Topology and Orchestration Specification for Cloud Applications Version 1.0

Working Draft 03

12 March 2012

Technical Committee:

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

Chairs:

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

Editors:

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

Declared XML namespaces:

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

Abstract:

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

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

Status:

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

OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Table of Contents

1 Introduction

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

2 Language Design

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

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

2.1 Dependencies on Other Specifications

TOSCA utilizes the following specifications:

- WSDL 1.1
- XML Schema 1.0

and relates to:

- OVF 1.1

2.2 Notational Conventions

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

2.3 Normative References

[RFC2119]	S. Bradner, *Key words for use in RFCs to Indicate Requirement Levels*, http://www.ietf.org/rfc/rfc2119.txt, IETF RFC 2119, March 1997.
[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 - Beta 1, 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 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

2.4 Non-Normative References

[Reference] [Full reference citation]

2.5 Namespaces

This specification uses a number of namespace prefixes throughout; they are listed in Table 1**.** 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
ste	http://docs.oasis-open.org/tosca/ns/2011/12
xs	http://www.w3.org/2001/XMLSchema
wsdl	http://schemas.xmlsoap.org/wsdl/
bpmn	http://www.omg.org/bpmn/2.0

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.

2.6 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.

2.7 Overall Language Structure

A *Service Template* is an XML document that consists of a Topology Template, Node Types, Relationship Types 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, Relationship Types and Plans.

2.7.1 Syntax

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

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

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

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

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

  <NodeTypes>?
         ...
  </NodeTypes>

  <RelationshipTypes>?
         ...
  </RelationshipTypes>

  <Plans>?
         ...
  </Plans>

</ServiceTemplate>
```

2.7.2 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.

- `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.

- `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`.

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

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

- `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`, `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.

Example of the use of a type definition:

```
<Types>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
             elementFormDefault="qualified"
             attributeFormDefault="unqualified">
    <xs:element name="ProjectProperties">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="Owner" type="xs:string"/>
          <xs:element name="ProjectName" type="xs:string"/>
          <xs:element name="AccountID" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:schema>
</Types>
```

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:

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

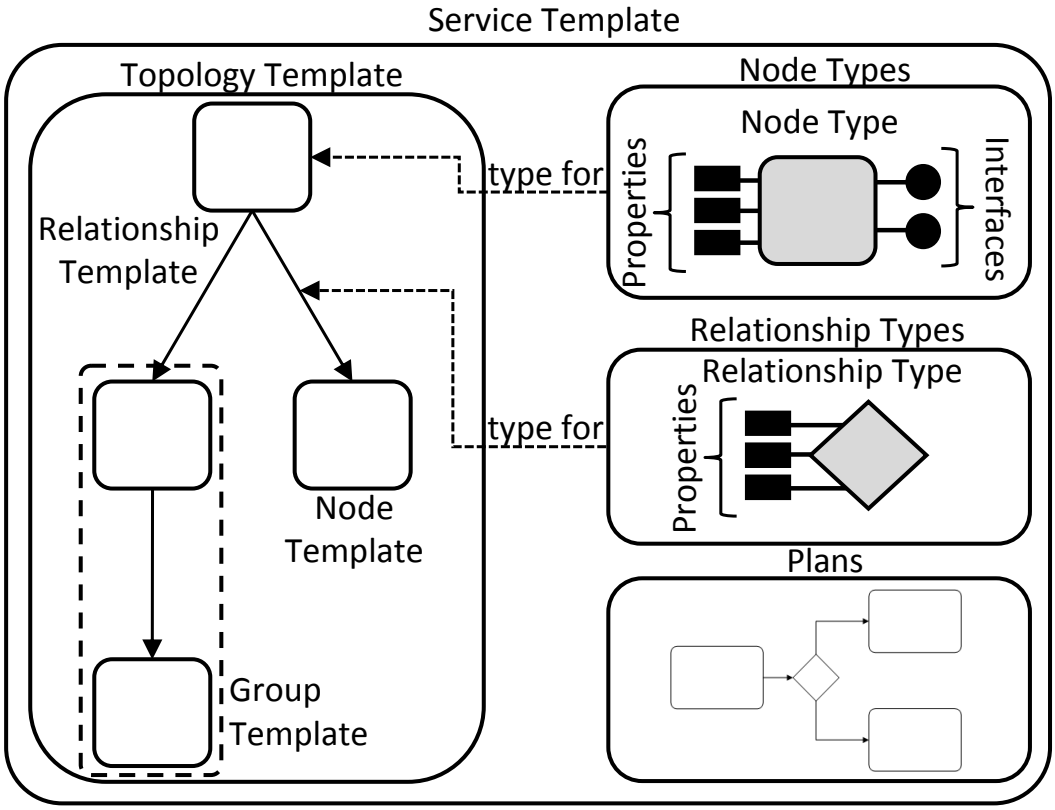

Figure 1: Structural Elements of a Service Template and their Relations

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.

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 observable properties via *Node Type Properties*. A Node Type can inherit properties from another Node Type by means of the `DerivedFrom` element. The functions that can be be performed on (an instance of) a corresponding Node Template are defined by the *Interfaces* of the Node Type. Finally, interfaces supporting management Policies are defined for a Node Type.

4.1 Syntax

