>
    </xs:element>
    <xs:element name="PropertiesDefinition" minOccurs="0">
     <xs:complexType>
      <xs:attribute name="element" type="xs:QName"/>
      <xs:attribute name="type" type="xs:QName"/>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
   <xs:attribute name="id" type="xs:ID" use="required"/>
   <xs:attribute name="abstract" type="tBoolean" default="no"/>
   <xs:attribute name="final" type="tBoolean" default="no"/>
   <xs:attribute name="targetNamespace" type="xs:anyURI"
    use="optional"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:element name="EntityTemplate" type="tEntityTemplate"
 abstract="true"/>
<xs:complexType name="tEntityTemplate" abstract="true">
 <xs:complexContent>
  <xs:extension base="tExtensibleElements">
   <xs:sequence>
    <xs:element name="Properties" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:any namespace="##other" processContents="lax"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="PropertyConstraints" minOccurs="0">
```



```
      <xs:complexType>
       <xs:sequence>
        <xs:element name="PropertyConstraint"
         type="tPropertyConstraint" maxOccurs="unbounded"/>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required"/>
    <xs:attribute name="type" type="xs:QName" use="required"/>
   </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:element name="NodeTemplate" type="tNodeTemplate"/>
<xs:complexType name="tNodeTemplate">
 <xs:complexContent>
  <xs:extension base="tEntityTemplate">
   <xs:sequence>
    <xs:element name="Requirements" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Requirement" type="tRequirement"
        maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="Capabilities" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Capability" type="tCapability"
        maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="Policies" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Policy" type="tPolicy"
        maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="DeploymentArtifacts" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="DeploymentArtifact"
        type="tDeploymentArtifact" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="ImplementationArtifacts" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="ImplementationArtifact"
        type="tImplementationArtifact" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:complexType>
```



```
     </xs:element>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="optional"/>
    <xs:attribute name="minInstances" type="xs:integer" use="optional"
     default="1"/>
    <xs:attribute name="maxInstances" use="optional" default="1">
     <xs:simpleType>
      <xs:union>
       <xs:simpleType>
        <xs:restriction base="xs:nonNegativeInteger">
         <xs:pattern value="([1-9]+[0-9]*)"/>
        </xs:restriction>
       </xs:simpleType>
       <xs:simpleType>
        <xs:restriction base="xs:string">
         <xs:enumeration value="unbounded"/>
        </xs:restriction>
       </xs:simpleType>
      </xs:union>
     </xs:simpleType>
    </xs:attribute>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>

 <xs:element name="TopologyTemplate" type="tTopologyTemplate"/>
 <xs:complexType name="tTopologyTemplate">
  <xs:complexContent>
   <xs:extension base="tTopologyElementCollection"/>
  </xs:complexContent>
 </xs:complexType>
 <xs:element name="GroupTemplate" type="tGroupTemplate"/>
 <xs:complexType name="tGroupTemplate">
  <xs:complexContent>
   <xs:extension base="tTopologyElementCollection">
    <xs:sequence>
     <xs:element name="Policies" minOccurs="0">
      <xs:complexType>
       <xs:sequence>
        <xs:element name="Policy" type="tPolicy"
         maxOccurs="unbounded"/>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
    </xs:sequence>
    <xs:attribute name="minInstances" type="xs:integer" use="optional"
     default="1"/>
    <xs:attribute name="maxInstances" use="optional" default="1">
     <xs:simpleType>
      <xs:union>
       <xs:simpleType>
        <xs:restriction base="xs:nonNegativeInteger">
         <xs:pattern value="([1-9]+[0-9]*)"/>
        </xs:restriction>
       </xs:simpleType>
       <xs:simpleType>
        <xs:restriction base="xs:string">
         <xs:enumeration value="unbounded"/>
```



```
        </xs:restriction>
       </xs:simpleType>
      </xs:union>
     </xs:simpleType>
    </xs:attribute>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>

