Topology and Orchestration Specification for Cloud Applications Version 1.0

Committee Specification Draft 07

18 March 2013

Specification URIs

This version:

http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd07/TOSCA-v1.0-csd07.pdf (Authoritative)
http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd07/TOSCA-v1.0-csd07.html
http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd07/TOSCA-v1.0-csd07.doc

Previous version:

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

Latest version:

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

Technical Committee:

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

Chairs:

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

Editors:

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

Additional artifacts:

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

- XML schema: http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd07/schemas/

Declared XML namespace:

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

Abstract:

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

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

Status:

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

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

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

Citation format:

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

[TOSCA-v1.0]

Topology and Orchestration Specification for Cloud Applications Version 1.0. 18 March 2013. OASIS Committee Specification Draft 07. http://docs.oasis-open.org/tosca/TOSCA/v1.0/csd07/TOSCA-v1.0-csd07.html.

Notices

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

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

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

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

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

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

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

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

Table of Contents

1 Introduction

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

2 Language Design

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

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

2.1 Dependencies on Other Specifications

TOSCA utilizes the following specifications:

- XML Schema 1.0

2.2 Notational Conventions

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

This specification follows XML naming and design rules as described in **Error! Reference source not found.**, i.e. uses upper camel-case notation for XML element names and lower camel-case notation for XML attribute names.

2.3 Normative References

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

[RFC 2396] Uniform Resource Identifiers (URI): Generic Syntax, RFC 2396, available via http://www.faqs.org/rfcs/rfc2396.html

[XML Base] XML Base (Second Edition), W3C Recommendation, http://www.w3.org/TR/xmlbase/

[XML Infoset] XML Information Set, W3C Recommendation, http://www.w3.org/TR/2001/REC-xml-infoset-20011024/

[XML Namespaces] Namespaces in XML 1.0 (Second Edition), W3C Recommendation, http://www.w3.org/TR/REC-xml-names/

[XML Schema Part 1] XML Schema Part 1: Structures, W3C Recommendation, October 2004, http://www.w3.org/TR/xmlschema-1/

[XML Schema Part 2] XML Schema Part 2: Datatypes, W3C Recommendation, October 2004, http://www.w3.org/TR/xmlschema-2/

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

2.4 Non-Normative References

[BPEL 2.0] *Web Services Business Process Execution Language Version 2.0*. OASIS Standard. 11 April 2007. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.

[BPMN 2.0] OMG Business Process Model and Notation (BPMN) Version 2.0, http://www.omg.org/spec/BPMN/2.0/

[OVF] Open Virtualization Format Specification Version 1.1.0, http://www.dmtf.org/standards/published_documents/DSP0243_1.1.0.pdf

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

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

2.5 Typographical Conventions

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

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

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

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

2.6 Namespaces

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

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

Table 1: Prefixes and namespaces used in this specification

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

2.7 Language Extensibility

The TOSCA extensibility mechanism allows:

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

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

3 Core Concepts and Usage Pattern

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

3.1 Core Concepts

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

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

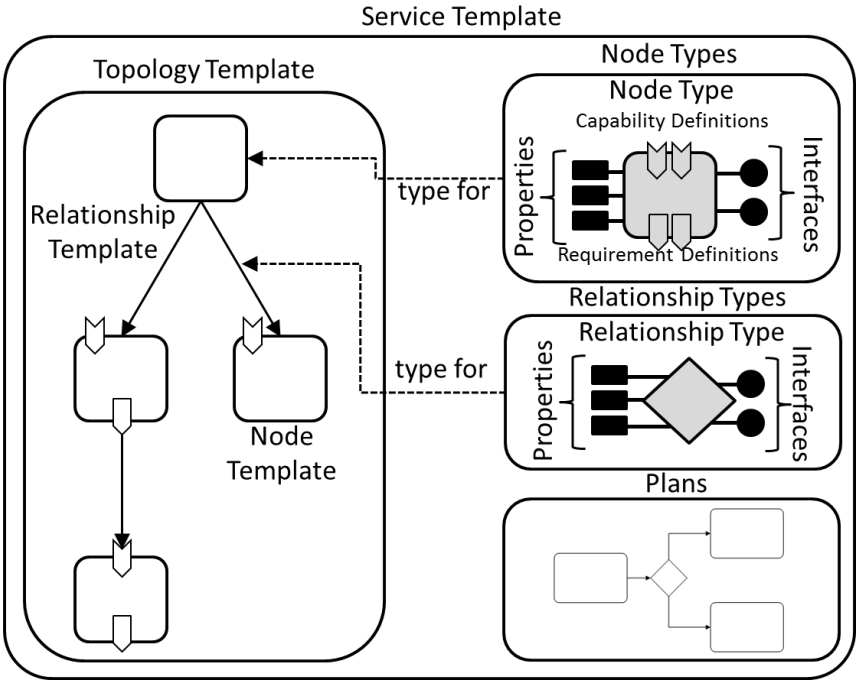

Figure 1: Structural Elements of a Service Template and their Relations

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

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

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

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

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

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

3.2 Use Cases

The specification supports at least the following major use cases.

3.2.1 Services as Marketable Entities

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

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

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

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

3.2.2 Portability of Service Templates

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

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

3.2.3 Service Composition

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

3.2.4 Relation to Virtual Images

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

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

3.3 Service Templates and Artifacts

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

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

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

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

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

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

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

3.4 Requirements and Capabilities

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

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

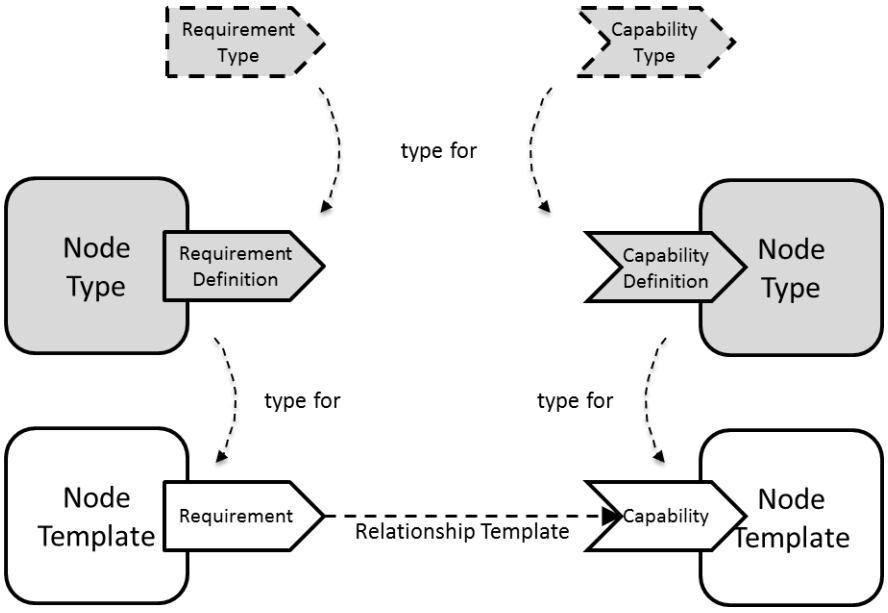

Figure 2: Requirements and Capabilities

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

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

3.5 Composition of Service Templates

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

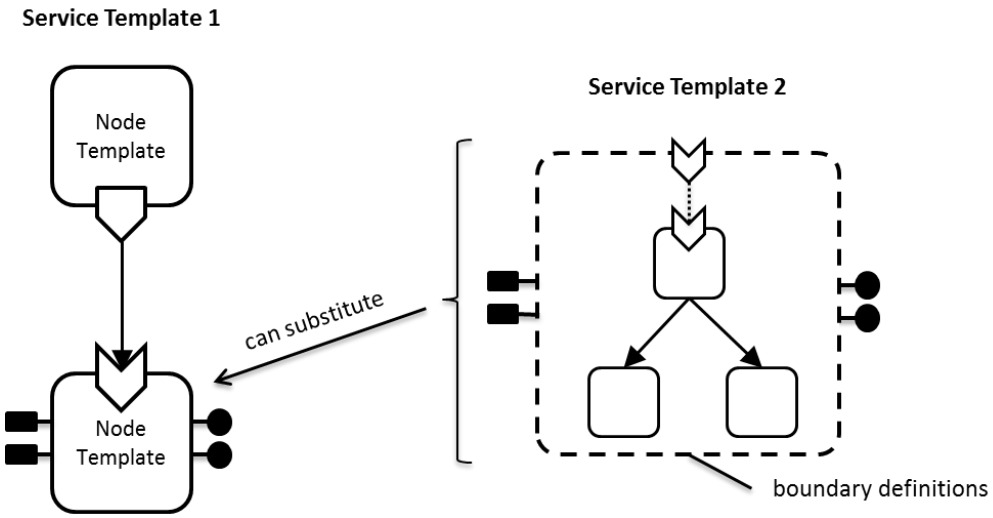

Figure 3: Service Template Composition

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

3.6 Policies in TOSCA

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

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

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

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

3.7 Archive Format for Cloud Applications

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

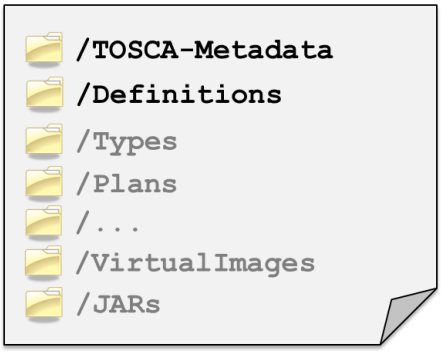

Figure 4: Structure of the CSAR

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

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

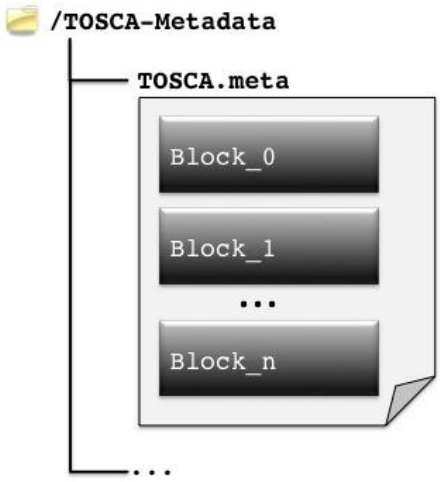

Figure 5: Structure of the TOSCA Meta File

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

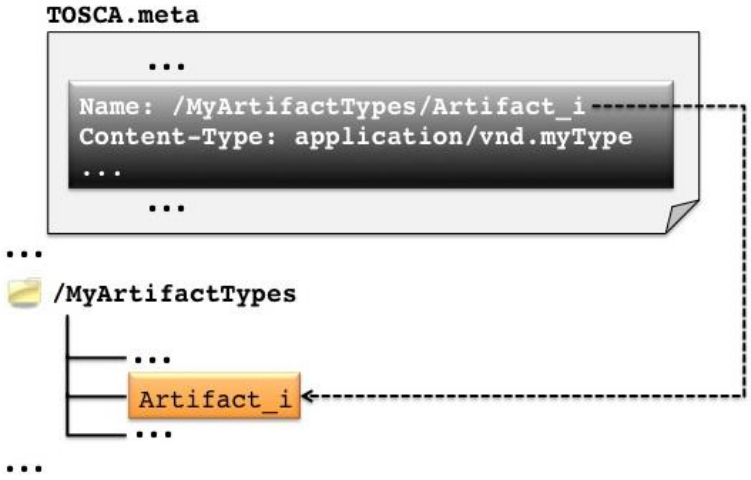

Figure 6: Providing Metadata for Artifacts

4 The TOSCA Definitions Document

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

4.1 XML Syntax

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

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

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

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

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

  (
    <ServiceTemplate> ... </ServiceTemplate>
  |
    <NodeType> ... </NodeType>
  |
    <NodeTypeImplementation> ... </NodeTypeImplementation>
  |
    <RelationshipType> ... </RelationshipType>
  |
    <RelationshipTypeImplementation> ... </RelationshipTypeImplementation>
  |
    <RequirementType> ... </RequirementType>
  |
    <CapabilityType> ... </CapabilityType>
  |
    <ArtifactType> ... </ArtifactType>
  |
    <ArtifactTemplate> ... </ArtifactTemplate>
  |
    <PolicyType> ... </PolicyType>
  |
    <PolicyTemplate> ... </PolicyTemplate>
  ) +

</Definitions>
```

4.2 Properties

The `Definitions` element has the following properties:

- `id`: This attribute specifies the identifier of the Definitions document which MUST be unique within the target namespace.
- `name`: This OPTIONAL attribute specifies a descriptive name of the Definitions document.
- `targetNamespace`: The value of this attribute specifies the target namespace for the Definitions document. All elements defined within the Definitions document will be added to this namespace unless they override this attribute by means of their own `targetNamespace` attributes.
- `Extensions`: This OPTIONAL element specifies namespaces of TOSCA extension attributes and extension elements. If present, the `Extensions` element MUST include at least one `Extension` element.
  The Extension element has the following properties:
  - `namespace`: This attribute specifies the namespace of TOSCA extension attributes and extension elements.
  - `mustUnderstand`: This OPTIONAL attribute specifies 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 mandatory extensions, then the Definitions document 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 TOSCA Definitions, XML Schema definitions, or WSDL definitions. Any number of `Import` elements MAY appear as children of the `Definitions` element.
  The Import element has the following properties:
  - `namespace`: This OPTIONAL attribute specifies an absolute URI that identifies the imported definitions. 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.
  - `location`: This OPTIONAL 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]. 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.

- `importType`: This REQUIRED 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 Definitions document MUST define or import all Node Types, Node Type Implementations, Relationship Types, Relationship Type Implementations, Requirement Type, Capability Types, Artifact Types, Policy Types, WSDL definitions, and XML Schema documents it uses. In order to support the use of definitions from namespaces spanning multiple documents, a Definitions document MAY include more than one import declaration for the same `namespace` and `importType`. Where a Definitions document 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 Definitions document MUST be rejected if the imported documents contain conflicting definitions of a component used by the importing Definitions document.

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 Definitions document, then a document (or namespace) that defines that item MUST be directly imported by the Definitions document. 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 Definitions document. Such definitions are provided within one or more separate Schema definitions (usually `xs:schema` elements). The `Types` element defines XML definitions within a Definitions document without having to define these XML definitions in separate files and importing 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 `Definitions` 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.

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

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

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

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

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

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

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

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

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

Example of use of a `documentation` element:

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


4.3 Example

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

```
01 <Definitions id="MyDefinitions" name="My Definitions"
02   targetNamespace="http://www.example.com/MyDefinitions"
03   xmlns:my="http://www.example.com/MyDefinitions">
04
05   <Import importType="http://www.w3.org/2001/XMLSchema"
06     namespace="http://www.example.com/MyDefinitions">
07
08   <NodeType name="Application">
09     <PropertiesDefinition element="my:ApplicationProperties"/>
10   </NodeType>
11
12   <NodeType name="ApplicationServer">
13     <PropertiesDefinition element="my:ApplicationServerProperties"/>
14   </NodeType>
15
16   <RelationshipType name="ApplicationHostedOnApplicationServer">
17     <ValidSource typeRef="my:Application"/>
18     <ValidTarget typeRef="my:ApplicationServer"/>
19   </RelationshipTemplate>
20
21 </Definitions>
```

5 Service Templates

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

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

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

5.1 XML Syntax

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