```
1    <NodeTypes>?
2
3     <NodeType id="ID"
4               name="string"?>+
5
6      <NodeTypeProperties element="QName"?
7                          type="QName"?/>?
8
9      <DerivedFrom nodeTypeRef="QName"/>?
10
11     <InstanceStates>?
12       <InstanceState state="anyURI">+
13     </InstanceStates>
14
15     <Interfaces>?
16
```

```
      <Interface>+

        (
          <WSDL portType="QName"
                operation="NCName"?>+
         |
          <REST method="GET | PUT | POST | DELETE"
                requestURI="anyURI"
                requestPayload="QName"?
                responsePayload="QName"?>+
         |
          <Operation name="NCame">+

            <InputParameters>?

              <InputParamter name="string"
                             type="string"
                             required="yes|no">+

            </InputParameters>

            <OutputParameters>?

              <OutputParamter name="string"
                              type="string"
                              required="yes|no">+

            </OutputParameters>

            <Implementations>

              <Implementation implementationID="anyURI"?
                              language="anyURI"?>+
               (
                 <ImplementationProper>?
                         code
                 </ImplementationProper>
                |
                 <ImplementationReference ref="anyURI"/>?
               )
             <Implementation>

            </Implementations>
          </Operation>
        )

      </Interface>

    </Interfaces>

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

    <DeploymentArtifacts>?
      <DeploymentArtifact name="string" type="anyURI">+
```