 <xs:complexType name="tTopologyElementCollection">
  <xs:complexContent>
   <xs:extension base="tExtensibleElements">
    <xs:choice maxOccurs="unbounded">
     <xs:element name="NodeTemplate" type="tNodeTemplate"/>
     <xs:element name="RelationshipTemplate"
      type="tRelationshipTemplate"/>
     <xs:element name="GroupTemplate" type="tGroupTemplate"/>
    </xs:choice>
    <xs:attribute name="id" type="xs:ID" use="required"/>
    <xs:attribute name="name" type="xs:string" use="optional"/>
    <xs:attribute name="targetNamespace" type="xs:anyURI"/>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>

 <xs:element name="RelationshipTypes" type="tRelationshipTypes"/>
 <xs:complexType name="tRelationshipTypes">
  <xs:sequence>
   <xs:element name="RelationshipType" type="tRelationshipType"
    maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="targetNamespace" type="xs:anyURI"/>
 </xs:complexType>

 <xs:element name="RelationshipType" type="tRelationshipType"/>
 <xs:complexType name="tRelationshipType">
  <xs:complexContent>
   <xs:extension base="tEntityType">
    <xs:sequence>
     <xs:element name="InstanceStates"
      type="tTopologyElementInstanceStates" minOccurs="0"/>
     <xs:element name="SourceInterfaces" minOccurs="0">
      <xs:complexType>
       <xs:sequence>
        <xs:element name="Interface" type="tInterface"
         maxOccurs="unbounded"/>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
     <xs:element name="TargetInterfaces" minOccurs="0">
      <xs:complexType>
       <xs:sequence>
        <xs:element name="Interface" type="tInterface"
         maxOccurs="unbounded"/>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
     <xs:element name="ValidSource" minOccurs="0">
```



```
      <xs:complexType>
       <xs:attribute name="typeRef" type="xs:QName" use="required"/>
      </xs:complexType>
     </xs:element>
     <xs:element name="ValidTarget" minOccurs="0">
      <xs:complexType>
       <xs:attribute name="typeRef" type="xs:QName" use="required"/>
      </xs:complexType>
     </xs:element>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="optional"/>
   </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:element name="RelationshipTemplate" type="tRelationshipTemplate"/>
<xs:complexType name="tRelationshipTemplate">
 <xs:complexContent>
  <xs:extension base="tEntityTemplate">
   <xs:sequence>
    <xs:element name="SourceElement">
     <xs:complexType>
      <xs:attribute name="ref" type="xs:IDREF" use="required"/>
     </xs:complexType>
    </xs:element>
    <xs:element name="TargetElement">
     <xs:complexType>
      <xs:attribute name="ref" type="xs:IDREF" use="optional"/>
      <xs:attribute name="externalRef" type="xs:QName"
       use="optional"/>
     </xs:complexType>
    </xs:element>
    <xs:element name="RelationshipConstraints" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="RelationshipConstraint"
        maxOccurs="unbounded">
        <xs:complexType>
         <xs:sequence>
          <xs:any namespace="##other" processContents="lax"
           minOccurs="0"/>
         </xs:sequence>
         <xs:attribute name="constraintType" type="xs:anyURI"
          use="required"/>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
   <xs:attribute name="name" type="xs:string" use="optional"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:element name="NodeTypes" type="tNodeTypes"/>
<xs:complexType name="tNodeTypes">
 <xs:sequence>
```



```
   <xs:element name="NodeType" type="tNodeType" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="targetNamespace" type="xs:anyURI"/>
</xs:complexType>