```
01 <ServiceTemplate id="xs:ID"
02                  name="xs:string"?
03                  targetNamespace="xs:anyURI"
04                  substitutableNodeType="xs:QName"?>
05
06   <Tags>
07     <Tag name="xs:string" value="xs:string"/> +
08   </Tags> ?
09
10   <BoundaryDefinitions>
11     <Properties>
12       XML fragment
13       <PropertyMappings>
14         <PropertyMapping serviceTemplatePropertyRef="xs:string"
15                          targetObjectRef="xs:IDREF"
16                          targetPropertyRef="xs:string"/> +
17         </PropertyMappings/> ?
18     </Properties> ?
19
20     <PropertyConstraints>
21       <PropertyConstraint property="xs:string"
22                           constraintType="xs:anyURI"> +
23         constraint ?
24       </PropertyConstraint>
25     </PropertyConstraints> ?
26
27     <Requirements>
28       <Requirement name="xs:string"? ref="xs:IDREF"/> +
29     </Requirements> ?
30
31     <Capabilities>
32       <Capability name="xs:string"? ref="xs:IDREF"/> +
33     </Capabilities> ?
34
35     <Policies>
36       <Policy name="xs:string"? policyType="xs:QName"
37               policyRef="xs:QName"?>
38         policy specific content ?
39       </Policy> +
40     </Policies> ?
```

```
41
42      <Interfaces>
43        <Interface name="xs:NCName">
44          <Operation name="xs:NCName">
45            (
46              <NodeOperation nodeRef="xs:IDREF"
47                             interfaceName="xs:anyURI"
48                             operationName="xs:NCName"/>
49            |
50              <RelationshipOperation relationshipRef="xs:IDREF"
51                                     interfaceName="xs:anyURI"
52                                     operationName="xs:NCName"/>
53            |
54              <Plan planRef="xs:IDREF"/>
55            )
56          </Operation> +
57        </Interface> +
58      </Interfaces> ?
59
60    </BoundaryDefinitions> ?
61
62    <TopologyTemplate>
63      (
64        <NodeTemplate id="xs:ID" name="xs:string"? type="xs:QName"
65                      minInstances="xs:integer"?
66                      maxInstances="xs:integer | xs:string"?>
67          <Properties>
68            XML fragment
69          </Properties> ?
70
71          <PropertyConstraints>
72            <PropertyConstraint property="xs:string"
73                                constraintType="xs:anyURI">
74              constraint ?
75            </PropertyConstraint> +
76          </PropertyConstraints> ?
77
78          <Requirements>
79            <Requirement id="xs:ID" name="xs:string" type="xs:QName"> +
80              <Properties>
81                XML fragment
82              <Properties> ?
83              <PropertyConstraints>
84                <PropertyConstraint property="xs:string"
85                                    constraintType="xs:anyURI"> +
86                  constraint ?
87                </PropertyConstraint>
88              </PropertyConstraints> ?
89            </Requirement>
90          </Requirements> ?
91
92          <Capabilities>
93            <Capability id="xs:ID" name="xs:string" type="xs:QName"> +
94              <Properties>
95                XML fragment
96              <Properties> ?
97              <PropertyConstraints>
98                <PropertyConstraint property="xs:string"
```



```
                                  constraintType="xs:anyURI">
                  constraint ?
                </PropertyConstraint> +
              </PropertyConstraints> ?
            </Capability>
          </Capabilities> ?

          <Policies>
            <Policy name="xs:string"? policyType="xs:QName"
                    policyRef="xs:QName"?>
              policy specific content ?
            </Policy> +
          </Policies> ?

          <DeploymentArtifacts>
            <DeploymentArtifact name="xs:string" artifactType="xs:QName"
                                artifactRef="xs:QName"?>
               artifact specific content ?
            </DeploymentArtifact> +
          </DeploymentArtifacts> ?
        </NodeTemplate>
      |
        <RelationshipTemplate id="xs:ID" name="xs:string"?
                              type="xs:QName">
          <Properties>
            XML fragment
          </Properties> ?

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

          <SourceElement ref="xs:IDREF"/>
          <TargetElement ref="xs:IDREF"/>

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

        </RelationshipTemplate>
      ) +
    </TopologyTemplate>

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

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

```



```
157          <InputParameters>
158            <InputParameter name="xs:string" type="xs:string"
159                            required="yes|no"?/> +
160          </InputParameters> ?
161
162          <OutputParameters>
163            <OutputParameter name="xs:string" type="xs:string"
164                             required="yes|no"?/> +
165          </OutputParameters> ?
166
167         (
168           <PlanModel>
169             actual plan
170           </PlanModel>
171          |
172           <PlanModelReference reference="xs:anyURI"/>
173         )
174
175       </Plan> +
176     </Plans> ?
177
178   </ServiceTemplate>
```

5.2 Properties

The `ServiceTemplate` element has the following properties:

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

    Note: The name/value pairs defined in tags have no normative interpretation.
- `BoundaryDefinitions`: This OPTIONAL element specifies the properties the Service Template exposes beyond its boundaries, i.e. properties that can be observed from outside the Service Template. The `BoundaryDefinitions` element has the following properties.
  - `Properties`: This OPTIONAL element specifies global properties of the Service Template in the form of an XML fragment contained in the body of the `Properties` element. Those properties MAY be mapped to properties of components within the

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

- `PropertyMappings`: This OPTIONAL element specifies mappings of one or more of the Service Template's properties to properties of components within the Service Template (e.g. Node Templates, Relationship Templates, etc.). Each property mapping is defined by a separate, nested `PropertyMapping` element. The `PropertyMapping` element has the following properties:
  - `serviceTemplatePropertyRef`:This attribute identifies a property of the Service Template by means of an XPath expression to be evaluated on the XML fragment defining the Service Template's properties.
  - `targetObjectRef`: This attribute specifies the object that provides the property to which the respective Service Template property is mapped. The referenced target object MUST be one of Node Template, Requirement of a Node Template, Capability of a Node Template, or Relationship Template.
  - `targetPropertyRef`: This attribute identifies a property of the target object by means of an XPath expression to be evaluated on the XML fragment defining the target object's properties.

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

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

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

    Note: If the property affected by the property constraint is mapped to a property of a component within the Service Template, the property constraint SHOULD be compatible with any property constraint defined for the mapped property.
  - `constraintType`: This attribute specifies the type of constraint by means of a URI, which defines both the semantic meaning of the constraint as well as the format of the content.
  - The body of the `PropertyConstraint` element provides the actual constraint.
    Note: The body MAY be empty in case the `constraintType` URI already specifies the constraint appropriately. For example, a "read-only" constraint could be expressed solely by the `constraintType` URI.

- `Requirements`: This OPTIONAL element specifies Requirements exposed by the Service Template. Those Requirements correspond to Requirements of Node Templates within the Service Template that are propagated beyond the boundaries of the Service Template. Each Requirement is defined by a separate, nested `Requirement` element.
  The `Requirement` element has the following properties:

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

- `Capabilities`: This OPTIONAL element specifies Capabilities exposed by the Service Template. Those Capabilities correspond to Capabilities of Node Templates within the Service Template that are propagated beyond the boundaries of the Service Template. Each Capability is defined by a separate, nested `Capability` element. The `Capability` element has the following properties:
  - `name`: This OPTIONAL attribute allows for specifying a name of the Capability other than that specified by the referenced Capability of a Node Template.
  - `ref`: This attribute references a `Capability` element of a Node Template within the Service Template.
- `Policies`: This OPTIONAL element specifies global policies of the Service Template related to a particular management aspect. All Policies defined within the `Policies` element MUST be enforced by a TOSCA implementation, i.e. Policies are AND-combined. Each policy is defined by a separate, nested `Policy` element.
  The `Policy` element has the following properties:
  - `name`: This OPTIONAL attribute allows for the definition of a name for the Policy. If specified, this name MUST be unique within the containing `Policies` element.
  - `policyType`: This attribute specifies the type of this Policy. The QName value of this attribute SHOULD correspond to the QName of a `PolicyType` defined in the same Definitions document or in an imported document.

    The `policyType` attribute specifies the artifact type specific content of the `Policy` element body and indicates the type of Policy Template referenced by the Policy via the `policyRef` attribute.
  - `policyRef`: The QName value of this OPTIONAL attribute references a Policy Template that is associated to the Service Template. This Policy Template can be defined in the same TOSCA Definitions document, or it can be defined in a separate document that is imported into the current Definitions document. The type of Policy Template referenced by the `policyRef` attribute MUST be the same type or a sub-type of the type specified in the `policyType` attribute.

    Note: if no Policy Template is referenced, the policy specific content of the `Policy` element alone is assumed to represent sufficient policy specific information in the context of the Service Template.

    Note: while Policy Templates provide invariant information about a non-functional behavior (i.e. information that is context independent, such as the availability class of an availability policy), the `Policy` element defined in a Service Template can provide variant information (i.e. information that is context specific, such as a specific heartbeat frequency for checking availability of a service) in the policy specific body of the `Policy` element.
- `Interfaces`: This OPTIONAL element specifies the interfaces with operations that can be invoked on complete service instances created from the Service Template.
  The `Interfaces` element has the following properties:
  - `Interface`: This element specifies one interfaces exposed by the Service Template.
    The `Interface` element has the following properties:

- `name`: This attribute specifies the name of the interfaces as either a URI or an NCName that MUST be unique in the scope of the Service Template's boundary definitions.
- `Operation`: This element specifies one exposed operation of an interface exposed by the Service Template.

  An operation exposed by a Service Template maps to an internal component of the Service Template which actually provides the operation: it can be mapped to an operation provided by a Node Template (i.e. an operation defined by the Node Type specified in the `type` attribute of the Node Template), it can be mapped to an operation provided by a Relationship Template (i.e. an operation defined by the Relationship Type specified in the `type` attribute of the Relationship Template), or it can be mapped to a Plan of the Service Template.

  When an exposed operation is invoked on a service instance created from the Service Template, the operation or Plan mapped to the exposed operation will actually be invoked.

  The `Operation` element has the following properties:

  - `name`: This attribute specifies the name of the operation, which MUST be unique within the containing interface.
  - `NodeOperation`: This element specifies a reference to an operation of a Node Template.
    The `nodeRef` attribute of this element specifies a reference to the respective Node Template. The specific interface and operation to be mapped to the operation exposed by the Service Template are specified by means of the `interfaceName` and `operationName` attributes, respectively.

    Note: An interface and operation with the specified names MUST be defined by the Node Type (or one of its super types) defined in the `type` attribute of the referenced Node Template.
  - `RelationshipOperation`: This element specifies a reference to an operation of a Relationship Template.
    The `relationshipRef` attribute of this element specifies a reference to the respective Relationship Template. The specific interface and operation to be mapped to the operation exposed by the Service Template are specified by means of the `interfaceName` and `operationName` attributes, respectively.

    Note: An interface and operation with the specified names MUST be defined by the Relationship Type (or one of its super types) defined in the `type` attribute of the referenced Relationship Template.
  - `Plan`: This element specifies by means of its `planRef` attribute a reference to a Plan that provides the implementation of the operation exposed by the Service Template.

    One of `NodeOperation`, `RelationshipOperation` or `Plan` MUST be specified within the `Operation` element.

- `TopologyTemplate`: This element specifies the overall structure of the cloud application defined by the Service Template, i.e. the components it consists of, and the relations between those components. The components of a service are referred to as *Node Templates*, the relations between the components are referred to as *Relationship Templates*.
  The `TopologyTemplate` element has the following properties:
  - `NodeTemplate`: This element specifies a kind of a component making up the cloud application.
    The `NodeTemplate` element has the following properties:
    - `id`: This attribute specifies the identifier of the Node Template. The identifier of the Node Template MUST be unique within the target namespace.
    - `name`: This OPTIONAL attribute specifies the name of the Node Template.
    - `type`: The QName value of this attribute refers to the Node Type providing the type of the Node Template.

      Note: If the Node Type referenced by the `type` attribute of a Node Template is declared as abstract, no instances of the specific Node Template can be created. Instead, a substitution of the Node Template with one having a specialized, derived Node Type has to be done at the latest during the instantiation time of the Node Template.
    - `minInstances`: This integer attribute specifies the minimun number of instances to be created when instantiating the Node Template. The default value of this attribute is 1.The value of `minInstances` MUST NOT be less than 0.
    - `maxInstances`: This attribute specifies the maximum number of instances that can be created when instantiating the Node Template. The default value of this attribute is 1. If the string is set to "`unbounded`", an unbouded number of instances can be created. The value of `maxInstances` MUST be 1 or greater and MUST NOT be less than the value specified for `minInstances`.
    - `Properties`: Specifies initial values for one or more of the Node Type Properties of the Node Type providing the property definitions in the concrete context of the Node Template.
      The initial values are specified by providing an instance document of the XML schema of the corresponding Node Type Properties. This instance document considers the inheritance structure deduced by the `DerivedFrom` property of the Node Type referenced by the `type` attribute of the Node Template.
      The instance document of the XML schema might not validate against the existence constraints of the corresponding schema: not all Node Type properties might have an initial value assigned, i.e. mandatory elements or attributes might be missing in the instance provided by the `Properties` element. Once the defined Node Template has been instantiated, any XML representation of the Node Type properties MUST validate according to the associated XML schema definition.
    - `PropertyConstraints`: Specifies constraints on the use of one or more of the Node Type Properties of the Node Type providing the property definitions for the Node Template. Each constraint is specified by means of a separate nested `PropertyConstraint` element.
      The `PropertyConstraint` element has the following properties:

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

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

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

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

▪ `Policies`: This OPTIONAL element specifies policies associated with the Node Template. All Policies defined within the `Policies` element MUST be enforced by a TOSCA implementation, i.e. Policies are AND-combined. Each policy is specified by means of a separate nested `Policy` element.
The `Policy` element has the following properties:

- `name`: This OPTIONAL attribute allows for the definition of a name for the Policy. If specified, this name MUST be unique within the containing `Policies` element.
- `policyType`: This attribute specifies the type of this Policy. The QName value of this attribute SHOULD correspond to the QName of a `PolicyType` defined in the same Definitions document or in an imported document.

  The `policyType` attribute specifies the artifact type specific content of the `Policy` element body and indicates the type of Policy Template referenced by the Policy via the `policyRef` attribute.
- `policyRef`: The QName value of this OPTIONAL attribute references a Policy Template that is associated to the Node Template. This Policy Template can be defined in the same TOSCA Definitions document, or it can be defined in a separate document that is imported into the current Definitions document. The type of Policy Template referenced by the `policyRef` attribute MUST be the same type or a sub-type of the type specified in the `policyType` attribute.

  Note: if no Policy Template is referenced, the policy specific content of the `Policy` element alone is assumed to represent sufficient policy specific information in the context of the Node Template.

Note: while Policy Templates provide invariant information about a non-functional behavior (i.e. information that is context independent, such as the availability class of an availability policy), the `Policy` element defined in a Node Template can provide variant information (i.e. information that is context specific, such as a specific heartbeat frequency for checking availability of a component) in the policy specific body of the `Policy` element.

- `DeploymentArtifacts`: This element specifies the deployment artifacts relevant for the Node Template under definition. Its nested `DeploymentArtifact` elements specify details about individual deployment artifacts.
  The `DeploymentArtifact` element has the following properties:
  - `name`: This attribute specifies the name of the artifact. Uniqueness of the name within the scope of the encompassing Node Template SHOULD be guaranteed by the definition.
  - `artifactType`: This attribute specifies the type of this artifact. The QName value of this attribute SHOULD correspond to the QName of an `ArtifactType` defined in the same Definitions document or in an imported document.

    The `artifactType` attribute specifies the artifact type specific content of the `DeploymentArtifact` element body and indicates the type of Artifact Template referenced by the Deployment Artifact via the `artifactRef` attribute.
  - `artifactRef`: This OPTIONAL attribute contains a QName that identifies an Artifact Template to be used as deployment artifact. This Artifact Template can be defined in the same Definitions document or in a separate, imported document.
    The type of Artifact Template referenced by the `artifactRef` attribute MUST be the same type or a sub-type of the type specified in the `artifactType` attribute.

    Note: if no Artifact Template is referenced, the artifact type specific content of the `DeploymentArtifact` element alone is assumed to represent the actual artifact. For example, the contents of a simple config file could be defined in place within the `DeploymentArtifact` element.

    Note, that a deployment artifact specified with the Node Template under definition overrides any deployment artifact of the same `name` and the same `artifactType` (or any Artifact Type it is derived from) specified with the Node Type Implementation implementing the Node Type given as value of the `type` attribute of the Node Template under definition. Otherwise, the deployment artifacts of Node Type Implementations and the deployment artifacts defined with the Node Template are combined.

- `RelationshipTemplate`: This element specifies a kind of relationship between the components of the cloud application. For each specified Relationship Template the

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

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

  Note: If the Relationship Type referenced by the `type` attribute of a Relationship Template is declared as abstract, no instances of the specific Relationship Template can be created. Instead, a substitution of the Relationship Template with one having a specialized, derived Relationship Type has to be done at the latest during the instantiation time of the Relationship Template.
- `Properties`: Specifies initial values for one or more of the Relationship Type Properties of the Relationship Type providing the property definitions in the concrete context of the Relationship Template.
  The initial values are specified by providing an instance document of the XML schema of the corresponding Relationship Type Properties. This instance document considers the inheritance structure deduced by the `DerivedFrom` property of the Relationship Type referenced by the `type` attribute of the Relationship Template.
  The instance document of the XML schema might not validate against the existence constraints of the corresponding schema: not all Relationship Type properties might have an initial value assigned, i.e. mandatory elements or attributes might be missing in the instance provided by the `Properties` element. Once the defined Relationship Template has been instantiated, any XML representation of the Relationship Type properties MUST validate according to the associated XML schema definition.
- `PropertyConstraints`: Specifies constraints on the use of one or more of the Relationship Type Properties of the Relationship Type providing the property definitions for the Relationship Template. Each constraint is specified by means of a separate nested `PropertyConstraint` element.
  The `PropertyConstraint` element has the following properties:
  - `property`: The string value of this property is an XPath expression pointing to the property within the Relationship Type Properties document that is constrained within the context of the Relationship Template. More than one constraint MUST NOT be defined for each property.
  - `constraintType`: The constraint type is specified by means of a URI, which defines both the semantic meaning of the constraint as well as the format of the content.

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

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

- `SourceElement`: This element specifies the origin of the relationship represented by the current Relationship Template.
  The `SourceElement` element has the following property:

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

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

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

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

- `TargetElement`: This element specifies the target of the relationship represented by the current Relationship Template.
  The TargetElement element has the following property:

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

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

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

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

- `RelationshipConstraints`: This element specifies a list of constraints on the use of the relationship in separate nested `RelationshipConstraint` elements.
  The `RelationshipConstraint` element has the following properties:

- `constraintType`: This attribute specifies the type of relationship constraint by means of a URI. Depending on the type, the body of the `RelationshipConstraint` element might contain type specific content that further details the actual constraint.

- `Plans`: This element specifies the operational behavior of the service. A `Plan` contained in the `Plans` element can specify how to create, terminate or manage the service.
  The `Plan` element has the following properties:
  - `id`: This attribute specifies the identifier of the Plan. The identifier of the Plan MUST be unique within the target namespace.
  - `name`: This OPTIONAL attribute specifies the name of the Plan.
  - `planType`: The value of the attribute specifies the type of the plan as an indication on what the effect of executing the plan on a service will have. The plan type is specified by means of a URI, allowing for an extensibility mechanism for authors of service templates to define new plan types over time.
    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.
  - `planLanguage`: This attribute denotes the process modeling language (or metamodel) used to specify the plan. For example, "http://www.omg.org/spec/BPMN/20100524/MODEL" would specify that BPMN 2.0 has been used to model the plan.

    TOSCA does not specify a separate metamodel for defining plans. Instead, it is assumed that a process modelling language (a.k.a. metamodel) like BPEL [BPEL 2.0] or BPMN [BPMN 2.0] is used to define plans. The specification favours the use of BPMN for modeling plans.
  - `Precondition`: This OPTIONAL element specifies a condition that needs to be satisfied in order for the plan to be executed. The `expressionLanguage` attribute of this element specifies the expression language the nested condition is provided in.

    Typically, the precondition will be an expression in the instance state attribute of some of the node templates or relationship templates of the topology template. It will be evaluated based on the actual values of the corresponding attributes at the time the plan is requested to be executed. Note, that any other kind of pre-condition is allowed.
  - `InputParameters`: This OPTIONAL property contains a list of one or more input parameter definitions for the Plan, each defined in a nested, separate `InputParameter` element.
    The `InputParameter` element has the following properties:

- `name`: This attribute specifies the name of the input parameter, which MUST be unique within the set of input parameters defined for the operation.
- `type`: This attribute specifies the type of the input parameter.
- `required`: This OPTIONAL attribute specifies whether or not the input parameter is REQUIRED (`required` attribute with a value of "yes" – default) or OPTIONAL (`required` attribute with a value of "no").

- `OutputParameters`: This OPTIONAL property contains a list of one or more output parameter definitions for the Plan, each defined in a nested, separate `OutputParameter` element.
  The `OutputParameter` element has the following properties:
  - `name`: This attribute specifies the name of the output parameter, which MUST be unique within the set of output parameters defined for the operation.
  - `type`: This attribute specifies the type of the output parameter.
  - `required`: This OPTIONAL attribute specifies whether or not the output parameter is REQUIRED (`required` attribute with a value of "yes" – default) or OPTIONAL (`required` attribute with a value of "no").
- `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.

5.3 Example

The following Service Template defines a Topology Template containing two Node Templates called "MyApplication" and "MyAppServer". These Node Templates have the node types "Application" and "ApplicationServer". The Node Template "MyApplication" is instantiated exactly once. Two of its Node Type Properties are initialized by a corresponding `Properties` 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 "MyHostedRelationship"; the behavior and semantics of the Relationship Template is defined in the Relationship Type "HostedOn", saying that "MyApplication" is hosted on "MyAppServer". The Service Template further defines a Plan "UpdateApplication" for performing an update of the "MyApplication" application hosted on the application server. This Plan refers to a BPMN 2.0 process definition contained in a separate file.