```
75              artifact specific content
76        </DeploymentArtifact>
77     </DeploymentArtifacts>
78
79    </NodeType>
80
81   </NodeTypes>
```

4.2 Properties

The `NodeType` element has the following properties:

- `id`: This attribute specifies the identifier of the Node Type. The identifier of the Node Type MUST be unique within the target namespace.
- `name`: This optional attribute specifies the name of the Node Type.
- `NodeTypeProperties`: These are the observable properties of the Node Type, such as its configuration and state.
- `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.
- `InstanceStates`: This optional element lists the set of states an instance of this Node Type can occupy at runtime.
- `Interfaces`: These are the definitions of functions that can be performed on (instances of) this Node Type.
- `Policies`: The nested list of elements provides information related to a particular management aspect like billing or monitornig.
- `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.

The `NodeTypeProperties` 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.

The `DerivedFrom` element has the following properties:

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

The `InstanceStates` element has the following properties:

- `InstanceState`: specifies a potential state.

The `InstanceState` element has the following properties:

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

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

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

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

The `WSDL` element has the following properties:

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

The `REST` element has the following properties:

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

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

The `Operation` element has the following properties:

- `name`: This is the name of the operation. The name of an operation SHOULD be unique within a Node Type.
- `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 has to 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' proper.
- `OutputParameters`: This optional property contains one or more nested `OutputParameter` elements. Each such element specifies the `name` of the parameter and its `type`.
- `Implementations`: This element contains one or more `Implementation` elements, each of which encompasses either the actual *code* of the implementation of the operation or a reference to the code. The `implementationID` attribute of the `Implementation` element allows for providing different implementions for the same operation – this is necessary because the implementation often depends on the environment the operation will run in. The `language` attribute allows to specify the language of the implementation, e.g. one implementation might be provided as a perl script, another one as php, and so on.

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.

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

  Note, that the combination of name and type SHOULD be unique within the scope of the Node Type.
- The body of this element contains the type-specific content.

  For example, if the `type` attribute contains the value http://docs.oasis-open.org/tosca/ns/2011/12/deploymentArtifacts/ovfRef, the body will contain an XML fragment with a reference to an OVF package and 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.
- Instance States: The set of instance states of this Node Type consists of the set union of the instances states defined by the Nodes Type derived from and the instance states defined by this Node Type. A set of instance states of the same name will be combined into a single instance state of the same name.
- WSDL: The set of WSDL operations of this Node Type consists of the set union of the WSDL operations defined by the Node Type derived from and the WSDL operations defined by the Node Type. A WSDL operation defined by this Node Type substitutes a WSDL operation with the same name and signature of the Node Type derived from.
- REST: The set of REST requests of this Node Type consists of the set union of the REST requests defined by the Nodes Type derived from and the REST requests defined by this Node Type. A REST request defined by this Node Type substitutes a REST request with the same identity of the Node Type derived from.
- Operation: The set of operations of this Node Type consists of the set union of the operations defined by the Nodes Type derived from and the operations defined by this Node Type. An operation defined by this Node Type substitutes an operation with the same name of the Node Type derived from.
- Deployment Artifacts: The set of deployment artifacts of this Node Type consists of the set union of the deployment artifacts defined by the Nodes Type derived from and the deployment artifacts

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

- Policies: The set of policies of this Node Type consists of the set union of the policies defined by the Nodes Type derived from and the policies defined by this Node Type. A policy defined by this Node Type 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.ibm.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.

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

  <NodeTypes>

    <NodeType id="Project" name="My Project">

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

      <NodeTypeProperties element="ProjectProperties"/>

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

      <Interfaces>
        <Interface>
          <Operation name="CreateProject">
            <InputParameters>
              <InputParamter name="ProjectName"
                             type="string"/>
              <InputParamter name="Owner"
                             type="string"/>
              <InputParamter name="AccountID"
                             type="string"/>
            </InputParameters>
            <Implementations>
              <Implementation>
                        ...
              </Implementation>
            </Implementations>
          </Operation>
        </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.my.com/active) or “on hold” (more precise in state “www.my.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 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. A Relationship Type can define observable properties via *Relationship Type Properties*. Furthermore, it defines the potential states an instance of it might reveal at runtime.

5.1 Syntax

```
<RelationshipTypes>

  <RelationshipType id="ID"
                    name="string"?
                    semantics="anyURI"
                    cascadingDeletion="yes|no"?>+

    <RelationshipTypeProperties element="QName"?
                                type="QName"?/>?

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

  </RelationshipType>

</RelationshipTypes>
```

5.2 Properties

The `RelationshipType` element has the following properties:

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

The `RelationshipTypeProperties` element has the following properties:

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

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

687 The `InstanceStates` element has the following properties:

688 - `InstanceState`: specifies a potential state.

689 The `InstanceState` element has the following properties:

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

691 5.3 Example

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

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

716 6 Topology Template

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

722 6.1 Syntax

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

```
      </PropertyDefaults>

      <PropertyConstraints>?

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

      </PropertyConstraints>

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

      <EnvironmentConstraints>?
        <EnvironmentConstraint constraintType="anyURI">+
              constraint type specific content?
        </EnvironmentConstraint>
      </EnvironmentConstraints>

      <DeploymentArtifacts>?
        <DeploymentArtifact name="string" type="anyURI">+
                artifact specific content
        </DeploymentArtifact>
      </DeploymentArtifacts>

   </NodeTemplate>
   |
    <RelationshipTemplate id="ID"
                          name="string"?
                          relationshipType="QName">

       <SourceElement id="IDREF"/>

      ( <TargetElement id="IDREF"/>
       |
        <TargetElementReference id="QName"/>
      )

       <PropertyDefaults>?
             XML fragment
       </PropertyDefaults>

       <PropertyConstraints>?

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

       </PropertyConstraints>

       <RelationshipConstraints>?