<xs:element name="NodeType" type="tNodeType"/>
<xs:complexType name="tNodeType">
 <xs:complexContent>
  <xs:extension base="tEntityType">
   <xs:sequence>
    <xs:element name="RequirementDefinitions" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="RequirementDefinition"
        type="tRequirementDefinition" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="CapabilityDefinitions" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="CapabilityDefinition"
        type="tCapabilityDefinition" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="InstanceStates"
     type="tTopologyElementInstanceStates" minOccurs="0"/>
    <xs:element name="Interfaces" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Interface" type="tInterface"
        maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="Policies" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Policy" type="tPolicy"
        maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
    <xs:element name="DeploymentArtifacts" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="DeploymentArtifact"
        type="tDeploymentArtifact" maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
   <xs:attribute name="name" type="xs:string" use="optional"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
```

```
<xs:element name="RequirementTypes" type="tRequirementTypes"/>
<xs:complexType name="tRequirementTypes">
 <xs:sequence>
  <xs:element name="RequirementType" type="tRequirementType"
   maxOccurs="unbounded"/>
 </xs:sequence>
 <xs:attribute name="targetNamespace" type="xs:anyURI"
  use="optional"/>
</xs:complexType>

<xs:element name="RequirementType" type="tRequirementType"/>
<xs:complexType name="tRequirementType">
 <xs:complexContent>
  <xs:extension base="tEntityType">
   <xs:attribute name="name" type="xs:string" use="optional"/>
   <xs:attribute name="requiredCapabilityType" type="xs:QName"
    use="optional"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:element name="RequirementDefinition"
 type="tRequirementDefinition"/>
<xs:complexType name="tRequirementDefinition">
 <xs:complexContent>
  <xs:extension base="tExtensibleElements">
   <xs:sequence>
    <xs:element name="Constraints" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Constraint" type="tConstraint"
        maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
   <xs:attribute name="name" type="xs:string" use="required"/>
   <xs:attribute name="requirementType" type="xs:QName"
    use="required"/>
   <xs:attribute name="lowerBound" type="xs:integer" use="optional"
    default="1"/>
   <xs:attribute name="upperBound" use="optional" default="1">
    <xs:simpleType>
     <xs:union>
      <xs:simpleType>
       <xs:restriction base="xs:nonNegativeInteger">
        <xs:pattern value="([1-9]+[0-9]*)"/>
       </xs:restriction>
      </xs:simpleType>
      <xs:simpleType>
       <xs:restriction base="xs:string">
        <xs:enumeration value="unbounded"/>
       </xs:restriction>
      </xs:simpleType>
     </xs:union>
    </xs:simpleType>
   </xs:attribute>
  </xs:extension>
```



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

<xs:element name="Requirement" type="tRequirement"/>
<xs:complexType name="tRequirement">
 <xs:complexContent>
  <xs:extension base="tEntityTemplate">
   <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:element name="CapabilityTypes" type="tCapabilityTypes"/>
<xs:complexType name="tCapabilityTypes">
 <xs:sequence>
  <xs:element name="CapabilityType" type="tCapabilityType"
   maxOccurs="unbounded"/>
 </xs:sequence>
 <xs:attribute name="targetNamespace" type="xs:anyURI"
  use="optional"/>
</xs:complexType>