```
01 <ServiceTemplate id="MyService"
02                  name="My Service">
03
04   <TopologyTemplate>
05
06     <NodeTemplate id="MyApplication"
07                   name="My Application"
08                   type="my:Application">
09       <Properties>
10         <ApplicationProperties>
11           <Owner>Frank</Owner>
12           <InstanceName>Thomas' favorite application</InstanceName>
13         </ApplicationProperties>
14        </Properties>
```



```
    </NodeTemplate>

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

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

  </TopologyTemplate>

  <Plans>
    <Plan id="UpdateApplication"
        planType="http://www.example.com/UpdatePlan"
        planLanguage="http://www.omg.org/spec/BPMN/20100524/MODEL">
      <PlanModelReference reference="plans:UpdateApp"/>
    </Plan>
  </Plans>

</ServiceTemplate>
```

6 Node Types

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

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

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

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

6.1 XML Syntax

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

```
01 <NodeType name="xs:NCName" targetNamespace="xs:anyURI"?
02           abstract="yes|no"? final="yes|no"?>
03
04   <Tags>
05     <Tag name="xs:string" value="xs:string"/> +
06   </Tags> ?
07
08   <DerivedFrom typeRef="xs:QName"/> ?
09
10   <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
11
12   <RequirementDefinitions>
13     <RequirementDefinition name="xs:string"
14                            requirementType="xs:QName"
15                            lowerBound="xs:integer"?
16                            upperBound="xs:integer | xs:string"?>
17       <Constraints>
18         <Constraint constraintType="xs:anyURI">
19           constraint type specific content
20         </Constraint> +
21       </Constraints> ?
22     </RequirementDefinition> +
23   </RequirementDefinitions> ?
24
25   <CapabilityDefinitions>
26     <CapabilityDefinition name="xs:string"
27                           capabilityType="xs:QName"
28                           lowerBound="xs:integer"?
29                           upperBound="xs:integer | xs:string"?>
30       <Constraints>
31         <Constraint constraintType="xs:anyURI">
32           constraint type specific content
33         </Constraint> +
34       </Constraints> ?
```

```
    </CapabilityDefinition> +
  </CapabilityDefinitions>

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

  <Interfaces>
    <Interface name="xs:NCName | xs:anyURI">
      <Operation name="xs:NCName">
        <InputParameters>
          <InputParameter name="xs:string" type="xs:string"
                          required="yes|no"?/> +
        </InputParameters> ?
        <OutputParameters>
          <OutputParameter name="xs:string" type="xs:string"
                           required="yes|no"?/> +
        </OutputParameters> ?
      </Operation> +
    </Interface> +
  </Interfaces> ?

</NodeType>
```

6.2 Properties

The `NodeType` element has the following properties:

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

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

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

  Note: a final Node Type MUST NOT be declared as abstract.
- `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by the author to describe the Node Type. Each tag is defined by a separate, nested `Tag` element. The `Tag` element has the following properties:
  - `name`: This attribute specifies the name of the tag.
  - `value`: This attribute specifies the value of the tag.

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

- `DerivedFrom`: This is an OPTIONAL reference to another Node Type from which this Node Type derives. Conflicting definitions are resolved by the rule that local new definitions always override derived definitions. See section 6.3 Derivation Rules for details.
  The `DerivedFrom` element has the following properties:
  - `typeRef`: The QName specifies the Node Type from which this Node Type derives its definitions.
- `PropertiesDefinition`: This element specifies the structure of the observable properties of the Node Type, such as its configuration and state, by means of XML schema.
  The `PropertiesDefinition` element has one but not both of the following properties:
  - `element`: This attribute provides the QName of an XML element defining the structure of the Node Type Properties.
  - `type`: This attribute provides the QName of an XML (complex) type defining the structure of the Node Type Properties.
- `RequirementDefinitions`: This OPTIONAL element specifies the requirements that the Node Type exposes (see section 3.4 for details). Each requirement is defined in a nested `RequirementDefinition` element.
  The `RequirementDefinition` element has the following properties:
  - `name`: This attribute specifies the name of the defined requirement and MUST be unique within the `RequirementsDefinitions` of the current Node Type.

    Note that one Node Type might define multiple requirements of the same Requirement Type, in which case each occurrence of a requirement definition is uniquely identified by its name. For example, a Node Type for an application might define two requirements for a database (i.e. of the same Requirement Type) where one could be named "customerDatabase" and the other one could be named "productsDatabase".
  - `requirementType`: This attribute identifies by QName the Requirement Type that is being defined by the current `RequirementDefinition`.
  - `lowerBound`: This OPTIONAL attribute specifies the lower boundary by which a requirement MUST be matched for Node Templates according to the current Node Type, or for instances created for those Node Templates. The default value for this attribute is one. A value of zero would indicate that matching of the requirement is optional.
  - `upperBound`: This OPTIONAL attribute specifies the upper boundary by which a requirement MUST be matched for Node Templates according to the current Node Type, or for instances created for those Node Templates. The default value for this attribute is one. A value of "`unbounded`" indicates that there is no upper boundary.
    `Constraints`: This OPTIONAL element contains a list of `Constraint` elements that specify additional constraints on the requirement definition. For example, if a database is needed a constraint on supported SQL features might be expressed.
    The nested `Constraint` element has the following properties:
    - `constraintType`: This attribute specifies the type of constraint. According to this type, the body of the `Constraint` element will contain type specific content.
- `CapabilityDefinitions`: This OPTIONAL element specifies the capabilities that the Node Type exposes (see section 3.4 for details). Each capability is defined in a nested `CapabilityDefinition` element.
  The `CapabilityDefinition` element has the following properties:
  - `name`: This attribute specifies the name of the defined capability and MUST be unique within the `CapabilityDefinitions` of the current Node Type.

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

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

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

    The `Operation` element has the following properties:
    - `name`: This attribute defines the name of the operation and MUST be unique within the containing `Interface` of the Node Type.
    - `InputParameters`: This OPTIONAL property contains a list of one or more input parameter definitions, each defined in a nested, separate `InputParameter` element.
      The `InputParameter` element has the following properties:
      - `name`: This attribute specifies the name of the input parameter, which MUST be unique within the set of input parameters defined for the operation.
      - `type`: This attribute specifies the type of the input parameter.
      - `required`: This OPTIONAL attribute specifies whether or not the input parameter is REQUIRED (`required` attribute with a value of "yes" – default) or OPTIONAL (`required` attribute with a value of "no").
    - `OutputParameters`: This OPTIONAL property contains a list of one or more output parameter definitions, each defined in a nested, separate `OutputParameter` element.
      The `OutputParameter` element has the following properties:

- `name`: This attribute specifies the name of the output parameter, which MUST be unique within the set of output parameters defined for the operation.
- `type`: This attribute specifies the type of the output parameter.
- `required`: This OPTIONAL attribute specifies whether or not the output parameter is REQUIRED (`required` attribute with a value of "yes" – default) or OPTIONAL (`required` attribute with a value of "no").

6.3 Derivation Rules

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

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

  In cases where the Node Type under definition defines a requirement or capability with a certain name where the Node Type derived from already contains a respective definition with the same name, the definition in the Node Type under definition overrides the definition of the Node Type derived from. In such a case, the requirement definition or capability definition, respectively, MUST reference a Requirement Type or Capability Type that is derived from the one in the corresponding requirement definition or capability definition of the Node Type derived from.
- Instance States: The set of instance states of the Node Type under definition consists of the set union of the instances states defined by the Nodes Type derived from and the instance states defined by the Node Type under definition. A set of instance states of the same name will be combined into a single instance state of the same name.
- Interfaces: The set of interfaces of the Node Type under definition consists of the set union of interfaces defined by the Node Type derived from and the interfaces defined by the Node Type under definition.
  Two interfaces of the same name will be combined into a single, derived interface with the same name. The set of operations of the derived interface consists of the set union of operations defined by both interfaces. An operation defined by the Node Type under definition substitutes an operation with the same name of the Node Type derived from.

6.4 Example

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

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



```
09      </documentation>
10
11      <PropertiesDefinition element="ProjectProperties"/>
12
13      <InstanceStates>
14        <InstanceState state="www.example.com/active"/>
15        <InstanceState state="www.example.com/onHold"/>
16      </InstanceStates>
17
18      <Interfaces>
19        <Interface name="ProjectInterface">
20          <Operation name="CreateProject">
21            <InputParameters>
22              <InputParamter name="ProjectName"
23                             type="xs:string"/>
24              <InputParamter name="Owner"
25                             type="xs:string"/>
26              <InputParamter name="AccountID"
27                             type="xs:string"/>
28            </InputParameters>
29          </Operation>
30        </Interface>
31      </Interfaces>
32    </NodeType>
33
34  </Definitions>
```

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

7 Node Type Implementations

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

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

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

7.1 XML Syntax

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

```
<NodeTypeImplementation name="xs:NCName" targetNamespace="xs:anyURI"?
                        nodeType="xs:QName"
                        abstract="yes|no"?
                        final="yes|no"?>

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

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

  <RequiredContainerFeatures>
    <RequiredContainerFeature feature="xs:anyURI"/> +
  </RequiredContainerFeatures> ?