         <RelationshipConstraint constraintType="anyURI">+
```

```
                constraint?
            </RelationshipConstraint>

        </RelationshipConstraints>

    </RelationshipTemplate>
   |
    <GroupTemplate id="ID"
                   name="string"?
                   minInstances="int"?
                   maxInstances="int|string"?>

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

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

    </GroupTemplate>
   )+

     </TopologyTemplate>
```

6.2 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 a target element reference).

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.
- `nodeType`: The QName value of this attribute refers to the Node Type providing the type 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`.
- `PropertyDefaults`: 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 `nodeType` 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 Property Defaults 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 separte nested `PropertyConstraint` element. This element contains the actual encoding of the contraint.
- `Policies`: Specifies policies of the Node Template. Each policy is specified by means of a separate nested `Policy` element. This element contains the actual policy specific content of the policy.

  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.
- `EnvironmentConstraints`: The nested `EnvironmentConstraint` elements of the Node Template under definition constrain the runtime environment for the corresponding component of a service. For example, constraints on network security settings of the hosting environment or requirements on the existence of certain resources might be defined within the environment constraints definition of a Node Template.
- `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 `name` attribute of a `DeploymentArtifact` element specifies the name of the artifact. Uniqueness of the name within the scope of the encompassing Node Template SHOULD be guaranteed by the definition. The `type` attribute of a `DeploymentArtifact` element specifies the type of the deployment artifact definition that is related to the Node Template, i.e.

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

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

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

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

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.

The `Policy` element has the following properties:

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

The `EnvironmentConstraint` element has the following properties:

- `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 contraint content.

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.
- `relationshipType`: The QName value of this property refers to the Relationship Type providing the type of the Relationship Template.
- `SourceElement`: The `id` attribute of this element references a Node Template or Group Template within the same Service Template document that is the source of the Relationship Template.

- `TargetElement`: The `id` attribute of this element references a Node Template or Group Template within the same Service Template document that is the target of the Relationship Template.
- `TargetElementReference`: The `id` attribute of this element refers by QName to an imported Node Template or Group Template that is the target of the Relationship Template. The referenced Node Template or Group Template will typically be the root node or root group of the corresponding Topology Template. In some cases a non-root Node Template or non-root Group Template might be referenced to support access to particular resources from a larger service, for example. Either `TargetElement` or `TargetElementReference` MUST be specified but not both.
- `PropertyDefaults`: Specifies initial values for one or more of the Relationship Type properties of the Relationship Type providing the property definitions in the concrete context of the Relationship Template.

  The initial values are specified by providing an instance document of the XML schema of the corresponding Relationship Type properties.

  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 Property Defaults 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. This element contains the actual encoding of the contraint.
- `RelationshipConstraints`: Specifies constraints on the use of the relationship.

  Each constraint is specified by means of a separate nested `RelationshipConstraint` element. This element can contain the actual encoding of the contraint, or its `constraintType` attribute already denotes the constraint itself. 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 any content.

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.

6.3 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”.

```
<ServiceTemplate id="myService"
                 name="My Service"
                 targetNamespace="http://www.ibm.com/sample"
                 xmlns:abc="http://www.ibm.com/sample">

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

  <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>

</ServiceTemplate>
```

7 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.

7.1 Syntax

```
<Plans>

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

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

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

  </Plan>

</Plans>
```

7.2 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.

7.3 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.

7.4 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.

```
<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.my.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>
```

8 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.

9 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 artifacts.

Appendix B. Complete TOSCA Grammar

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

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

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

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

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

   (
    <NodeTemplate id="ID"
                  name="string"?
                  nodeType="QName"
                  minInstances="int"?
                  maxInstances="int|string"?>

      <PropertyDefaults>?
            XML fragment
      </PropertyDefaults>

      <PropertyConstraints>?

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

      </PropertyConstraints>

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

      <EnvironmentConstraints>?
        <EnvironmentConstraint constraintType="anyURI">+
             constraint type specific content?
        </EnvironmentConstraint>
```

```
54        </EnvironmentConstraints>
55
56        <DeploymentArtifacts>?
57          <DeploymentArtifact name="string" type="anyURI">+
58                  artifact specific content
59          </DeploymentArtifact>
60        </DeploymentArtifacts>
61
62     </NodeTemplate>
63    |
64     <RelationshipTemplate id="ID"
65                           name="string"?
66                           relationshipType="QName">+
67
68         <SourceElement id="IDREF"/>
69
70       ( <TargetElement id="IDREF"/>
71        |
72         <TargetElementReference id="QName"/>
73       )
74
75        <PropertyDefaults>?
76              XML fragment
77        </PropertyDefaults>
78
79        <PropertyConstraints>?
80
81          <PropertyConstraint property="string"
82                              constraintType="anyURI">+
83              constraint?
84          </PropertyConstraint>
85
86        </PropertyConstraints>
87
88        <RelationshipConstraints>?
89
90          <RelationshipConstraint constraintType="anyURI">+
91              constraint?
92          </RelationshipConstraint>
93
94        </RelationshipConstraints>
95
96     </RelationshipTemplate>
97    |
98     <GroupTemplate id="ID"
99                    name="string"?
100                      minInstances="int"?
101                      maxInstances="int|string"?>
102
103          (
104           <NodeTemplate ... />
105          |
106           <RelationshipTemplate ... />
107          |
108           <GroupTemplate ... />
109          )+
110
111          <Policies>?
```

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

       </GroupTemplate>
      )+

     </TopologyTemplate>
    )?