<xs:element name="CapabilityType" type="tCapabilityType"/>
<xs:complexType name="tCapabilityType">
 <xs:complexContent>
  <xs:extension base="tEntityType">
   <xs:attribute name="name" type="xs:string" use="optional"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:element name="CapabilityDefinition" type="tCapabilityDefinition"/>
<xs:complexType name="tCapabilityDefinition">
 <xs:complexContent>
  <xs:extension base="tExtensibleElements">
   <xs:sequence>
    <xs:element name="Constraints" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="Constraint" type="tConstraint"
        maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
   <xs:attribute name="name" type="xs:string" use="required"/>
   <xs:attribute name="capabilityType" type="xs:QName"
    use="required"/>
   <xs:attribute name="lowerBound" type="xs:integer" use="optional"
    default="1"/>
   <xs:attribute name="upperBound" use="optional" default="1">
    <xs:simpleType>
     <xs:union>
      <xs:simpleType>
       <xs:restriction base="xs:nonNegativeInteger">
        <xs:pattern value="([1-9]+[0-9]*)"/>
       </xs:restriction>
      </xs:simpleType>
```



```
      <xs:simpleType>
       <xs:restriction base="xs:string">
        <xs:enumeration value="unbounded"/>
       </xs:restriction>
      </xs:simpleType>
     </xs:union>
    </xs:simpleType>
   </xs:attribute>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:element name="Capability"/>
<xs:complexType name="tCapability">
 <xs:complexContent>
  <xs:extension base="tEntityTemplate">
   <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:element name="ArtifactTypes" type="tArtifactTypes"/>
<xs:complexType name="tArtifactTypes">
 <xs:sequence>
  <xs:element name="ArtifactType" type="tArtifactType"
   maxOccurs="unbounded"/>
 </xs:sequence>
 <xs:attribute name="targetNamespace" type="xs:anyURI"
  use="optional"/>
</xs:complexType>

<xs:element name="ArtifactType" type="tArtifactType"/>
<xs:complexType name="tArtifactType">
 <xs:complexContent>
  <xs:extension base="tEntityType">
   <xs:attribute name="name" type="xs:string" use="optional"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:element name="ArtifactTemplate" type="tArtifactTemplate"/>
<xs:complexType name="tArtifactTemplate">
 <xs:complexContent>
  <xs:extension base="tEntityTemplate">
   <xs:sequence>
    <xs:element name="ArtifactReferences" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="ArtifactReference" type="tArtifactReference"
        maxOccurs="unbounded"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
   <xs:attribute name="name" type="xs:string" use="optional"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
```



```
<xs:element name="DeploymentArtifact" type="tDeploymentArtifact"/>
<xs:complexType name="tDeploymentArtifact">
 <xs:complexContent>
  <xs:extension base="tExtensibleElements">
   <xs:attribute name="name" type="xs:string" use="required"/>
   <xs:attribute name="artifactType" type="xs:QName" use="required"/>
   <xs:attribute name="artifactRef" type="xs:QName" use="optional"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:element name="ImplementationArtifact"
 type="tImplementationArtifact"/>
<xs:complexType name="tImplementationArtifact">
 <xs:complexContent>
  <xs:extension base="tExtensibleElements">
   <xs:sequence>
    <xs:element name="RequiredContainerFeatures" minOccurs="0">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="RequiredContainerFeature"
        maxOccurs="unbounded">
        <xs:complexType>
         <xs:attribute name="feature" type="xs:anyURI"
          use="required"/>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