  <ImplementationArtifacts>
    <ImplementationArtifact interfaceName="xs:NCName | xs:anyURI"?
                            operationName="xs:NCName"?
                            artifactType="xs:QName"
                            artifactRef="xs:QName"?>
        artifact specific content ?
    <ImplementationArtifact> +
  </ImplementationArtifacts> ?

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

</NodeTypeImplementation>
```

7.2 Properties

The `NodeTypeImplementation` element has the following properties:

- `name`: This attribute specifies the name or identifier of the Node Type Implementation, which MUST be unique within the target namespace.
- `targetNamespace`: This OPTIONAL attribute specifies the target namespace to which the definition of the Node Type Implementation will be added. If not specified, the Node Type Implementation will be added to the target namespace of the enclosing Definitions document.
- `nodeType`: The QName value of this attribute specifies the Node Type implemented by this Node Type Implementation.
- `abstract`: This OPTIONAL attribute specifies that this Node Type Implementation cannot be used directly as an implementation for the Node Type specified in the `nodeType` attribute.

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

  Note: an abstract Node Type Implementation MUST NOT be declared as final.
- `final`: This OPTIONAL attribute specifies that other Node Type Implementations MUST NOT be derived from this Node Type Implementation.

  Note: a final Node Type Implementation MUST NOT be declared as abstract.
- `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by the author to describe the Node Type Implementation. Each tag is defined by a separate, nested `Tag` element.
  The `Tag` element has the following properties:
  - `name`: This attribute specifies the name of the tag.
  - `value`: This attribute specifies the value of the tag.

    Note: The name/value pairs defined in tags have no normative interpretation.
- `DerivedFrom`: This is an OPTIONAL reference to another Node Type Implementation from which this Node Type Implementation derives. See section 7.3 Derivation Rules**Error! Reference source not found.** for details.
  The `DerivedFrom` element has the following properties:
  - `nodeTypeImplementationRef`: The QName specifies the Node Type Implementation from which this Node Type Implementation derives.
- `RequiredContainerFeatures`: An implementation of a Node Type might depend on certain features of the environment it is executed in, such as specific (potentially proprietary) APIs of the TOSCA container. For example, an implementation to deploy a virtual machine based on an image could require access to some API provided by a public cloud, while another implementation could require an API of a vendor-specific virtual image library. Thus, the contents of the `RequiredContainerFeatures` element provide “hints” to the TOSCA container allowing it to select the appropriate Node Type Implementation if multiple alternatives are provided.
  Each such dependency is defined by a separate `RequiredContainerFeature` element.
  The `RequiredContainerFeature` element has the following properties:
  - `feature`: The value of this attribute is a URI that denotes the corresponding needed feature of the environment.

- `ImplementationArtifacts`: This element specifies a set of implementation artifacts for interfaces or operations of a Node Type.
  The `ImplementationArtifacts` element has the following properties:
  - `ImplementationArtifact`: This element specifies one implementation artifact of an interface or an operation.

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

    The `ImplementationArtifact` element has the following properties:
    - `name`: This attribute specifies the name of the artifact, which SHOULD be unique within the scope of the encompassing Node Type Implementation.
    - `interfaceName`: This OPTIONAL attribute specifies the name of the interface that is implemented by the actual implementation artifact. If not specified, the implementation artifact is assumed to provide the implementation for all interfaces defined by the Node Type referred to by the `nodeType` attribute of the containing `NodeTypeImplementation`.
    - `operationName`: This OPTIONAL attribute specifies the name of the operation that is implemented by the actual implementation artifact. If specified, the `interfaceName` MUST be specified and the specified `operationName` MUST refer to an operation of the specified interface. If not specified, the implementation artifact is assumed to provide the implementation for all operations defined within the specified interface.
    - `artifactType`: This attribute specifies the type of this artifact. The QName value of this attribute SHOULD correspond to the QName of an `ArtifactType` defined in the same Definitions document or in an imported document.

      The `artifactType` attribute specifies the artifact type specific content of the `ImplementationArtifact` element body and indicates the type of Artifact Template referenced by the Implementation Artifact via the `artifactRef` attribute.
    - `artifactRef`: This OPTIONAL attribute contains a QName that identifies an Artifact Template to be used as implementation artifact. This Artifact Template can be defined in the same Definitions document or in a separate, imported document.
      The type of Artifact Template referenced by the `artifactRef` attribute MUST be the same type or a sub-type of the type specified in the `artifactType` attribute.

      Note: if no Artifact Template is referenced, the artifact type specific content of the `ImplementationArtifact` element alone is assumed to represent the actual artifact. For example, a simple script could be defined in place within the `ImplementationArtifact` element.
- `DeploymentArtifacts`: This element specifies a set of deployment artifacts relevant for materializing instances of nodes of the Node Type being implemented.
  The `DeploymentArtifacts` element has the following properties:
  - `DeploymentArtifact`: This element specifies one deployment artifact.

Note: Multiple deployment artifacts MAY be defined in a Node Type Implementation. One reason could be that multiple artifacts (maybe of different types) are needed to materialize a node as a whole. Another reason could be that alternative artifacts are provided for use in different contexts (e.g. different installables of a software for use in different operating systems).

The `DeploymentArtifact` element has the following properties:

- `name`: This attribute specifies the name of the artifact, which SHOULD be unique within the scope of the encompassing Node Type Implementation.
- `artifactType`: This attribute specifies the type of this artifact. The QName value of this attribute SHOULD correspond to the QName of an `ArtifactType` defined in the same Definitions document or in an imported document.

  The `artifactType` attribute specifies the artifact type specific content of the `DeploymentArtifact` element body and indicates the type of Artifact Template referenced by the Deployment Artifact via the `artifactRef` attribute.
- `artifactRef`: This OPTIONAL attribute contains a QName that identifies an Artifact Template to be used as deployment artifact. This Artifact Template can be defined in the same Definitions document or in a separate, imported document.
  The type of Artifact Template referenced by the `artifactRef` attribute MUST be the same type or a sub-type of the type specified in the `artifactType` attribute.

  Note: if no Artifact Template is referenced, the artifact type specific content of the `DeploymentArtifact` element alone is assumed to represent the actual artifact. For example, the contents of a simple config file could be defined in place within the `DeploymentArtifact` element.

7.3 Derivation Rules

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

- Implementation Artifacts: The set of implementation artifacts of a Node Type Implementation consists of the set union of implementation artifacts defined by the Node Type Implementation itself and the implementation artifacts defined by any Node Type Implemenation the Node Type Implementation is derived from.
  An implementation artifact defined by a Node Type Implementation overrides an implementation artifact having the same interface name and operation name of a Node Type Implementation the Node Type Implementation is derived from.
  If an implementation artifact defined in a Node Type Implementation specifies only an interface name, it substitutes implementation artifacts having the same interface name (with or without an operation name defined) of any Node Type Implementation the Node Type Implementation is derived from. In this case, the implementation of a complete interface of a Node Type is overridden.
  If an implementation artifact defined in a Node Type Implementation neither defines an interface name nor an operation name, it overrides all implementation artifacts of any Node Type Implementation the Node Type Implementation is derived from. In this case, the complete implementation of a Node Type is overridden.

- Deployment Artifacts: The set of deployment artifacts of a Node Type Implementation consists of the set union of the deployment artifacts defined by the Nodes Type Implementation itself and the deployment artifacts defined by any Node Type Implementation the Node Type Implementation is derived from. A deployment artifact defined by a Node Type Implementation overrides a deployment artifact with the same name and type (or any type it is derived from) of any Node Type Implementation the Node Type Implementation is derived from.

7.4 Example

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

```
01 <Definitions id="MyImpls" name="My Implementations"
02   targetNamespace="http://www.example.com/SampleImplementations"
03   xmlns:bn="http://www.example.com/BaseNodeTypes"
04   xmlns:ba="http://www.example.com/BaseArtifactTypes"
05   xmlns:sa="http://www.example.com/SampleArtifacts">
06
07   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
08           namespace="http://www.example.com/BaseArtifactTypes"/>
09
10   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
11           namespace="http://www.example.com/BaseNodeTypes"/>
12
13   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
14           namespace="http://www.example.com/SampleArtifacts"/>
15
16   <NodeTypeImplementation name="MyDBMSImplementation"
17                           nodeType="bn:DBMS">
18
19     <ImplementationArtifacts>
20       <ImplementationArtifact interfaceName="MgmtInterface"
21                               artifactType="ba:WARFile"
22                               artifactRef="sa:MyMgmtWebApp">
23       </ImplementationArtifact>
24     </ImplementationArtifacts>
25
26     <DeploymentArtifacts>
27       <DeploymentArtifact name="MyDBMS"
28                           artifactType="ba:ZipFile"
29                           artifactRef="sa:MyInstallable">
30       </DeploymentArtifact>
31     </DeploymentArtifacts>
32
33   </NodeTypeImplementation>
34
35 </Definitions>
```

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

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

8 Relationship Types

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

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

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

8.1 XML Syntax

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

```
01 <RelationshipType name="xs:NCName"
02                   targetNamespace="xs:anyURI"?
03                   abstract="yes|no"?
04                   final="yes|no"?> +
05
06   <Tags>
07     <Tag name="xs:string" value="xs:string"/> +
08   </Tags> ?
09
10   <DerivedFrom typeRef="xs:QName"/> ?
11
12   <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
13
14   <InstanceStates>
15     <InstanceState state="xs:anyURI"> +
16   </InstanceStates> ?
17
18   <SourceInterfaces>
19     <Interface name="xs:NCName | xs:anyURI">
20       ...
21     </Interface> +
22   </SourceInterfaces> ?
23
24   <TargetInterfaces>
25     <Interface name="xs:NCName | xs:anyURI">
26       ...
27     </Interface> +
28   </TargetInterfaces> ?
29
30   <ValidSource typeRef="xs:QName"/> ?
31
32   <ValidTarget typeRef="xs:QName"/> ?
33
34 </RelationshipType>
```

8.2 Properties

The `RelationshipType` element has the following properties:

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

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

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

  Note: a final Relationship Type MUST NOT be declared as abstract.
- `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by the author to describe the Relationship Type. Each tag is defined by a separate, nested `Tag` element.
  The `Tag` element has the following properties:
  - `name`: This attribute specifies the name of the tag.
  - `value`: This attribute specifies the value of the tag.

    Note: The name/value pairs defined in tags have no normative interpretation.
- `DerivedFrom`: This is an OPTIONAL reference to another Relationship Type from which this Relationship Type is derived. Conflicting definitions are resolved by the rule that local new definitions always override derived definitions. See section 8.3 Derivation Rules for details.
  The `DerivedFrom` element has the following properties:
  - `typeRef`: The QName specifies the Relationship Type from which this Relationship Type derives its definitions.
- `PropertiesDefinition`: This element specifies the structure of the observable properties of the Relationship Type, such as its configuration and state, by means of XML schema.
  The `PropertiesDefinition` element has one but not both of the following properties:
  - `element`: This attribute provides the QName of an XML element defining the structure of the Relationship Type Properties.
  - `type`: This attribute provides the QName of an XML (complex) type defining the structure of the Relationship Type Properties.
- `InstanceStates`: This OPTIONAL element lists the set of states an instance of this Relationship Type can occupy at runtime. Those states are defined in nested `InstanceState` elements.
  The `InstanceState` element has the following nested properties:
  - `state`: This attribute specifies a URI that identifies a potential state.
- `SourceInterfaces`: This OPTIONAL element contains definitions of manageability interfaces that can be performed on the source of a relationship of this Relationship Type to actually establish the relationship between the source and the target in the deployed service.

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

- `TargetInterfaces`: This OPTIONAL element contains definitions of manageability interfaces that can be performed on the target of a relationship of this Relationship Type to actually establish the relationship between the source and the target in the deployed service.
  Those interface definitions are contained in nested `Interface` elements, the content of which is that described for Node Type interfaces (see section 6.2).
- `ValidSource`: This OPTIONAL element specifies the type of object that is allowed as a valid origin for relationships defined using the Relationship Type under definition. If not specified, any Node Type is allowed to be the origin of the relationship.
  The `ValidSource` element has the following properties:
  - `typeRef`: This attribute specifies the QName of a Node Type or Requirement Type that is allowed as a valid source for relationships defined using the Relationship Type under definition. Node Types or Requirements Types derived from the specified Node Type or Requirement Type, respectively, MUST also be accepted as valid relationship source.

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

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

8.3 Derivation Rules

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

- Relationship Type Properties: It is assumed that the XML element (or type) representing the Relationship Type properties of the Relationship Type under definition extends the XML element (or type) of the Relationship Type properties of the Relationship Type referenced in the `DerivedFrom` element.
- Instance States: The resulting set of instance states of the Relationship Type under definition consists of the set union of the instances states defined by the Relationship Type derived from

and the instance states explicitly defined by the Relationship Type under definition. Instance states with the same state attribute will be combined into a single instance state of the same state.

- Valid source and target: An object specified as a valid source or target, respectively, of the Relationship Type under definition MUST be of a subtype defined as valid source or target, respectively, of the Relationship Type derived from.

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

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

- Interfaces: The set of interfaces (both source and target interfaces) of the Relationship Type under definition consists of the set union of interfaces defined by the Relationship Type derived from and the interfaces defined by the Relationship Type under definition.
  Two interfaces of the same name will be combined into a single, derived interface with the same name. The set of operations of the derived interface consists of the set union of operations defined by both interfaces. An operation defined by the Relationship Type under definition substitutes an operation with the same name of the Relationship Type derived from.

8.4 Example

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

```
01 <RelationshipType name="processDeployedOn">
02
03   <RelationshipTypeProperties element="ProcessDeployedOnProperties"/>
04
05   <InstanceStates>
06     <InstanceState state="www.example.com/successfullyDeployed"/>
07     <InstanceState state="www.example.com/failed"/>
08   </InstanceStates>
09
10 </RelationshipType>
```

9 Relationship Type Implementations

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

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

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

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

9.1 XML Syntax

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

```
<RelationshipTypeImplementation name="xs:NCName"
                                targetNamespace="xs:anyURI"?
                                relationshipType="xs:QName"
                                abstract="yes|no"?
                                final="yes|no"?>

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

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

  <RequiredContainerFeatures>
    <RequiredContainerFeature feature="xs:anyURI"/> +
  </RequiredContainerFeatures> ?

  <ImplementationArtifacts>
    <ImplementationArtifact interfaceName="xs:NCName | xs:anyURI"?
                            operationName="xs:NCName"?
                            artifactType="xs:QName"
                            artifactRef="xs:QName"?>
        artifact specific content ?
    <ImplementationArtifact> +
  </ImplementationArtifacts> ?