     <NodeTypes>?

      <NodeType id="ID"
                name="string"?>+

       <NodeTypeProperties element="QName"?
                           type="QName"?/>?

       <DerivedFrom nodeTypeRef="QName"/>?

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

       <Interfaces>?

         <Interface>+

           (
          <WSDL portType="QName"
                operation="NCName"?>+
            |
          <REST method="GET | PUT | POST | DELETE"
                requestURI="anyURI"
                requestPayload="QName"?
                responsePayload="QName"?>+
            |
          <Operation name="NCame">+

            <InputParameters>?

              <InputParamter name="string"
                             type="string"
                             required="yes|no">+

            </InputParameters>

            <OutputParameters>?

              <OutputParamter name="string"
                              type="string"
                              required="yes|no">+

            </OutputParameters>

            <Implementations>
```

```
170             <Implementation implementationID="anyURI"?
171                             language="anyURI"?>+
172               (
173                 <ImplementationProper>?
174                         code
175                 </ImplementationProper>
176               |
177                 <ImplementationReference ref="anyURI"/>?
178               )
179             <Implementation>
180
181           </Implementations>
182         </Operation>
183        )
184
185       </Interface>
186
187     </Interfaces>
188
189     <DeploymentArtifacts>?
190       <DeploymentArtifact name="string" type="anyURI">+
191             artifact specific content
192       </DeploymentArtifact>
193     </DeploymentArtifacts>
194
195
196     <Policies>?
197
198       <Policy name="string" type="anyURI">+
199          policy specific content
200       </Policy>
201
202     </Policies>
203
204    </NodeType>
205
206   </NodeTypes>
207
208   <RelationshipTypes>?
209
210     <RelationshipType id="ID"
211                       name="string"?
212                       semantics="anyURI"
213                       cascadingDeletion="yes|no"?>+
214
215       <RelationshipTypeProperties element="QName"?
216                                   type="QName"?/>?
217
218       <InstanceStates>?
219         <InstanceState state="anyURI">+
220       </InstanceStates>
221
222     </RelationshipType>
223
224   </RelationshipTypes>
225
226   <Plans>?
227
```

```
228      <Plan id="ID"
229            name="string"?
230            planType="anyURI"
231            languageUsed="anyURI">+
232
233        <PreCondition expressionLanguage="anyURI">?
234            condition
235        </PreCondition>
236
237        ( <PlanModel>
238             actual plan
239          </PlanModel>
240         |
241          <PlanModelReference reference="anyURI"/>
242        )
243
244      </Plan>
245
246     </Plans>
247
248   </ServiceTemplate>
```

Appendix C. TOSCA Schema

```
<?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="Types" minOccurs="0">
      <xs:complexType>
```