   <xs:attribute name="operationName" type="xs:string"
    use="optional"/>
   <xs:attribute name="artifactType" type="xs:QName" use="required"/>
   <xs:attribute name="artifactRef" type="xs:QName" use="optional"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:element name="Plans" type="tPlans"/>
<xs:complexType name="tPlans">
 <xs:sequence>
  <xs:element name="Plan" type="tPlan" maxOccurs="unbounded"/>
 </xs:sequence>
 <xs:attribute name="targetNamespace" type="xs:anyURI"/>
</xs:complexType>

<xs:element name="Plan" type="tPlan"/>
<xs:complexType name="tPlan">
 <xs:complexContent>
  <xs:extension base="tExtensibleElements">
   <xs:sequence>
    <xs:element name="Precondition" type="tCondition" minOccurs="0"/>
    <xs:choice>
     <xs:element name="PlanModel">
      <xs:complexType>
       <xs:sequence>
```



```
        <xs:any namespace="##other" processContents="lax"/>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
     <xs:element name="PlanModelReference">
      <xs:complexType>
       <xs:attribute name="reference" type="xs:anyURI"
        use="required"/>
      </xs:complexType>
     </xs:element>
    </xs:choice>
   </xs:sequence>
   <xs:attribute name="id" type="xs:ID" use="required"/>
   <xs:attribute name="name" type="xs:string" use="optional"/>
   <xs:attribute name="planType" type="xs:anyURI" use="required"/>
   <xs:attribute name="languageUsed" type="xs:anyURI" use="required"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="tPolicy">
 <xs:complexContent>
  <xs:extension base="tExtensibleElements">
   <xs:attribute name="name" type="xs:string" use="required"/>
   <xs:attribute name="type" type="xs:anyURI" use="required"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="tConstraint">
 <xs:sequence>
  <xs:any namespace="##other" processContents="lax"/>
 </xs:sequence>
 <xs:attribute name="constraintType" type="xs:anyURI" use="required"/>
</xs:complexType>

<xs:complexType name="tPropertyConstraint">
 <xs:complexContent>
  <xs:extension base="tConstraint">
   <xs:attribute name="property" type="xs:string" use="required"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="tExtensions">
 <xs:complexContent>
  <xs:extension base="tExtensibleElements">
   <xs:sequence>
    <xs:element name="Extension" type="tExtension"
     maxOccurs="unbounded"/>
   </xs:sequence>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="tExtension">
 <xs:complexContent>
  <xs:extension base="tExtensibleElements">
```



```
    <xs:attribute name="namespace" type="xs:anyURI" use="required"/>
    <xs:attribute name="mustUnderstand" type="tBoolean" use="optional"
     default="yes"/>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>

 <xs:complexType name="tParameter">
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="type" type="xs:string" use="required"/>
  <xs:attribute name="required" type="tBoolean" use="optional"
   default="yes"/>
 </xs:complexType>

 <xs:complexType name="tInterface">
  <xs:sequence>
   <xs:element name="Operation" type="tOperation" minOccurs="0"
    maxOccurs="unbounded"/>
   <xs:element name="ImplementationArtifacts" minOccurs="0">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="ImplementationArtifact"
       type="tImplementationArtifact" maxOccurs="unbounded"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
  <xs:attribute name="name" type="xs:anyURI" use="required"/>
 </xs:complexType>

 <xs:complexType name="tWSDL">
  <xs:attribute name="portType" type="xs:QName" use="required"/>
  <xs:attribute name="operation" type="xs:NCName" use="required"/>
 </xs:complexType>

 <xs:complexType name="tOperation">
  <xs:complexContent>
   <xs:extension base="tExtensibleElements">