</RelationshipTypeImplementation>
```

9.2 Properties

The `RelationshipTypeImplementation` element has the following properties:

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

- `targetNamespace`: This OPTIONAL attribute specifies the target namespace to which the definition of the Relationship Type Implementation will be added. If not specified, the Relationship Type Implementation will be added to the target namespace of the enclosing Definitions document.
- `relationshipType`: The QName value of this attribute specifies the Relationship Type implemented by this Relationship Type Implementation.
- `abstract`: This OPTIONAL attribute specifies that this Relationship Type Implementation cannot be used directly as an implementation for the Relationship Type specified in the `relationshipType` attribute.

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

  Note: an abstract Relationship Type Implementation MUST NOT be declared as final.
- `final`: This OPTIONAL attribute specifies that other Relationship Type Implementations MUST NOT be derived from this Relationship Type Implementation.

  Note: a final Relationship Type Implementation MUST NOT be declared as abstract.
- `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by the author to describe the Relationship Type Implementation. Each tag is defined by a separate, nested `Tag` element.
  The `Tag` element has the following properties:
  - `name`: This attribute specifies the name of the tag.
  - `value`: This attribute specifies the value of the tag.

    Note: The name/value pairs defined in tags have no normative interpretation.
- `DerivedFrom`: This is an OPTIONAL reference to another Relationship Type Implementation from which this Relationship Type Implementation derives. See section 9.3 Derivation Rules or details.
  The `DerivedFrom` element has the following properties:
  - `relationshipTypeImplementationRef`: The QName specifies the Relationship Type Implementation from which this Relationship Type Implementation derives.
- `RequiredContainerFeatures`: An implementation of a Relationship Type might depend on certain features of the environment it is executed in, such as specific (potentially proprietary) APIs of the TOSCA container.
  Thus, the contents of the `RequiredContainerFeatures` element provide “hints” to the TOSCA container allowing it to select the appropriate Relationship Type Implementation if multiple alternatives are provided.
  Each such dependency is defined by a separate `RequiredContainerFeature` element.
  The `RequiredContainerFeature` element has the following properties:
  - `feature`: The value of this attribute is a URI that denotes the corresponding needed feature of the environment.
- `ImplementationArtifacts`: This element specifies a set of implementation artifacts for interfaces or operations of a Relationship Type.
  The `ImplementationArtifacts` element has the following properties:
  - `ImplementationArtifact`: This element specifies one implementation artifact of an interface or an operation.

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

The `ImplementationArtifact` element has the following properties:

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

  Note that the referenced interface can be defined in either the `SourceInterfaces` element or the `TargetInterfaces` element of the Relationship Type implemented by this Relationship Type Implementation.
- `operationName`: This OPTIONAL attribute specifies the name of the operation that is implemented by the actual implementation artifact. If specified, the `interfaceName` MUST be specified and the specified `operationName` MUST refer to an operation of the specified interface. If not specified, the implementation artifact is assumed to provide the implementation for all operations defined within the specified interface.
- `artifactType`: This attribute specifies the type of this artifact. The QName value of this attribute SHOULD correspond to the QName of an `ArtifactType` defined in the same Definitions document or in an imported document.

  The `artifactType` attribute specifies the artifact type specific content of the `ImplementationArtifact` element body and indicates the type of Artifact Template referenced by the Implementation Artifact via the `artifactRef` attribute.
- `artifactRef`: This OPTIONAL attribute contains a QName that identifies an Artifact Template to be used as implementation artifact. This Artifact Template can be defined in the same Definitions document or in a separate, imported document.
  The type of Artifact Template referenced by the `artifactRef` attribute MUST be the same type or a sub-type of the type specified in the `artifactType` attribute.

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

9.3 Derivation Rules

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

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

Type Implementation itself and the implementation artifacts defined by any Relationship Type Implemenation the Relationship Type Implementation is derived from.

An implementation artifact defined by a Node Type Implementation overrides an implementation artifact having the same interface name and operation name of a Relationship Type Implementation the Relationship Type Implementation is derived from.

If an implementation artifact defined in a Relationship Type Implementation specifies only an interface name, it substitutes implementation artifacts having the same interface name (with or without an operation name defined) of any Relationship Type Implementation the Relationship Type Implementation is derived from. In this case, the implementation of a complete interface of a Relationship Type is overridden.

If an implementation artifact defined in a Relationship Type Implementation neither defines an interface name nor an operation name, it overrides all implementation artifacts of any Relationship Type Implementation the Relationship Type Implementation is derived from. In this case, the complete implementation of a Relationship Type is overridden.

9.4 Example

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

```
<Definitions id="MyImpls" name="My Implementations"
  targetNamespace="http://www.example.com/SampleImplementations"
  xmlns:bn="http://www.example.com/BaseRelationshipTypes"
  xmlns:ba="http://www.example.com/BaseArtifactTypes"
  xmlns:sa="http://www.example.com/SampleArtifacts">

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

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

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

  <RelationshipTypeImplementation name="MyDBConnectImplementation"
                                  relationshipType="bn:DBConnection">

     <ImplementationArtifacts>
       <ImplementationArtifact interfaceName="ConnectionInterface"
                               operationName="connectTo"
                               artifactType="ba:ScriptArtifact"
                               artifactRef="sa:MyConnectScript">
       <ImplementationArtifact>
     </ImplementationArtifacts>

  </RelationshipTypeImplementation>

</Definitions>
```

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

10 Requirement Types

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

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

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

10.1 XML Syntax

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

```
01 <RequirementType name="xs:NCName"
02                  targetNamespace="xs:anyURI"?
03                  abstract="yes|no"?
04                  final="yes|no"?
05                  requiredCapabilityType="xs:QName"?>
06
07   <Tags>
08     <Tag name="xs:string" value="xs:string"/> +
09   </Tags> ?
10
11   <DerivedFrom typeRef="xs:QName"/> ?
12
13   <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
14
15 </RequirementType>
```

10.2 Properties

The `RequirementType` element has the following properties:

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

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

be instantiated during the creation of a service according to a Service Template.

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

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

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

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

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

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

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

- `DerivedFrom`: This is an OPTIONAL reference to another Requirement Type from which this Requirement Type derives. See section 10.3 Derivation Rules for details.
  The `DerivedFrom` element has the following properties:
  - `typeRef`: The QName specifies the Requirement Type from which this Requirement Type derives its definitions and semantics.

- `PropertiesDefinition`: This element specifies the structure of the observable properties of the Requirement Type, such as its configuration and state, by means of XML schema.
  The `PropertiesDefinition` element has one but not both of the following properties:
  - `element`: This attribute provides the QName of an XML element defining the structure of the Requirement Type Properties.
  - `type`: This attribute provides the QName of an XML (complex) type defining the structure of the Requirement Type Properties.

10.3 Derivation Rules

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

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

10.4 Example

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

```
01 <Definitions id="MyRequirements" name="My Requirements"
02   targetNamespace="http://www.example.com/SampleRequirements"
03   xmlns:br="http://www.example.com/BaseRequirementTypes"
04   xmlns:mrp="http://www.example.com/SampleRequirementProperties>
05
06   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
07     namespace="http://www.example.com/BaseRequirementTypes"/>
08
09   <Import importType="http://www.w3.org/2001/XMLSchema"
10     namespace="http://www.example.com/SampleRequirementProperties"/>
11
12   <RequirementType name="DatabaseClientEndpoint">
13     <DerivedFrom typeRef="br:ClientEndpoint"/>
14     <PropertiesDefinition
15       element="mrp:DatabaseClientEndpointProperties"/>
16   </RequirementType>
17
18 </Definitions>
```

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

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

11 Capability Types

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

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

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

11.1 XML Syntax

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

```
01 <CapabilityType name="xs:NCName"
02                 targetNamespace="xs:anyURI"?
03                 abstract="yes|no"?
04                 final="yes|no"?>
05
06   <Tags>
07     <Tag name="xs:string" value="xs:string"/> +
08   </Tags> ?
09
10   <DerivedFrom typeRef="xs:QName"/> ?
11
12   <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
13
14 </CapabilityType>
```

11.2 Properties

The `CapabilityType` element has the following properties:

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

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

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

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

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

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

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

- `DerivedFrom`: This is an OPTIONAL reference to another Capability Type from which this Capability Type derives. See section 11.3 Derivation Rules for details.
  The `DerivedFrom` element has the following properties:
  - `typeRef`: The QName specifies the Capability Type from which this Capability Type derives its definitions and semantics.

- `PropertiesDefinition`: This element specifies the structure of the observable properties of the Capability Type, such as its configuration and state, by means of XML schema.
  The `PropertiesDefinition` element has one but not both of the following properties:
  - `element`: This attribute provides the QName of an XML element defining the structure of the Capability Type Properties.
  - `type`: This attribute provides the QName of an XML (complex) type defining the structure of the Capability Type Properties.

11.3 Derivation Rules

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

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

11.4 Example

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

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



```
11
12   <CapabilityType name="DatabaseServerEndpoint">
13     <DerivedFrom typeRef="bc:ServerEndpoint"/>
14     <PropertiesDefinition
15       element="mcp:DatabaseServerEndpointProperties"/>
16   </CapabilityType>
17
18 </Definitions>
```

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

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

12 Artifact Types

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

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

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

12.1 XML Syntax

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

```
01 <ArtifactType name="xs:NCName"
02                targetNamespace="xs:anyURI"?
03                abstract="yes|no"?
04                final="yes|no"?>
05
06   <Tags>
07     <Tag name="xs:string" value="xs:string"/> +
08   </Tags> ?
09
10   <DerivedFrom typeRef="xs:QName"/> ?
11
12   <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
13
14 </ArtifactType>
```

12.2 Properties

The `ArtifactType` element has the following properties:

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

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

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

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

  Note: a final Artifact Type MUST NOT be declared as abstract.
- `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by the author to describe the Artifact Type. Each tag is defined by a separate, nested `Tag` element. The `Tag` element has the following properties:
  - `name`: This attribute specifies the name of the tag.
  - `value`: This attribute specifies the value of the tag.

    Note: The name/value pairs defined in tags have no normative interpretation.
- `DerivedFrom`: This is an OPTIONAL reference to another Artifact Type from which this Artifact Type derives. See section 12.3 Derivation Rules for details.
  The `DerivedFrom` element has the following properties:
  - `typeRef`: The QName specifies the Artifact Type from which this Artifact Type derives its definitions and semantics.
- `PropertiesDefinition`: This element specifies the structure of the observable properties of the Artifact Type, such as its configuration and state, by means of XML schema.
  The `PropertiesDefinition` element has one but not both of the following properties:
  - `element`: This attribute provides the QName of an XML element defining the structure of the Artifact Type Properties.
  - `type`: This attribute provides the QName of an XML (complex) type defining the structure of the Artifact Type Properties.

12.3 Derivation Rules

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

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

12.4 Example

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

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

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

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

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

13 Artifact Templates

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

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

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

13.1 XML Syntax

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

```
01 <ArtifactTemplate id="xs:ID" name="xs:string"? type="xs:QName">
02
03   <Properties>
04     XML fragment
05   </Properties> ?
06
07   <PropertyConstraints>
08     <PropertyConstraint property="xs:string"
09                         constraintType="xs:anyURI"> +
10       constraint ?
11     </PropertyConstraint>
12   </PropertyConstraints> ?
13
14   <ArifactReferences>
15     <ArtifactReference reference="xs:anyURI">
16       (
17        <Include pattern="xs:string"/>
18       |
19        <Exclude pattern="xs:string"/>
20       )*
21     </ArtifactReference> +
22   </ArtifactReferences> ?
23
24 </ArtifactTemplate>
```

13.2 Properties

The `ArtifactTemplate` element has the following properties:

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

- `type`: The QName value of this attribute refers to the Artifact Type providing the type of the Artifact Template.

  Note: If the Artifact Type referenced by the `type` attribute of an Artifact Template is declared as abstract, no instances of the specific Artifact Template can be created, i.e. the artifact cannot be used directly as deployment or implementation artifact. Instead, a substitution of the Artifact Template with one having a specialized, derived Artifact Type has to be done at the latest during the instantiation time of a Service Template.

- `Properties`: This OPTIONAL element specifies the invariant properties of the Artifact Template, i.e. those properties that will be commonly used across different contexts in which the Artifact Template is used.

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

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

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

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

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

- `ArtifactReferences`: This OPTIONAL element contains one or more references to the actual artifact proper, each represented by a separate `ArtifactReference` element.
  The `ArtifactReference` element has the following properties:

  - `reference`: This attribute contains a URI pointing to an actual artifact. If this URI is a relative URI, it is interpreted relative to the root directory of the CSAR containing the Service Template (see also sections 3.7 and 14).

  - `Include`: This OPTIONAL element can be used to define a pattern of files that are to be included in the artifact reference in case the reference points to a complete directory.
    The `Include` element has the following properties:

    - `pattern`: This attribute contains a pattern definition for files that are to be included in the overall artifact reference. For example, a pattern of "`*.py`" would include all python scripts contained in a directory.

  - `Exclude`: This OPTIONAL element can be used to define a pattern of files that are to be excluded from the artifact reference in case the reference points to a complete directory.
    The `Exclude` element has the following properties:

- `pattern`: This attribute contains a pattern definition for files that are to be excluded in the overall artifact reference. For example, a pattern of "`*.sh`" would exclude all bash scripts contained in a directory.

13.3 Example

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

```
01 <Definitions id="MyArtifacts" name="My Artifacts"
02   targetNamespace="http://www.example.com/SampleArtifacts"
03   xmlns:ba="http://www.example.com/BaseArtifactTypes">
04
05   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
06           namespace="http://www.example.com/BaseArtifactTypes"/>
07
08   <ArtifactTemplate id="MyInstallable"
09                     name="My installable"
10                     type="ba:ZipFile">
11     <ArtifactReferences>
12       <ArtifactReference reference="files/MyInstallable.zip"/>
13     </ArtifactReferences>
14   </ArtifactTemplate>
15
16 </Definitions>
```

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

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

14 Policy Types

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

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

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

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

14.1 XML Syntax

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

```
01 <PolicyType name="xs:NCName"
02             policyLanguage="xs:anyURI"?
03             abstract="yes|no"?
04             final="yes|no"?
05             targetNamespace="xs:anyURI"?>
06   <Tags>
07     <Tag name="xs:string" value="xs:string"/> +
08   </Tags> ?
09
10   <DerivedFrom typeRef="xs:QName"/> ?
11
12   <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
13
14   <AppliesTo>
15     <NodeTypeReference typeRef="xs:QName"/> +
16   </AppliesTo> ?
17
18   policy type specific content ?
19
20 </PolicyType>
```

14.2 Properties

The `PolicyType` element has the following properties:

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

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

  As a consequence, a Policy Template of an abstract Policy Type MUST be replaced by a policy of a derived Policy Type at the latest during deployment of the element that policy is attached to.
- `final`: This OPTIONAL attribute specifies that other Policy Types MUST NOT be derived from this Policy Type.

  Note: a final Policy Type MUST NOT be declared as abstract.
- `Tags`: This OPTIONAL element allows the definition of any number of tags which can be used by the author to describe the Policy Type. Each tag is defined by a separate, nested `Tag` element. The `Tag` element has the following properties:
  - `name`: This attribute specifies the name of the tag.
  - `value`: This attribute specifies the value of the tag.

    Note: The name/value pairs defined in tags have no normative interpretation.
- `DerivedFrom`: This is an OPTIONAL reference to another Policy Type from which this Policy Type derives. See section 14.3 Derivation Rules for details.
  The `DerivedFrom` element has the following properties:
  - `typeRef`: The QName specifies the Policy Type from which this Policy Type derives its definitions from.
- `PropertiesDefinition`: This element specifies the structure of the observable properties of the Policy Type by means of XML schema.
  The `PropertiesDefinition` element has one but not both of the following properties:
  - `element`: This attribute provides the QName of an XML element defining the structure of the Policy Type Properties.
  - `type`: This attribute provides the QName of an XML (complex) type defining the structure of the Policy Type Properties.
- `AppliesTo`: This OPTIONAL element specifies the set of Node Types the Policy Type is applicable to, each defined as a separate, nested `NodeTypeReference` element.
  The `NodeTypeReference` element has the following property:
  - `typeRef`: The attribute provides the QName of a Node Type to which the Policy Type applies.

14.3 Derivation Rules

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

- Properties Definitions: It is assumed that the XML element (or type) representing the Policy Type Properties Definitions extends the XML element (or type) of the Policy Type Properties Definitions of the Policy Type referenced in the `DerivedFrom` element.
- Applies To: The set of Node Types the Policy Type is applicable to consist of the set union of Node Types derived from and Node Types explicitly referenced by the Policy Type by means of its `AppliesTo` element.
- Policy Language: A Policy Type MUST define the same policy language as the Policy Type it derives from. In case the Policy Type used as basis for derivation has no `policyLanguage` attribute defined, the deriving Policy Type can define any appropriate policy language.

14.4 Example

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

