TOSCA Version 2.0

Committee Specification Draft 01

23 April 2020

This stage:
https://docs.oasis-open.org/tosca/TOSCA/v2.0/csd01/TOSCA-v2.0-csd01.docx (Authoritative)
https://docs.oasis-open.org/tosca/TOSCA/v2.0/csd01/TOSCA-v2.0-csd01.html
https://docs.oasis-open.org/tosca/TOSCA/v2.0/csd01/TOSCA-v2.0-csd01.pdf

Previous stage:
N/A

Latest stage:
https://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.docx (Authoritative)
https://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.html
https://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.pdf

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

Chairs:
Paul Lipton (paul.lipton@live.com), Individual Member
Chris Lauwers (lauwers@ubicity.com), Individual Member

Editors:
Chris Lauwers (lauwers@ubicity.com), Individual Member
Calin Curescu (calin.curescu@ericsson.com), Ericsson

Additional artifacts:
This prose specification is one component of a Work Product that also includes:
- TBD - schemas?

Related work:
This specification replaces or supersedes:
- *Topology and Orchestration Specification for Cloud Applications Version 1.0*. Edited by Derek Palma and Thomas Spatzier. OASIS Standard. Latest version: http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html.
- *TOSCA Simple Profile in YAML Version 1.3*. Edited by Matt Rutkowski, Chris Lauwers, Claude Noshpitz, and Calin Curescu. Latest version: https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/TOSCA-Simple-Profile-YAML-v1.3.html.

This specification is related to:
- *Introduction to TOSCA Version 2.0.* Edited by Chris Lauwers and Calin Curescu. Work in progress.

Declared XML namespaces:
- http://docs.oasis-open.org/tosca/ns/2.0

Abstract:
This document defines a simplified profile of the TOSCA version 1.0 specification in a YAML rendering which is intended to simplify the authoring of TOSCA service templates. This profile defines a less verbose and more human-readable YAML rendering, reduced level of indirection between different modeling artifacts as well as the assumption of a base type system.

The OASIS TOSCA TC works to enhance the portability of cloud applications and services across their entire lifecycle. TOSCA will enable the interoperable description of application and infrastructure cloud services, the relationships between parts of the service, and the operational behavior of these services (e.g., deploy, patch, shutdown) independent of the supplier creating the service or of any particular cloud provider or hosting technology. TOSCA will also make it possible for higher-level operational behavior to be associated with cloud infrastructure management.

By increasing service and application portability in a vendor-neutral ecosystem, TOSCA will enable:

- Portable deployment to any compliant cloud
- Smoother migration of existing applications to the cloud
- Flexible bursting (consumer choice)
- Dynamic, multi-cloud provider applications

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 stage" location noted above for possible later revisions of this document. Any other numbered Versions and other technical work produced by the Technical Committee (TC) are listed at https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca#technical.

TC members should send comments on this specification to the TC's email list. Others should send comments to the TC's public comment list, after subscribing to it by following the instructions at the "Send A Comment" button on the TC's web page at https://www.oasis-open.org/committees/tosca/.

This specification is provided under the RF on Limited Terms Mode of the OASIS IPR Policy, the mode chosen when the Technical Committee was established. 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 TC's web page (https://www.oasis-open.org/committees/tosca/ipr.php).

Note that any machine-readable content (Computer Language Definitions) declared Normative for this Work Product is provided in separate plain text files. In the event of a discrepancy between any such plain text file and display content in the Work Product's prose narrative document(s), the content in the separate plain text file prevails.

Citation format:

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

[TOSCA-v2.0]

TOSCA Version 2.0. Edited by Chris Lauwers and Calin Curescu. 23 April 2020. OASIS Committee Specification Draft 01. https://docs.oasis-open.org/tosca/TOSCA/v2.0/csd01/TOSCA-v2.0-csd01.html. Latest stage: https://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.html.

Notices

Copyright © OASIS Open 2020. All Rights Reserved.

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

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

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

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

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

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

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

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

Table of Contents

1 Introduction

[All text is normative unless otherwise labeled]

1.1 IPR Policy

This specification is provided under the RF on Limited Terms Mode of the OASIS IPR Policy, the mode chosen when the Technical Committee was established. 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 TC's web page (https://www.oasis-open.org/committees/tosca/ipr.php).

1.2 Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] and [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.3 Normative References

[RFC2119]   Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC8174]   Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <http://www.rfc-editor.org/info/rfc8174>.

[YAML-1.2]   YAML, Version 1.2, 3rd Edition, Patched at 2009-10-01, Oren Ben-Kiki, Clark Evans, Ingy döt Net http://www.yaml.org/spec/1.2/spec.html

[YAML-TS-1.1] Timestamp Language-Independent Type for YAML Version 1.1, Working Draft 2005-01-18, http://yaml.org/type/timestamp.html

1.4 Non-Normative References

[RFC3552]   Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, <https://www.rfc-editor.org/info/rfc3552>.