```
54        <xs:sequence>
55         <xs:any namespace="##other" processContents="lax" minOccurs="0"
56          maxOccurs="unbounded"/>
57        </xs:sequence>
58       </xs:complexType>
59      </xs:element>
60      <xs:element name="Extensions" minOccurs="0">
61       <xs:complexType>
62        <xs:sequence>
63         <xs:element name="Extension" type="tExtension"
64          maxOccurs="unbounded"/>
65        </xs:sequence>
66       </xs:complexType>
67      </xs:element>
68      <xs:choice minOccurs="0">
69       <xs:element name="TopologyTemplateReference">
70        <xs:complexType>
71         <xs:attribute name="reference" type="xs:QName"/>
72        </xs:complexType>
73       </xs:element>
74       <xs:element name="TopologyTemplate" type="tTopologyTemplate"/>
75      </xs:choice>
76      <xs:element name="NodeTypes" type="tNodeTypes" minOccurs="0"/>
77      <xs:element name="RelationshipTypes" type="tRelationshipTypes"
78       minOccurs="0"/>
79      <xs:element name="Plans" type="tPlans" minOccurs="0"/>
80     </xs:sequence>
81     <xs:attribute name="id" type="xs:ID" use="required"/>
82     <xs:attribute name="name" type="xs:string" use="optional"/>
83     <xs:attribute name="targetNamespace" type="xs:anyURI"/>
84    </xs:extension>
85   </xs:complexContent>
86  </xs:complexType>
87
88  <xs:complexType name="tDeploymentArtifact">
89   <xs:complexContent>
90    <xs:extension base="tExtensibleElements">
91     <xs:attribute name="name" type="xs:string" use="required"/>
92     <xs:attribute name="type" type="xs:anyURI" use="required"/>
93    </xs:extension>
94   </xs:complexContent>
95  </xs:complexType>
96
97  <xs:element name="NodeTemplate" type="tNodeTemplate"/>
98  <xs:complexType name="tNodeTemplate">
99   <xs:complexContent>
100      <xs:extension base="tExtensibleElements">
101       <xs:sequence>
102        <xs:element name="PropertyDefaults" minOccurs="0">
103         <xs:complexType>
104          <xs:sequence>
105           <xs:any namespace="##other" processContents="lax"/>
106          </xs:sequence>
107         </xs:complexType>
108        </xs:element>
109        <xs:element name="PropertyConstraints" minOccurs="0">
110         <xs:complexType>
111          <xs:sequence>
```

```
          <xs:element name="PropertyConstraint"
           type="tPropertyConstraint" 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="EnvironmentConstraints" minOccurs="0">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="EnvironmentConstraint"
           type="tEnvironmentConstraint" maxOccurs="unbounded"/>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
      <xs:attribute name="id" type="xs:ID" use="required"/>
      <xs:attribute name="name" type="xs:string" use="optional"/>
      <xs:attribute name="nodeType" type="xs:QName" use="required"/>
      <xs:attribute name="minInstances" type="xs:int" 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="tPropertyConstraint">
    <xs:sequence>
     <xs:any namespace="##other" processContents="lax" minOccurs="0"/>
```

```
    </xs:sequence>
    <xs:attribute name="property" type="xs:string" use="required"/>
    <xs:attribute name="constraintType" type="xs:anyURI" use="required"/>
  </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:int" 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"/>
```

```
228       <xs:attribute name="name" type="xs:string" use="optional"/>
229       <xs:attribute name="targetNamespace" type="xs:anyURI"/>
230      </xs:extension>
231     </xs:complexContent>
232    </xs:complexType>
233
234    <xs:element name="RelationshipTypes" type="tRelationshipTypes"/>
235    <xs:complexType name="tRelationshipTypes">
236     <xs:sequence>
237      <xs:element name="RelationshipType" type="tRelationshipType"
238       maxOccurs="unbounded"/>
239     </xs:sequence>
240     <xs:attribute name="targetNamespace" type="xs:anyURI"/>
241    </xs:complexType>
242
243    <xs:element name="RelationshipType" type="tRelationshipType"/>
244    <xs:complexType name="tRelationshipType">
245     <xs:complexContent>
246      <xs:extension base="tExtensibleElements">
247       <xs:sequence>
248        <xs:element name="RelationshipTypeProperties" minOccurs="0">
249         <xs:complexType>
250          <xs:attribute name="element" type="xs:QName"/>
251          <xs:attribute name="type" type="xs:QName"/>
252         </xs:complexType>
253        </xs:element>
254        <xs:element name="InstanceStates"
255         type="tTopologyElementInstanceStates" minOccurs="0"/>
256       </xs:sequence>
257       <xs:attribute name="id" type="xs:ID" use="required"/>
258       <xs:attribute name="name" type="xs:string" use="optional"/>
259       <xs:attribute name="semantics" type="xs:anyURI" use="required"/>
260       <xs:attribute name="cascadingDeletion" type="tBoolean"
261        use="optional" default="no"/>
262       <xs:attribute name="targetNamespace" type="xs:anyURI"/>
263      </xs:extension>
264     </xs:complexContent>
265    </xs:complexType>
266
267    <xs:element name="RelationshipTemplate" type="tRelationshipTemplate"/>
268    <xs:complexType name="tRelationshipTemplate">
269     <xs:complexContent>
270      <xs:extension base="tExtensibleElements">
271       <xs:sequence>
272        <xs:element name="SourceElement">
273         <xs:complexType>
274          <xs:attribute name="id" type="xs:IDREF" use="required"/>
275         </xs:complexType>
276        </xs:element>
277        <xs:choice>
278         <xs:element name="TargetElement">
279          <xs:complexType>
280           <xs:attribute name="id" type="xs:IDREF" use="required"/>
281          </xs:complexType>
282         </xs:element>
283         <xs:element name="TargetElementReference">
284          <xs:complexType>
285           <xs:attribute name="id" type="xs:QName" use="required"/>
```