```
01 <Definitions id="MyPolicyTypes" name="My Policy Types"
02   targetNamespace="http://www.example.com/SamplePolicyTypes"
03   xmlns:bnt="http://www.example.com/BaseNodeTypes">
04   xmlns:spp="http://www.example.com/SamplePolicyProperties">
05
06   <Import importType="http://www.w3.org/2001/XMLSchema"
07           namespace="http://www.example.com/SamplePolicyProperties"/>
08
09   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
10           namespace="http://www.example.com/BaseNodeTypes"/>
11
12
13   <PolicyType name="HighAvailability">
14     <PropertiesDefinition element="spp:HAProperties"/>
15   </PolicyType>
16
17   <PolicyType name="ContinuousAvailability">
18     <DerivedFrom typeRef="HighAvailability"/>
19     <PropertiesDefinition element="spp:CAProperties"/>
20     <AppliesTo>
21       <NodeTypeReference typeRef="bnt:DBMS"/>
22     </AppliesTo>
23   </PolicyType>
24
25 </Definitions>
```

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

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

15 Policy Templates

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

15.1 XML Syntax

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

```
01 <PolicyTemplate id="xs:ID" name="xs:string"? type="xs:QName">
02
03   <Properties>
04     XML fragment
05   </Properties> ?
06
07   <PropertyConstraints>
08     <PropertyConstraint property="xs:string"
09                         constraintType="xs:anyURI"> +
10       constraint ?
11     </PropertyConstraint>
12   </PropertyConstraints> ?
13
14   policy type specific content ?
15
16 </PolicyTemplate>
```

15.2 Properties

The `PolicyTemplate` element has the following properties:

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

  The initial values are specified by providing an instance document of the XML schema of the corresponding Policy Type Properties. This instance document considers the inheritance structure deduced by the DerivedFrom property of the Policy Type referenced by the type attribute of the Policy Template.
- `PropertyConstraints`: This OPTIONAL element specifies constraints on the use of one or more of the Policy Type Properties of the Policy Type providing the property definitions for the Policy Template. Each constraint is specified by means of a separate nested `PropertyConstraint` element.
  The `PropertyConstraint` element has the following properties:

- `property`: The string value of this property is an XPath expression pointing to the property within the Policy Type Properties document that is constrained within the context of the Policy 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.

15.3 Example

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

```
01 <Definitions id="MyPolices" name="My Policies"
02   targetNamespace="http://www.example.com/SamplePolicies"
03   xmlns:spt="http://www.example.com/SamplePolicyTypes">
04
05   <Import importType="http://docs.oasis-open.org/tosca/ns/2011/12"
06           namespace="http://www.example.com/SamplePolicyTypes"/>
07
08   <PolicyTemplate id="MyHAPolicy"
09                   name="My High Availability Policy"
10                   type="bpt:HighAvailability">
11     <Properties>
12       <HAProperties>
13         <AvailabilityClass>4</AvailabilityClass>
14         <HeartbeatFrequency measuredIn="msec">
15            250
16         </HeartbeatFrequency>
17       </HAProperties>
18     </Properties>
19   </PolicyTemplate>
20
21 </Definitions>
```

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

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

16 Cloud Service Archive (CSAR)

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

16.1 Overall Structure of a CSAR

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

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

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

16.2 TOSCA Meta File

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

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

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

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

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

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

The name/value pairs are as follows:

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

- `Entry-Definitions`: This OPTIONAL name/value pair references a TOSCA Definitions file from the Definitions directory of the CSAR that SHOULD be used as entry point for processing the contents of the CSAR.
  Note, that a CSAR may contain multiple Definitions files. One reason for this is completeness, e.g. a Service Template defined in one of the Definitions files could refer to Node Types defined in another Definitions file that might be included in the Definitions directory to avoid importing it from external locations. The `Entry-Definitions` name/value pair is a hint to allow optimized processing of the set of files in the Definitions directory.

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

```
01 Name: <path-name_1>
02 Content-Type: type_1/subtype_1
03 <name_11>: <value_11>
04 <name_12>: <value_12>
05 ...
06 <name_1n>: <value_1n>
07
08 ...
09
10 Name: <path-name_k>
11 Content-Type: type_k/subtype_k
12 <name_k1>: <value_k1>
13 <name_k2>: <value_k2>
14 ...
15 <name_km>: <value_km>
```

The name/value pairs are as follows:

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

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

16.3 Example

Figure 7 depicts a sample Definitions file named `Payroll.tosca` containing a Service Template of an application. The application is a payroll application written in Java that MUST be deployed on a proper application server. The Service Template of the application defines the Node Template `Payroll Application`, the Node Template `Application Server`, as well as the Relationship Template `deployed_on`. The `Payroll Application` is associated with an EAR file (named `Payroll.ear`) which is provided as corresponding Deployment Artifact of the `Payroll Application` Node Template. An Amazon Machine Image (AMI) is the Deployment Artifact of the `Application Server` Node Template; this Deployment Artifact is a reference to the image in the Amazon EC2 environment. The Implementation Artifacts of some operations of the Node Templates are

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

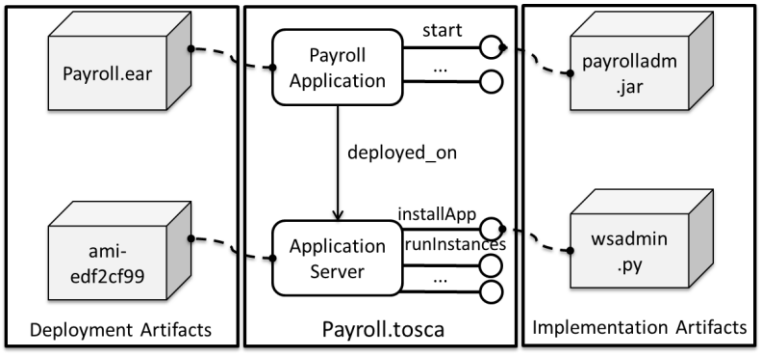

Figure 7: Sample Service Template

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

```
01 <Definitions id="PayrollDefinitions"
02              targetNamespace="http://www.example.com/ste"
03              xmlns:pay="http://www.example.com/ste/Types">
04
05   <Import namespace="http://www.example.com/ste/Types"
06           location="http://www.example.com/ste/Types/PayrollTypes.tosca"
07           importType=" http://docs.oasis-open.org/tosca/ns/2011/12"/>
08
09   <Types>
10      ...
11   </Types>
12
13   <ServiceTemplate id="Payroll" name="Payroll Service Template">
14
15     <TopologyTemplate ID="PayrollTemplate">
16
17       <NodeTemplate id="Payroll Application"
18                     type="pay:ApplicationNodeType">
19         ...
20
21         <DeploymentArtifacts>
22           <DeploymentArtifact name="PayrollEAR"
23                               type="http://www.example.com/
24                                     ns/tosca/2011/12/
25                                     DeploymentArtifactTypes/CSARref">
26                 EARs/Payroll.ear
27           </DeploymentArtifact>
28         </DeploymentArtifacts>
29
30       </NodeTemplate>
31
32       <NodeTemplate id="Application Server"
33                     type="pay:ApplicationServerNodeType">
```



```
34          ...
35
36          <DeploymentArtifacts>
37            <DeploymentArtifact name="ApplicationServerImage"
38                                type="http://www.example.com/
39                                      ns/tosca/2011/12/
40                                      DeploymentArtifactTypes/AMIref">
41                    ami-edf2cf99
42            </DeploymentArtifact>
43          </DeploymentArtifacts>
44
45        </NodeTemplate>
46
47        <RelationshipTemplate id="deployed_on"
48                              type="pay:deployed_on">
49            <SourceElement ref="Payroll Application"/>
50            <TargetElement ref="Application Server"/>
51        </RelationshipTemplate>
52
53      </TopologyTemplate>
54
55    </ServiceTemplate>
56
57 </Definitions>
```

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

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

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

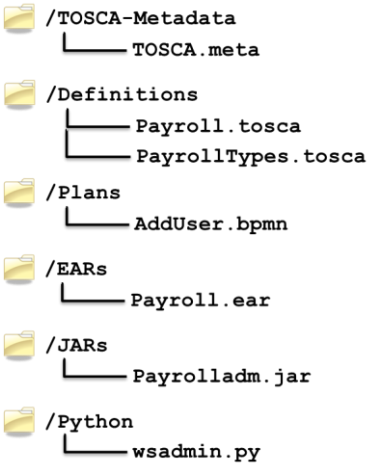

Figure 8: Structure of CSAR Sample

The `TOSCA.meta` file is as follows:

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

17 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.

18 Conformance

A TOSCA Definitions document conforms to this specification if it conforms to the TOSCA schema and follows the syntax and semantics defined in the normative portions of this specification. The TOSCA schema takes precedence over the TOSCA grammar (pseudo schema as defined in section 2.5), which in turn takes precedence over normative text, which in turn takes precedence over examples.

An implementation conforms to this specification if it can process a conformant TOSCA Definitions document according to the rules described in chapters 4 through 16 of this specification.

This specification allows extensions. Each implementation SHALL fully support all required functionality of the specification exactly as specified. The use of extensions SHALL NOT contradict nor cause the non-conformance of functionality defined in the specification.

Appendix A. Portability and Interoperability Considerations

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

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

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

Portability demands support of TOSCA elements.

Appendix B. Acknowledgements

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

Participants:

Aaron Zhang	Huawei Technologies Co., Ltd.
Adolf Hohl	NetApp
Afkham Azeez	WSO2
Al DeLucca	IBM
Alex Heneveld	Cloudsoft Corporation Limited
Allen Bannon	SAP AG
Anthony Rutkowski	Yaana Technologies, LLC
Arvind Srinivasan	IBM
Bryan Haynie	VCE
Bryan Murray	Hewlett-Packard
Chandrasekhar Sundaresh	CA Technologies
Charith Wickramarachchi	WSO2
Colin Hopkinson	3M HIS
Dale Moberg	Axway Software
Debojyoti Dutta	Cisco Systems
Dee Schur	OASIS
Denis Nothern	CenturyLink
Denis Weerasiri	WSO2
Derek Palma	Vnomic
Dhiraj Pathak	PricewaterhouseCoopers LLP:
Diane Mueller	ActiveState Software, Inc.
Doug Davis	IBM
Douglas Neuse	CA Technologies
Duncan Johnston-Watt	Cloudsoft Corporation Limited
Efraim Moscovich	CA Technologies
Frank Leymann	IBM
Gerd Breiter	IBM
James Thomason	Gale Technologies
Jan Ignatius	Nokia Siemens Networks GmbH & Co. KG
Jie Zhu	Huawei Technologies Co., Ltd.
John Wilmes	Individual
Joseph Malek	VCE
Ken Zink	CA Technologies
Kevin Poulter	SAP AG
Kevin Wilson	Hewlett-Packard
Koert Struijk	CA Technologies
Lee Thompson	Morphlabs, Inc.
li peng	Huawei Technologies Co., Ltd.
Marvin Waschke	CA Technologies
Mascot Yu	Huawei Technologies Co., Ltd.
Matthew Dovey	JISC Executive, University of Bristol
Matthew Rutkowski	IBM
Michael Schuster	SAP AG
Mike Edwards	IBM

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

3150

Appendix C. Complete TOSCA Grammar

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

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

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

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

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

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

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

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

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

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

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

```
52
53          <Policies>
54            <Policy name="xs:string"? policyType="xs:QName"
55                    policyRef="xs:QName"?>
56              policy specific content ?
57            </Policy> +
58          </Policies> ?
59
60          <Interfaces>
61            <Interface name="xs:NCName">
62              <Operation name="xs:NCName">
63                (
64                  <NodeOperation nodeRef="xs:IDREF"
65                                 interfaceName="xs:anyURI"
66                                 operationName="xs:NCName"/>
67                |
68                  <RelationshipOperation relationshipRef="xs:IDREF"
69                                         interfaceName="xs:anyURI"
70                                         operationName="xs:NCName"/>
71                |
72                  <Plan planRef="xs:IDREF"/>
73                )
74              </Operation> +
75            </Interface> +
76          </Interfaces> ?
77
78        </BoundaryDefinitions> ?
79
80        <TopologyTemplate>
81          (
82            <NodeTemplate id="xs:ID" name="xs:string"? type="xs:QName"
83                          minInstances="xs:integer"?
84                          maxInstances="xs:integer | xs:string"?>
85              <Properties>
86                XML fragment
87              </Properties> ?
88
89              <PropertyConstraints>
90                <PropertyConstraint property="xs:string"
91                                    constraintType="xs:anyURI">
92                  constraint ?
93                </PropertyConstraint> +
94              </PropertyConstraints> ?
95
96              <Requirements>
97                <Requirement id="xs:ID" name="xs:string" type="xs:QName"> +
98                  <Properties>
99                    XML fragment
100                   <Properties> ?
101                   <PropertyConstraints>
102                     <PropertyConstraint property="xs:string"
103                                         constraintType="xs:anyURI"> +
104                       constraint ?
105                     </PropertyConstraint>
106                   </PropertyConstraints> ?
107                 </Requirement>
108               </Requirements> ?
109
```



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

              <Policies>
                <Policy name="xs:string"? policyType="xs:QName"
                        policyRef="xs:QName"?>
                  policy specific content ?
                </Policy> +
              </Policies> ?

              <DeploymentArtifacts>
                <DeploymentArtifact name="xs:string"
                                    artifactType="xs:QName"
                                    artifactRef="xs:QName"?>
                  artifact specific content ?
                </DeploymentArtifact> +
              </DeploymentArtifacts> ?
            </NodeTemplate>
          |
            <RelationshipTemplate id="xs:ID" name="xs:string"?
                                  type="xs:QName">
              <Properties>
                XML fragment
              </Properties> ?

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

              <SourceElement ref="xs:IDREF"/>
              <TargetElement ref="xs:IDREF"/>

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

            </RelationshipTemplate>
          ) +
        </TopologyTemplate>

        <Plans>
```



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

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

            <InputParameters>
              <InputParameter name="xs:string" type="xs:string"
                              required="yes|no"?/> +
            </InputParameters> ?

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

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

          </Plan> +
        </Plans> ?

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

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

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

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

        <CapabilityDefinitions>
          <CapabilityDefinition name="xs:string"
                                capabilityType="xs:QName"
                                lowerBound="xs:integer"?
                                upperBound="xs:integer | xs:string"?>
            <Constraints>
```



```
              <Constraint constraintType="xs:anyURI">
                constraint type specific content
              </Constraint> +
            </Constraints> ?
          </CapabilityDefinition> +
        </CapabilityDefinitions>

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

        <Interfaces>
          <Interface name="xs:NCName | xs:anyURI">
            <Operation name="xs:NCName">
              <InputParameters>
                <InputParameter name="xs:string" type="xs:string"
                                required="yes|no"?/> +
              </InputParameters> ?
              <OutputParameters>
                <OutputParameter name="xs:string" type="xs:string"
                                 required="yes|no"?/> +
              </OutputParameters> ?
            </Operation> +
          </Interface> +
        </Interfaces> ?