    <xs:choice>
     <xs:element name="WSDL" type="tWSDL"/>
     <xs:element name="REST" type="tREST"/>
     <xs:element name="ScriptOperation" type="tScriptOperation"/>
    </xs:choice>
    <xs:attribute name="name" type="xs:NCName" use="required"/>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>

 <xs:complexType name="tREST">
  <xs:sequence>
   <xs:element name="Parameters" minOccurs="0">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="Parameter" maxOccurs="unbounded">
       <xs:complexType>
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="required" type="tBoolean" use="optional"
         default="yes"/>
```



```
      </xs:complexType>
     </xs:element>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <xs:element name="Headers" minOccurs="0">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="Header" maxOccurs="unbounded">
      <xs:complexType>
       <xs:attribute name="name" type="xs:string" use="required"/>
       <xs:attribute name="required" type="tBoolean" use="optional"
        default="yes"/>
      </xs:complexType>
     </xs:element>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
 <xs:attribute name="method" default="GET">
  <xs:simpleType>
   <xs:restriction base="xs:string">
    <xs:enumeration value="GET"/>
    <xs:enumeration value="PUT"/>
    <xs:enumeration value="POST"/>
    <xs:enumeration value="DELETE"/>
   </xs:restriction>
  </xs:simpleType>
 </xs:attribute>
 <xs:attribute name="abs_path" type="xs:anyURI" use="optional"/>
 <xs:attribute name="absoluteURI" type="xs:anyURI" use="optional"/>
 <xs:attribute name="requestBody" type="xs:QName" use="optional"/>
 <xs:attribute name="responseBody" type="xs:QName" use="optional"/>
</xs:complexType>

<xs:complexType name="tScriptOperation">
 <xs:sequence>
  <xs:element name="InputParameters" minOccurs="0">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="InputParameter" type="tParameter"
      maxOccurs="unbounded"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <xs:element name="OutputParameters" minOccurs="0">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="OutputParameter" type="tParameter"
      maxOccurs="unbounded"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
 </xs:sequence>
</xs:complexType>

<xs:complexType name="tCondition">
 <xs:sequence>
```



```
    <xs:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
   </xs:sequence>
   <xs:attribute name="expressionLanguage" type="xs:anyURI"
    use="required"/>
  </xs:complexType>

  <xs:complexType name="tTopologyElementInstanceStates">
   <xs:sequence>
    <xs:element name="InstanceState" maxOccurs="unbounded">
     <xs:complexType>
      <xs:attribute name="state" type="xs:anyURI" use="required"/>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>

  <xs:element name="ArtifactReference" type="tArtifactReference"/>
  <xs:complexType name="tArtifactReference">
   <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element name="Include">
     <xs:complexType>
      <xs:attribute name="pattern" type="xs:string" use="required"/>
     </xs:complexType>
    </xs:element>
    <xs:element name="Exclude">
     <xs:complexType>
      <xs:attribute name="pattern" type="xs:string" use="required"/>
     </xs:complexType>
    </xs:element>
   </xs:choice>
   <xs:attribute name="reference" type="xs:anyURI" use="required"/>
  </xs:complexType>

  <xs:simpleType name="tBoolean">
   <xs:restriction base="xs:string">
    <xs:enumeration value="yes"/>
    <xs:enumeration value="no"/>
   </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="importedURI">
   <xs:restriction base="xs:anyURI"/>
  </xs:simpleType>

</xs:schema>
```