```
          </xs:complexType>
         </xs:element>
        </xs:choice>
        <xs:element name="PropertyDefaults" 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: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="id" type="xs:ID" use="required"/>
       <xs:attribute name="name" type="xs:string" use="optional"/>
       <xs:attribute name="relationshipType" type="xs:QName"
        use="required"/>
      </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="tExtensibleElements">
       <xs:sequence>
        <xs:element name="NodeTypeProperties" minOccurs="0">
```

```
        <xs:complexType>
         <xs:attribute name="element" type="xs:QName"/>
         <xs:attribute name="type" type="xs:QName"/>
        </xs:complexType>
       </xs:element>
       <xs:element name="DerivedFrom" minOccurs="0">
        <xs:complexType>
         <xs:attribute name="nodeTypeRef" type="xs:QName"
          use="required"/>
        </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" maxOccurs="unbounded">
           <xs:complexType>
            <xs:choice>
             <xs:element name="WSDL" type="tWSDL" maxOccurs="unbounded"/>
             <xs:element name="REST" type="tREST" maxOccurs="unbounded"/>
             <xs:element name="Operation" maxOccurs="unbounded">
              <xs:complexType>
               <xs:complexContent>
                <xs:extension base="tOperation"/>
               </xs:complexContent>
              </xs:complexType>
             </xs:element>
            </xs:choice>
           </xs:complexType>
          </xs:element>
         </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="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="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="tEnvironmentConstraint">
 <xs:sequence>
  <xs:any namespace="##other" processContents="lax"/>
 </xs:sequence>
 <xs:attribute name="constraintType" type="xs:anyURI" use="required"/>
</xs:complexType>

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

```
460         <xs:element name="Extension" type="tExtension"
461          maxOccurs="unbounded"/>
462        </xs:sequence>
463       </xs:extension>
464      </xs:complexContent>
465     </xs:complexType>
466
467     <xs:complexType name="tExtension">
468      <xs:complexContent>
469       <xs:extension base="tExtensibleElements">
470        <xs:attribute name="namespace" type="xs:anyURI" use="required"/>
471        <xs:attribute name="mustUnderstand" type="tBoolean" use="optional"
472         default="yes"/>
473       </xs:extension>
474      </xs:complexContent>
475     </xs:complexType>
476
477     <xs:complexType name="tParameter">
478      <xs:attribute name="name" type="xs:string" use="required"/>
479      <xs:attribute name="type" type="xs:string" use="required"/>
480      <xs:attribute name="required" type="tBoolean" use="optional"
481       default="yes"/>
482     </xs:complexType>
483
484     <xs:complexType name="tWSDL">
485      <xs:attribute name="portType" type="xs:QName" use="required"/>
486      <xs:attribute name="operation" type="xs:NCName" use="optional"/>
487     </xs:complexType>
488
489     <xs:complexType name="tOperation">
490      <xs:complexContent>
491       <xs:extension base="tExtensibleElements">
492        <xs:sequence>
493         <xs:element name="Implementations">
494          <xs:complexType>
495           <xs:sequence>
496            <xs:element name="Implementation" maxOccurs="unbounded">
497             <xs:complexType>
498              <xs:choice>
499               <xs:element name="ImplementationProper" type="xs:anyType"
500                minOccurs="0"/>
501               <xs:element name="ImplementationReference" minOccurs="0">
502                <xs:complexType>
503                 <xs:attribute name="ref" type="xs:anyURI"/>
504                </xs:complexType>
505               </xs:element>
506              </xs:choice>
507              <xs:attribute name="implementationID" type="xs:anyURI"/>
508              <xs:attribute name="language" type="xs:anyURI"/>
509             </xs:complexType>
510            </xs:element>
511           </xs:sequence>
512          </xs:complexType>
513         </xs:element>
514         <xs:element name="InputParameters" minOccurs="0">
515          <xs:complexType>
516           <xs:sequence>
517            <xs:element name="InputParameter" type="tParameter"
```

```
518             maxOccurs="unbounded"/>
519           </xs:sequence>
520         </xs:complexType>
521       </xs:element>
522       <xs:element name="OutputParameters" minOccurs="0">
523        <xs:complexType>
524         <xs:sequence>
525          <xs:element name="OutputParameter" type="tParameter"
526           maxOccurs="unbounded"/>
527         </xs:sequence>
528        </xs:complexType>
529       </xs:element>
530      </xs:sequence>
531      <xs:attribute name="name" type="xs:NCName" use="required"/>
532    </xs:extension>
533   </xs:complexContent>
534  </xs:complexType>
535
536  <xs:complexType name="tREST">
537   <xs:attribute name="method" default="GET">
538    <xs:simpleType>
539     <xs:restriction base="xs:string">
540      <xs:enumeration value="GET"/>
541      <xs:enumeration value="PUT"/>
542      <xs:enumeration value="POST"/>
543      <xs:enumeration value="DELETE"/>
544     </xs:restriction>
545    </xs:simpleType>
546   </xs:attribute>
547   <xs:attribute name="requestURI" type="xs:anyURI" use="required"/>
548   <xs:attribute name="requestPayload" type="xs:QName"/>
549   <xs:attribute name="responsePayload" type="xs:QName"/>
550  </xs:complexType>
551
552  <xs:complexType name="tCondition">
553   <xs:sequence>
554    <xs:any processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
555   </xs:sequence>
556   <xs:attribute name="expressionLanguage" type="xs:anyURI"
557    use="required"/>
558  </xs:complexType>
559
560  <xs:complexType name="tTopologyElementInstanceStates">
561   <xs:sequence>
562    <xs:element name="InstanceState" maxOccurs="unbounded">
563     <xs:complexType>
564      <xs:attribute name="state" type="xs:anyURI" use="required"/>
565     </xs:complexType>
566    </xs:element>
567   </xs:sequence>
568  </xs:complexType>
569
570  <xs:simpleType name="tBoolean">
571   <xs:restriction base="xs:string">
572    <xs:enumeration value="yes"/>
573    <xs:enumeration value="no"/>
574   </xs:restriction>
575  </xs:simpleType>
```

```
576
577    <xs:simpleType name="importedURI">
578     <xs:restriction base="xs:anyURI"/>
579    </xs:simpleType>
580
581   </xs:schema>
```