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

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

        <RequiredContainerFeatures>
          <RequiredContainerFeature feature="xs:anyURI"/> +
        </RequiredContainerFeatures> ?

        <ImplementationArtifacts>
          <ImplementationArtifact interfaceName="xs:NCName | xs:anyURI"?
                                  operationName="xs:NCName"?
                                  artifactType="xs:QName"
                                  artifactRef="xs:QName"?>
            artifact specific content ?
          <ImplementationArtifact> +
        </ImplementationArtifacts> ?

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

      </NodeTypeImplementation>
    |
```



```
284       <RelationshipType name="xs:NCName"
285                         targetNamespace="xs:anyURI"?
286                         abstract="yes|no"?
287                         final="yes|no"?> +
288
289         <DerivedFrom typeRef="xs:QName"/> ?
290
291         <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
292
293         <InstanceStates>
294           <InstanceState state="xs:anyURI"> +
295         </InstanceStates> ?
296
297         <SourceInterfaces>
298           <Interface name="xs:NCName | xs:anyURI">
299             <Operation name="xs:NCName">
300               <InputParameters>
301                 <InputParameter name="xs:string" type="xs:string"
302                                 required="yes|no"?/> +
303               </InputParameters> ?
304               <OutputParameters>
305                 <OutputParameter name="xs:string" type="xs:string"
306                                  required="yes|no"?/> +
307               </OutputParameters> ?
308             </Operation> +
309           </Interface> +
310         </SourceInterfaces> ?
311
312         <TargetInterfaces>
313           <Interface name="xs:NCName | xs:anyURI">
314             <Operation name="xs:NCName">
315               <InputParameters>
316                 <InputParameter name="xs:string" type="xs:string"
317                                 required="yes|no"?/> +
318               </InputParameters> ?
319               <OutputParameters>
320                 <OutputParameter name="xs:string" type="xs:string"
321                                  required="yes|no"?/> +
322               </OutputParameters> ?
323             </Operation> +
324           </Interface> +
325         </TargetInterfaces> ?
326
327         <ValidSource typeRef="xs:QName"/> ?
328
329         <ValidTarget typeRef="xs:QName"/> ?
330
331       </RelationshipType>
332     |
333       <RelationshipTypeImplementation name="xs:NCName"
334                                       targetNamespace="xs:anyURI"?
335                                       relationshipType="xs:QName"
336                                       abstract="yes|no"?
337                                       final="yes|no"?>
338
339         <DerivedFrom relationshipTypeImplementationRef="xs:QName"/> ?
340
341         <RequiredContainerFeatures>
```



```
342           <RequiredContainerFeature feature="xs:anyURI"/> +
343         </RequiredContainerFeatures> ?
344
345         <ImplementationArtifacts>
346           <ImplementationArtifact interfaceName="xs:NCName | xs:anyURI"?
347                                   operationName="xs:NCName"?
348                                   artifactType="xs:QName"
349                                   artifactRef="xs:QName"?>
350             artifact specific content ?
351           <ImplementationArtifact> +
352         </ImplementationArtifacts> ?
353
354       </RelationshipTypeImplementation>
355     |
356       <RequirementType name="xs:NCName"
357                        targetNamespace="xs:anyURI"?
358                        abstract="yes|no"?
359                        final="yes|no"?
360                        requiredCapabilityType="xs:QName"?>
361
362         <DerivedFrom typeRef="xs:QName"/> ?
363
364         <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
365
366       </RequirementType>
367     |
368       <CapabilityType name="xs:NCName"
369                       targetNamespace="xs:anyURI"?
370                       abstract="yes|no"?
371                       final="yes|no"?>
372
373         <DerivedFrom typeRef="xs:QName"/> ?
374
375         <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
376
377       </CapabilityType>
378     |
379       <ArtifactType name="xs:NCName"
380                     targetNamespace="xs:anyURI"?
381                     abstract="yes|no"?
382                     final="yes|no"?>
383
384         <DerivedFrom typeRef="xs:QName"/> ?
385
386         <PropertiesDefinition element="xs:QName"? type="xs:QName"?/> ?
387
388       </ArtifactType>
389     |
390       <ArtifactTemplate id="xs:ID" name="xs:string"? type="xs:QName">
391
392         <Properties>
393           XML fragment
394         </Properties> ?
395
396         <PropertyConstraints>
397           <PropertyConstraint property="xs:string"
398                               constraintType="xs:anyURI"> +
399             constraint ?
```



```
          </PropertyConstraint>
        </PropertyConstraints> ?

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

      </ArtifactTemplate>
    |
      <PolicyType name="xs:NCName"
                  policyLanguage="xs:anyURI"?
                  abstract="yes|no"?
                  final="yes|no"?
                  targetNamespace="xs:anyURI"?>
        <Tags>
          <Tag name="xs:string" value="xs:string"/> +
        </Tags> ?

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

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

        <AppliesTo>
          <NodeTypeReference typeRef="xs:QName"/> +
        </AppliesTo> ?

        policy type specific content ?

      </PolicyType>
    |
      <PolicyTemplate id="xs:ID" name="xs:string"? type="xs:QName">

        <Properties>
          XML fragment
        </Properties> ?

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

        policy type specific content ?

      </PolicyTemplate>
    ) +

  </Definitions>
```

Appendix D. TOSCA Schema

TOSCA-v1.0.xsd:

```
<?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="Definitions">
  <xs:complexType>
   <xs:complexContent>
    <xs:extension base="tDefinitions"/>
   </xs:complexContent>
  </xs:complexType>
 </xs:element>
 <xs:complexType name="tDefinitions">
  <xs:complexContent>
   <xs:extension base="tExtensibleElements">
    <xs:sequence>
     <xs:element name="Extensions" minOccurs="0">
      <xs:complexType>
       <xs:sequence>
        <xs:element name="Extension" type="tExtension"
```

```
          maxOccurs="unbounded"/>
        </xs:sequence>
       </xs:complexType>
      </xs:element>
      <xs:element name="Import" type="tImport" minOccurs="0"
       maxOccurs="unbounded"/>
      <xs:element name="Types" minOccurs="0">
       <xs:complexType>
        <xs:sequence>
         <xs:any namespace="##other" processContents="lax" minOccurs="0"
          maxOccurs="unbounded"/>
        </xs:sequence>
       </xs:complexType>
      </xs:element>
      <xs:choice maxOccurs="unbounded">
       <xs:element name="ServiceTemplate" type="tServiceTemplate"/>
       <xs:element name="NodeType" type="tNodeType"/>
       <xs:element name="NodeTypeImplementation"
        type="tNodeTypeImplementation"/>
       <xs:element name="RelationshipType" type="tRelationshipType"/>
       <xs:element name="RelationshipTypeImplementation"
        type="tRelationshipTypeImplementation"/>
       <xs:element name="RequirementType" type="tRequirementType"/>
       <xs:element name="CapabilityType" type="tCapabilityType"/>
       <xs:element name="ArtifactType" type="tArtifactType"/>
       <xs:element name="ArtifactTemplate" type="tArtifactTemplate"/>
       <xs:element name="PolicyType" type="tPolicyType"/>
       <xs:element name="PolicyTemplate" type="tPolicyTemplate"/>
      </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="targetNamespace" type="xs:anyURI" use="required"/>
    </xs:extension>
   </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="tServiceTemplate">
   <xs:complexContent>
    <xs:extension base="tExtensibleElements">
     <xs:sequence>
      <xs:element name="Tags" type="tTags" minOccurs="0"/>
      <xs:element name="BoundaryDefinitions" type="tBoundaryDefinitions"
       minOccurs="0"/>
      <xs:element name="TopologyTemplate" type="tTopologyTemplate"/>
      <xs:element name="Plans" type="tPlans" minOccurs="0"/>
     </xs:sequence>
        <xs:attribute name="id" type="xs:ID" use="required"/>
        <xs:attribute name="name" type="xs:string" use="optional"/>
        <xs:attribute name="targetNamespace" type="xs:anyURI"/>
        <xs:attribute name="substitutableNodeType" type="xs:QName"
         use="optional"/>
      </xs:extension>
     </xs:complexContent>
    </xs:complexType>

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



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

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

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



```
      <xs:complexType>
       <xs:sequence>
        <xs:element name="Interface" type="tExportedInterface"
         maxOccurs="unbounded"/>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
    </xs:sequence>
   </xs:complexType>

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

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

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

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

   <xs:complexType name="tEntityTemplate" abstract="true">
    <xs:complexContent>
     <xs:extension base="tExtensibleElements">
      <xs:sequence>
```



```
227         <xs:element name="Properties" minOccurs="0">
228          <xs:complexType>
229           <xs:sequence>
230            <xs:any namespace="##other" processContents="lax"/>
231           </xs:sequence>
232          </xs:complexType>
233         </xs:element>
234         <xs:element name="PropertyConstraints" minOccurs="0">
235          <xs:complexType>
236           <xs:sequence>
237            <xs:element name="PropertyConstraint"
238             type="tPropertyConstraint" maxOccurs="unbounded"/>
239           </xs:sequence>
240          </xs:complexType>
241         </xs:element>
242        </xs:sequence>
243        <xs:attribute name="id" type="xs:ID" use="required"/>
244        <xs:attribute name="type" type="xs:QName" use="required"/>
245      </xs:extension>
246     </xs:complexContent>
247    </xs:complexType>
248
249    <xs:complexType name="tNodeTemplate">
250     <xs:complexContent>
251      <xs:extension base="tEntityTemplate">
252       <xs:sequence>
253        <xs:element name="Requirements" minOccurs="0">
254         <xs:complexType>
255          <xs:sequence>
256           <xs:element name="Requirement" type="tRequirement"
257            maxOccurs="unbounded"/>
258          </xs:sequence>
259         </xs:complexType>
260        </xs:element>
261        <xs:element name="Capabilities" minOccurs="0">
262         <xs:complexType>
263          <xs:sequence>
264           <xs:element name="Capability" type="tCapability"
265            maxOccurs="unbounded"/>
266          </xs:sequence>
267         </xs:complexType>
268        </xs:element>
269        <xs:element name="Policies" minOccurs="0">
270         <xs:complexType>
271          <xs:sequence>
272           <xs:element name="Policy" type="tPolicy"
273            maxOccurs="unbounded"/>
274          </xs:sequence>
275         </xs:complexType>
276        </xs:element>
277        <xs:element name="DeploymentArtifacts" type="tDeploymentArtifacts"
278         minOccurs="0"/>
279       </xs:sequence>
280       <xs:attribute name="name" type="xs:string" use="optional"/>
281       <xs:attribute name="minInstances" type="xs:int" use="optional"
282        default="1"/>
283       <xs:attribute name="maxInstances" use="optional" default="1">
284        <xs:simpleType>
```



```
285          <xs:union>
286           <xs:simpleType>
287            <xs:restriction base="xs:nonNegativeInteger">
288             <xs:pattern value="([1-9]+[0-9]*)"/>
289            </xs:restriction>
290           </xs:simpleType>
291           <xs:simpleType>
292            <xs:restriction base="xs:string">
293             <xs:enumeration value="unbounded"/>
294            </xs:restriction>
295           </xs:simpleType>
296          </xs:union>
297         </xs:simpleType>
298        </xs:attribute>
299       </xs:extension>
300      </xs:complexContent>
301     </xs:complexType>
302
303     <xs:complexType name="tTopologyTemplate">
304      <xs:complexContent>
305       <xs:extension base="tExtensibleElements">
306        <xs:choice maxOccurs="unbounded">
307         <xs:element name="NodeTemplate" type="tNodeTemplate"/>
308         <xs:element name="RelationshipTemplate"
309          type="tRelationshipTemplate"/>
310        </xs:choice>
311       </xs:extension>
312      </xs:complexContent>
313     </xs:complexType>
314
315     <xs:complexType name="tRelationshipType">
316      <xs:complexContent>
317       <xs:extension base="tEntityType">
318        <xs:sequence>
319         <xs:element name="InstanceStates"
320          type="tTopologyElementInstanceStates" minOccurs="0"/>
321         <xs:element name="SourceInterfaces" minOccurs="0">
322          <xs:complexType>
323           <xs:sequence>
324            <xs:element name="Interface" type="tInterface"
325             maxOccurs="unbounded"/>
326           </xs:sequence>
327          </xs:complexType>
328         </xs:element>
329         <xs:element name="TargetInterfaces" minOccurs="0">
330          <xs:complexType>
331           <xs:sequence>
332            <xs:element name="Interface" type="tInterface"
333             maxOccurs="unbounded"/>
334           </xs:sequence>
335          </xs:complexType>
336         </xs:element>
337         <xs:element name="ValidSource" minOccurs="0">
338          <xs:complexType>
339           <xs:attribute name="typeRef" type="xs:QName" use="required"/>
340          </xs:complexType>
341         </xs:element>
342         <xs:element name="ValidTarget" minOccurs="0">
```



```
343         <xs:complexType>
344          <xs:attribute name="typeRef" type="xs:QName" use="required"/>
345         </xs:complexType>
346        </xs:element>
347       </xs:sequence>
348      </xs:extension>
349     </xs:complexContent>
350    </xs:complexType>
351
352    <xs:complexType name="tRelationshipTypeImplementation">
353     <xs:complexContent>
354      <xs:extension base="tExtensibleElements">
355       <xs:sequence>
356        <xs:element name="Tags" type="tTags" minOccurs="0"/>
357        <xs:element name="DerivedFrom" minOccurs="0">
358         <xs:complexType>
359          <xs:attribute name="relationshipTypeImplementationRef"
360           type="xs:QName" use="required"/>
361         </xs:complexType>
362        </xs:element>
363        <xs:element name="RequiredContainerFeatures"
364         type="tRequiredContainerFeatures" minOccurs="0"/>
365        <xs:element name="ImplementationArtifacts"
366         type="tImplementationArtifacts" minOccurs="0"/>
367       </xs:sequence>
368       <xs:attribute name="name" type="xs:NCName" use="required"/>
369       <xs:attribute name="targetNamespace" type="xs:anyURI"
370        use="optional"/>
371       <xs:attribute name="relationshipType" type="xs:QName"
372        use="required"/>
373       <xs:attribute name="abstract" type="tBoolean" use="optional"
374        default="no"/>
375       <xs:attribute name="final" type="tBoolean" use="optional"
376        default="no"/>
377      </xs:extension>
378     </xs:complexContent>
379    </xs:complexType>
380
381    <xs:complexType name="tRelationshipTemplate">
382     <xs:complexContent>
383      <xs:extension base="tEntityTemplate">
384       <xs:sequence>
385        <xs:element name="SourceElement">
386         <xs:complexType>
387          <xs:attribute name="ref" type="xs:IDREF" use="required"/>
388         </xs:complexType>
389        </xs:element>
390        <xs:element name="TargetElement">
391         <xs:complexType>
392          <xs:attribute name="ref" type="xs:IDREF" use="required"/>
393         </xs:complexType>
394        </xs:element>
395        <xs:element name="RelationshipConstraints" minOccurs="0">
396         <xs:complexType>
397          <xs:sequence>
398           <xs:element name="RelationshipConstraint"
399            maxOccurs="unbounded">
400             <xs:complexType>
```



```
            <xs:sequence>
             <xs:any namespace="##other" processContents="lax"
              minOccurs="0"/>
            </xs:sequence>
            <xs:attribute name="constraintType" type="xs:anyURI"
             use="required"/>
           </xs:complexType>
          </xs:element>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="optional"/>
     </xs:extension>
    </xs:complexContent>
   </xs:complexType>

   <xs:complexType name="tNodeType">
    <xs:complexContent>
     <xs:extension base="tEntityType">
      <xs:sequence>
       <xs:element name="RequirementDefinitions" minOccurs="0">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="RequirementDefinition"
           type="tRequirementDefinition" maxOccurs="unbounded"/>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
       <xs:element name="CapabilityDefinitions" minOccurs="0">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="CapabilityDefinition"
           type="tCapabilityDefinition" maxOccurs="unbounded"/>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
       <xs:element name="InstanceStates"
        type="tTopologyElementInstanceStates" minOccurs="0"/>
       <xs:element name="Interfaces" minOccurs="0">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="Interface" type="tInterface"
           maxOccurs="unbounded"/>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
     </xs:extension>
    </xs:complexContent>
   </xs:complexType>

   <xs:complexType name="tNodeTypeImplementation">
    <xs:complexContent>
     <xs:extension base="tExtensibleElements">
      <xs:sequence>
       <xs:element name="Tags" type="tTags" minOccurs="0"/>
       <xs:element name="DerivedFrom" minOccurs="0">
```



```
        <xs:complexType>
         <xs:attribute name="nodeTypeImplementationRef" type="xs:QName"
          use="required"/>
        </xs:complexType>
       </xs:element>
       <xs:element name="RequiredContainerFeatures"
        type="tRequiredContainerFeatures" minOccurs="0"/>
       <xs:element name="ImplementationArtifacts"
        type="tImplementationArtifacts" minOccurs="0"/>
       <xs:element name="DeploymentArtifacts" type="tDeploymentArtifacts"
        minOccurs="0"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:NCName" use="required"/>
      <xs:attribute name="targetNamespace" type="xs:anyURI"
       use="optional"/>
      <xs:attribute name="nodeType" type="xs:QName" use="required"/>
      <xs:attribute name="abstract" type="tBoolean" use="optional"
       default="no"/>
      <xs:attribute name="final" type="tBoolean" use="optional"
       default="no"/>
     </xs:extension>
    </xs:complexContent>
   </xs:complexType>

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

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



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

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

    <xs:complexType name="tCapabilityType">
     <xs:complexContent>
      <xs:extension base="tEntityType"/>
     </xs:complexContent>
    </xs:complexType>

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