Appendix E. Sample

This appendix contains the full sample used in this specification.

E.1 Sample Service Topology Definition

```
01 <ServiceTemplate name="myService"
                    targetNamespace="http://www.example.com/sample">
  <Tags>
   <Tag name=”author” value=”someone@example.com”/>
  </Tags>

  <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"
                          relationshipType="deployedOn">
      <SourceElement id="MyApplication"/>
      <TargetElement id="MyAppServer"/>
    </RelationshipTemplate>

  </TopologyTemplate>

  <NodeTypes>
    <NodeType name="Application">
      <documentation xml:lang="EN">
        A reusable definition of a node type representing an
        application that can be deployed on application servers.
      </documentation>
      <NodeTypeProperties element="ApplicationProperties"/>
      <InstanceStates>
        <InstanceState state="http://www.example.com/started"/>
        <InstanceState state="http://www.example.com/stopped"/>
      </InstanceStates>
      <Interfaces>
        <Interface name="DeploymentInterface">
          <Operation name="DeployApplication">
            <ScriptOperation>
              <InputParameters>
                <InputParamter name="InstanceName"
                               type="string"/>
                <InputParamter name="AppServerHostname"
                               type="string"/>
                <InputParamter name="ContextRoot"
                               type="string"/>
              </InputParameters>
            </ScriptOperation>
          </Operation>
          <ImplementationArtifacts>
            <ImplementationArtifact operationName="DeployApplication"
            type="http://www.example.com/ScriptArtifact/PhythonReference">
              scripts/phython/deployApplication.py
            </ImplementationArtifact>
          </ImplementationArtifacts>
        </Interface>
      </Interfaces
    </NodeType>
    <NodeType name="ApplicationServer"
              targetNamespace="http://www.example.com/sample">
      <NodeTypeProperties element="AppServerProperties"/>
      <Interfaces>
        <Interface name="MyAppServerInterface">
          <Operation name="AcquireNetworkAddress">
            <WSDL portType="my:NetworkPT"
                  operation="AcquireNetworkAddress"/>
          </Operation>
          <Operation name="DeployApplicationServer">
            <WSDL portType="my:AppServerPT"
                  operation="DeployApplicationServer"/>
          </Operation>
          <ImplementationArtifacts>
```



```
            <ImplementationArtifact
             operationName="AcquireNetworkAddress"
             type="http://www.example.com/MyJeeArtifact/EarRef">
              artifacts/jee/MyEAR.ear
            </ImplementationArtifact>
            <ImplementationArtifact
              operationName="DeployApplicationServer"
              type="http://www.example.com/MyJeeArtifact/EarRef">
              artifacts/jee/AppServerManagement.ear
            </ImplementationArtifact>
          </ImplementationArtifacts>
        </Interface>
      </Interfaces>
    </NodeType>
  </NodeTypes>

  <RelationshipTypes>
    <documentation xml:lang="EN">
      A reusable definition of relation that expresses deployment of
      an artifact on a hosting environment.
    </documentation>
    <RelationshipType name="deployedOn">
    </RelationshipType>
  </RelationshipTypes>

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

       <PreCondition expressionLanguage="www.example.com/text"> ?
         Run only if funding is available
       </PreCondition>

       <PlanModel>
         <process name="DeployNewApplication" id="p1">
           <documentation>This process deploys a new instance of the
             sample application.
           </documentation>

           <task id="t1" name="CreateAccount"/>

           <task id="t2" name="AcquireNetworkAddresses"
                 isSequential="false"
                 loopDataInput="t2Input.LoopCounter"/>
             <documentation>Assumption: t2 gets data of type “input”
               as input and this data has a field names “LoopCounter”
               that contains the actual multiplicity of the task.
             </documentation>

           <task id="t3" name="DeployApplicationServer"
                 isSequential="false"
                 loopDataInput="t3Input.LoopCounter"/>

           <task id="t4" name="DeployApplication"
                 isSequential="false"
```



```
                loopDataInput="t4Input.LoopCounter"/>

          <sequenceFlow id="s1" targetRef="t2" sourceRef="t1"/>
          <sequenceFlow id="s2" targetRef="t3" sourceRef="t2"/>
          <sequenceFlow id="s3" targetRef="t4" sourceRef="t3"/>
        </process>
      </PlanModel>
    </Plan>

    <Plan id="RemoveApplication"
      planType="http://docs.oasis-
       open.org/tosca/ns/2011/12/PlanTypes/TerminationPlan"
      languageUsed="http://docs.oasis-
        open.org/wsbpel/2.0/process/executable">
      <PlanModelReference reference="prj:RemoveApp"/>
  </Plan>
</Plans>

</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 `NodeType Interfaces` element; adapted text and examples
wd-05	2012-03-30	Thomas Spatzier, Frank Leymann	Changes for JIRA Issue TOSCA-5: Changed definition of `NodeTemplate` to include `ImplementationArtifact` 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 `abstract` attribute to NodeType for sub-issue 2; Added `final` attribute to NodeType 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 RelationshipType; 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.
wd-10	2012-07-30	Thomas Spatzier, Derek Palma	Changes for JIRA Issue TOSCA-18 and related issues: Introduced concept of Requirements and Capabilities; Restructuring of some paragraphs to improve readability
wd-11	2012-08-25	Thomas Spatzier, Derek Palma, Dale Moberg, Doug Davis	Changes for JIRA Issue TOSCA-13: Clarifying rewording of introduction Changes for JIRA Issue TOSCA-38: Add `substituableNodeType` attribute and `BoundaryDefinitions` to Service Template to allow for Service Template composition. Changes for JIRA Issue TOSCA-41: Add Tags to Service Template as simple means for Service Template versioning; Changes for JIRA Issue TOSCA-47: Use `name` and `targetNamespace` for uniquely identifying TOSCA types; Changes for JIRA Issue TOSCA-48 (partly): implement notational conventions in pseudo schemas

4250