Appendix D. Sample

This appendix contains the full sample used in this specification.

D.1 Sample Service Topology Definition

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

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

  <TopologyTemplate id="SampleApplication">

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

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

    <RelationshipTemplate id="MyDeploymentRelationship"
```

```
                          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.my.com/started"/>
        <InstanceState state="http://www.my.com/stopped"/>
      </InstanceStates>
      <Interfaces>
        <Interface>
          <Operation name="DeployApplication">
            <InputParameters>
              <InputParamter name="InstanceName"
                             type="string"/>
              <InputParamter name="AppServerHostname"
                             type="string"/>
              <InputParamter name="ContextRoot"
                             type="string"/>
            </InputParameters>
            <Implementations>
              <Implementation>
                       ...
              </Implementation>
            </Implementations>
          </Operation>
        </Interface>
      </Interfaces
    </NodeType>
    <NodeType name="ApplicationServer"
              targetNamespace="http://www.ibm.com/sample">
      <NodeTypeProperties element="AppServerProperties"/>
      <Interfaces>
        <Interface>
          <Operation name="AcquireNetworkAddress">
            <OutputParameters>
              <OutputParamter name="Hostname"
                              type="string"/>
              <OutputParamter name="IPAddress"
                              type="string"/>
            </OutputParameters>
            <Implementations>
              <Implementation>
                       ...
              </Implementation>
            </Implementations>
          </Operation>
          <Operation name="DeployApplicationServer">
            <InputParameters>
```

```
              <InputParamter name="Hostname"
                             type="string"/>
              <InputParamter name="IPAddress"
                             type="string"/>
              <InputParamter name="HeapSize"
                             type="int"/>
              <InputParamter name="SoapPort"
                             type="int"/>
            </InputParameters>
            <OutputParameters>
              <OutputParamter name="ServerID"
                              type="string"/>
            </OutputParameters>
            <Implementations>
              <Implementation>
                        ...
              </Implementation>
            </Implementations>
          </Operation>
        </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"
                      semantics="www.my.com/RelSemantics/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.my.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 E. 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.