```
575         </xs:union>
576        </xs:simpleType>
577       </xs:attribute>
578      </xs:extension>
579     </xs:complexContent>
580    </xs:complexType>
581
582    <xs:complexType name="tCapability">
583     <xs:complexContent>
584      <xs:extension base="tEntityTemplate">
585       <xs:attribute name="name" type="xs:string" use="required"/>
586      </xs:extension>
587     </xs:complexContent>
588    </xs:complexType>
589
590    <xs:complexType name="tArtifactType">
591     <xs:complexContent>
592      <xs:extension base="tEntityType"/>
593     </xs:complexContent>
594    </xs:complexType>
595
596    <xs:complexType name="tArtifactTemplate">
597     <xs:complexContent>
598      <xs:extension base="tEntityTemplate">
599       <xs:sequence>
600        <xs:element name="ArtifactReferences" minOccurs="0">
601         <xs:complexType>
602          <xs:sequence>
603           <xs:element name="ArtifactReference" type="tArtifactReference"
604            maxOccurs="unbounded"/>
605          </xs:sequence>
606         </xs:complexType>
607        </xs:element>
608       </xs:sequence>
609       <xs:attribute name="name" type="xs:string" use="optional"/>
610      </xs:extension>
611     </xs:complexContent>
612    </xs:complexType>
613
614    <xs:complexType name="tDeploymentArtifacts">
615     <xs:sequence>
616      <xs:element name="DeploymentArtifact" type="tDeploymentArtifact"
617       maxOccurs="unbounded"/>
618     </xs:sequence>
619    </xs:complexType>
620
621    <xs:complexType name="tDeploymentArtifact">
622     <xs:complexContent>
623      <xs:extension base="tExtensibleElements">
624       <xs:attribute name="name" type="xs:string" use="required"/>
625       <xs:attribute name="artifactType" type="xs:QName" use="required"/>
626       <xs:attribute name="artifactRef" type="xs:QName" use="optional"/>
627      </xs:extension>
628     </xs:complexContent>
629    </xs:complexType>
630
631    <xs:complexType name="tImplementationArtifacts">
632     <xs:sequence>
```



```
      <xs:element name="ImplementationArtifact" maxOccurs="unbounded">
       <xs:complexType>
        <xs:complexContent>
         <xs:extension base="tImplementationArtifact"/>
        </xs:complexContent>
       </xs:complexType>
      </xs:element>
     </xs:sequence>
    </xs:complexType>

    <xs:complexType name="tImplementationArtifact">
     <xs:complexContent>
      <xs:extension base="tExtensibleElements">
       <xs:attribute name="interfaceName" type="xs:anyURI"
        use="optional"/>
       <xs:attribute name="operationName" type="xs:NCName"
        use="optional"/>
       <xs:attribute name="artifactType" type="xs:QName" use="required"/>
       <xs:attribute name="artifactRef" type="xs:QName" use="optional"/>
      </xs:extension>
     </xs:complexContent>
    </xs:complexType>

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

    <xs:complexType name="tPlan">
     <xs:complexContent>
      <xs:extension base="tExtensibleElements">
       <xs:sequence>
        <xs:element name="Precondition" type="tCondition" minOccurs="0"/>
        <xs:element name="InputParameters" minOccurs="0">
         <xs:complexType>
          <xs:sequence>
           <xs:element name="InputParameter" type="tParameter"
            maxOccurs="unbounded"/>
          </xs:sequence>
         </xs:complexType>
        </xs:element>
        <xs:element name="OutputParameters" minOccurs="0">
         <xs:complexType>
          <xs:sequence>
           <xs:element name="OutputParameter" type="tParameter"
            maxOccurs="unbounded"/>
          </xs:sequence>
         </xs:complexType>
        </xs:element>
        <xs:choice>
         <xs:element name="PlanModel">
          <xs:complexType>
           <xs:sequence>
            <xs:any namespace="##other" processContents="lax"/>
           </xs:sequence>
```



```
691           </xs:complexType>
692         </xs:element>
693         <xs:element name="PlanModelReference">
694          <xs:complexType>
695           <xs:attribute name="reference" type="xs:anyURI"
696            use="required"/>
697          </xs:complexType>
698         </xs:element>
699        </xs:choice>
700       </xs:sequence>
701       <xs:attribute name="id" type="xs:ID" use="required"/>
702       <xs:attribute name="name" type="xs:string" use="optional"/>
703       <xs:attribute name="planType" type="xs:anyURI" use="required"/>
704       <xs:attribute name="planLanguage" type="xs:anyURI" use="required"/>
705      </xs:extension>
706     </xs:complexContent>
707    </xs:complexType>
708
709    <xs:complexType name="tPolicyType">
710     <xs:complexContent>
711      <xs:extension base="tEntityType">
712       <xs:sequence>
713        <xs:element name="AppliesTo" type="tAppliesTo" minOccurs="0"/>
714       </xs:sequence>
715       <xs:attribute name="policyLanguage" type="xs:anyURI"
716        use="optional"/>
717      </xs:extension>
718     </xs:complexContent>
719    </xs:complexType>
720
721    <xs:complexType name="tPolicyTemplate">
722     <xs:complexContent>
723      <xs:extension base="tEntityTemplate">
724       <xs:attribute name="name" type="xs:string" use="optional"/>
725      </xs:extension>
726     </xs:complexContent>
727    </xs:complexType>
728
729    <xs:complexType name="tAppliesTo">
730     <xs:sequence>
731      <xs:element name="NodeTypeReference" maxOccurs="unbounded">
732       <xs:complexType>
733        <xs:attribute name="typeRef" type="xs:QName" use="required"/>
734       </xs:complexType>
735      </xs:element>
736     </xs:sequence>
737    </xs:complexType>
738
739    <xs:complexType name="tPolicy">
740     <xs:complexContent>
741      <xs:extension base="tExtensibleElements">
742       <xs:attribute name="name" type="xs:string" use="optional"/>
743       <xs:attribute name="policyType" type="xs:QName" use="required"/>
744       <xs:attribute name="policyRef" type="xs:QName" use="optional"/>
745      </xs:extension>
746     </xs:complexContent>
747    </xs:complexType>
748
```



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

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

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

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

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

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

<xs:complexType name="tExportedInterface">
 <xs:sequence>
  <xs:element name="Operation" type="tExportedOperation"
   maxOccurs="unbounded"/>
 </xs:sequence>
 <xs:attribute name="name" type="xs:anyURI" use="required"/>
</xs:complexType>
```



```
807
808     <xs:complexType name="tOperation">
809      <xs:complexContent>
810       <xs:extension base="tExtensibleElements">
811        <xs:sequence>
812         <xs:element name="InputParameters" minOccurs="0">
813          <xs:complexType>
814           <xs:sequence>
815            <xs:element name="InputParameter" type="tParameter"
816             maxOccurs="unbounded"/>
817           </xs:sequence>
818          </xs:complexType>
819         </xs:element>
820         <xs:element name="OutputParameters" minOccurs="0">
821          <xs:complexType>
822           <xs:sequence>
823            <xs:element name="OutputParameter" type="tParameter"
824             maxOccurs="unbounded"/>
825           </xs:sequence>
826          </xs:complexType>
827         </xs:element>
828        </xs:sequence>
829        <xs:attribute name="name" type="xs:NCName" use="required"/>
830       </xs:extension>
831      </xs:complexContent>
832     </xs:complexType>
833
834     <xs:complexType name="tExportedOperation">
835      <xs:choice>
836       <xs:element name="NodeOperation">
837        <xs:complexType>
838         <xs:attribute name="nodeRef" type="xs:IDREF" use="required"/>
839         <xs:attribute name="interfaceName" type="xs:anyURI"
840          use="required"/>
841         <xs:attribute name="operationName" type="xs:NCName"
842          use="required"/>
843        </xs:complexType>
844       </xs:element>
845       <xs:element name="RelationshipOperation">
846        <xs:complexType>
847         <xs:attribute name="relationshipRef" type="xs:IDREF"
848          use="required"/>
849         <xs:attribute name="interfaceName" type="xs:anyURI"
850          use="required"/>
851         <xs:attribute name="operationName" type="xs:NCName"
852          use="required"/>
853        </xs:complexType>
854       </xs:element>
855       <xs:element name="Plan">
856        <xs:complexType>
857         <xs:attribute name="planRef" type="xs:IDREF" use="required"/>
858        </xs:complexType>
859       </xs:element>
860      </xs:choice>
861      <xs:attribute name="name" type="xs:NCName" use="required"/>
862     </xs:complexType>
863
864     <xs:complexType name="tCondition">
```



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

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

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

   <xs:complexType name="tRequiredContainerFeatures">
    <xs:sequence>
     <xs:element name="RequiredContainerFeature"
      type="tRequiredContainerFeature" maxOccurs="unbounded"/>
    </xs:sequence>
   </xs:complexType>

   <xs:complexType name="tRequiredContainerFeature">
    <xs:attribute name="feature" type="xs:anyURI" use="required"/>
   </xs:complexType>

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

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

  </xs:schema>
```


Appendix E. Sample

This appendix contains the full sample used in this specification.

E.1 Sample Service Topology Definition

```
<Definitions name="MyServiceTemplateDefinition"
             targetNamespace="http://www.example.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="xs:string"/>
            <element name="IPAddress" type="xs:string"/>
            <element name="HeapSize" type="xs:positiveInteger"/>
            <element name="SoapPort" type="xs:positiveInteger"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </Types>

  <ServiceTemplate id="MyServiceTemplate">

    <Tags>
      <Tag name=”author” value=”someone@example.com”/>
    </Tags>

    <TopologyTemplate id="SampleApplication">

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

      <NodeTemplate id="MyAppServer"
                    name="My Application Server"
```

```
                    nodeType="abc:ApplicationServer"
                    minInstances="0"
                    maxInstances="unbounded"/>

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

  </TopologyTemplate>

  <Plans>
    <Plan id="DeployApplication"
          name="Sample Application Build Plan"
          planType="http://docs.oasis-
           open.org/tosca/ns/2011/12/PlanTypes/BuildPlan"
          planLanguage="http://www.omg.org/spec/BPMN/20100524/MODEL">

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

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

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

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

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

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

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

      <Plan id="RemoveApplication"
            planType="http://docs.oasis-
             open.org/tosca/ns/2011/12/PlanTypes/TerminationPlan"
            planLanguage="http://docs.oasis-
```



```
              open.org/wsbpel/2.0/process/executable">
          <PlanModelReference reference="prj:RemoveApp"/>
        </Plan>
      </Plans>

    </ServiceTemplate>

    <NodeType name="Application">
      <documentation xml:lang="EN">
        A reusable definition of a node type representing an
        application that can be deployed on application servers.
      </documentation>
      <NodeTypeProperties element="ApplicationProperties"/>
      <InstanceStates>
        <InstanceState state="http://www.example.com/started"/>
        <InstanceState state="http://www.example.com/stopped"/>
      </InstanceStates>
      <Interfaces>
        <Interface name="DeploymentInterface">
          <Operation name="DeployApplication">
            <InputParameters>
              <InputParamter name="InstanceName"
                             type="xs:string"/>
              <InputParamter name="AppServerHostname"
                             type="xs:string"/>
              <InputParamter name="ContextRoot"
                             type="xs:string"/>
            </InputParameters>
          </Operation>
        </Interface>
      </Interfaces
    </NodeType>

    <NodeType name="ApplicationServer"
              targetNamespace="http://www.example.com/sample">
      <NodeTypeProperties element="AppServerProperties"/>
      <Interfaces>
        <Interface name="MyAppServerInterface">
          <Operation name="AcquireNetworkAddress"/>
          <Operation name="DeployApplicationServer"/>
        </Interface>
      </Interfaces>
    </NodeType>

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

  </Definitions>
```


Appendix F. Revision History

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

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

wd-13	2012-10-26	Thomas Spatzier, Derek Palma	Changes for JIRA Issue TOSCA-10: More fixes to formatting and references in document according to OASIS staff comments Changes for JIRA Issues TOSCA-36/37: Added `PolicyType` and `PolicyTemplate` elements to allow for re-usable definitions of policies. Changes for JIRA Issue TOSCA-57: Restructure TOSCA schema to allow for better modular definitions and separation of concerns. Changes for JIRA Issue TOSCA-59: Rewording to clarify overriding of deployment artifacts of Node Templates. Some additional minor changes in wording. Changes for JIRA Issue TOSCA-63: clarifying rewording
wd-14	2012-11-19	Thomas Spatzier	Changes for JIRA Issue TOSCA-76: Add Entry-Definitions property for TOSCA.meta file. Multiple general editorial fixes: Typos, namespaces and MIME types used in examples Fixed schema problems in tPolicyTemplate and tPolicyType Added text to Conformance section.
wd-15	2013-02-26	Thomas Spatzier	Changes for JIRA Issue TOSCA-79: Handle public review comments: fixes of typos and other non-material changes like inconsistencies between the specification document and the schema in this document and the TOSCA schema

4694