[Apache]   Apache Server, https://httpd.apache.org/

[Chef]   Chef, https://wiki.opscode.com/display/chef/Home

[NodeJS]   Node.js, https://nodejs.org/

[Puppet]   Puppet, http://puppetlabs.com/

[WordPress]   WordPress, https://wordpress.org/

[Maven-Version]   Apache Maven version policy draft: https://cwiki.apache.org/confluence/display/MAVEN/Version+number+policy

[JSON-Spec]   The JSON Data Interchange Format (ECMA and IETF versions):

- http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf
- https://tools.ietf.org/html/rfc7158

[JSON-Schema]   JSON Schema specification:

- http://json-schema.org/documentation.html

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

[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/

[IANA register for Hash Function Textual Names]   https://www.iana.org/assignments/hash-
               function-text-names/hash-function-text-names.xhtml

[Jinja2]   Jinja2, jinja.pocoo.org/

[Twig]   Twig, https://twig.symfony.com

(**Note**: Each reference to a separate document or artifact in this work must be listed here and must be identified as either a Normative or a Non-Normative Reference.

For all References – Normative and Non-Normative:

Recommended approach: Set up [Reference] label elements as "Bookmarks", then create hyperlinks to them within the document. (**Here's how:** Insert hyperlink→Place in this document→scroll down to Bookmarks, select appropriate one.)

Use the "Ref" paragraph style to format references.

The proper format for citation of technical work produced by an OASIS TC (whether Standards Track or Non-Standards Track) is:

[Citation Label] Work Product title (italicized). Edited by Albert Alston, Bob Ballston, and Calvin Carlson. Approval date (DD Month YYYY). OASIS Stage Identifier and Revision Number (*e.g.*, OASIS Committee Specification Draft 01). Principal URI (version-specific URI, *e.g.*, with stage component: somespec-v1.0-csd01.html). Latest version: (latest version URI, without stage identifiers).

For example:

[OpenDoc-1.2] *Open Document Format for Office Applications (OpenDocument) Version 1.2*. Edited by Patrick Durusau and Michael Brauer. 19 January 2011. OASIS Committee Specification Draft 07. http://docs.oasis-open.org/office/v1.2/csd07/OpenDocument-v1.2-csd07.html. Latest version: http://docs.oasis-open.org/office/v1.2/OpenDocument-v1.2.html.

Reference sources:

For references to **IETF RFCs**, use the approved citation formats at: http://docs.oasis-open.org/templates/ietf-rfc-list/ietf-rfc-list.html.

For references to **W3C Recommendations**, use the approved citation formats at: http://docs.oasis-open.org/templates/w3c-recommendations-list/w3c-recommendations-list.html.

Remove this note before submitting for publication.)

2 Overview

2.1 Objective

TOSCA specifies a rendering of TOSCA which aims to provide a more accessible syntax as well as a more concise and incremental expressiveness of the TOSCA DSL in order to minimize the learning curve and speed the adoption of the use of TOSCA to portably describe cloud applications.

This proposal describes a YAML rendering for TOSCA. YAML is a human friendly data serialization standard (http://yaml.org/) with a syntax much easier to read and edit than XML. As there are a number of DSLs encoded in YAML, a YAML encoding of the TOSCA DSL makes TOSCA more accessible by these communities.

This proposal prescribes an isomorphic rendering in YAML of a subset of the TOSCA v1.0 XML specification ensuring that TOSCA semantics are preserved and can be transformed from XML to YAML or from YAML to XML. Additionally, in order to streamline the expression of TOSCA semantics, the YAML rendering is sought to be more concise and compact through the use of the YAML syntax.

2.2 Summary of key TOSCA concepts

The TOSCA metamodel uses the concept of service templates that describe cloud workloads as a topology template, which is a graph of node templates modeling the components a workload is made up of and of relationship templates modeling the relations between those components. TOSCA further provides a type system of node types to describe the possible building blocks for constructing a service template, as well as relationship types to describe possible kinds of relations. Both node and relationship types may define lifecycle operations to implement the behavior an orchestration engine can invoke when instantiating a service template. For example, a node type for some software product might provide a 'create' operation to handle the creation of an instance of a component at runtime, or a 'start' or 'stop' operation to handle a start or stop event triggered by an orchestration engine. Those lifecycle operations are backed by implementation artifacts such as scripts or Chef recipes that implement the actual behavior.

An orchestration engine processing a TOSCA service template uses the mentioned lifecycle operations to instantiate single components at runtime, and it uses the relationship between components to derive the order of component instantiation. For example, during the instantiation of a two-tier application that includes a web application that depends on a database, an orchestration engine would first invoke the 'create' operation on the database component to install and configure the database, and it would then invoke the 'create' operation of the web application to install and configure the application (which includes configuration of the database connection).

TOSCA assumes a number of base types (node types and relationship types) to be supported by each compliant environment such as a 'Compute' node type, a 'Network' node type or a generic 'Database' node type. Furthermore, it is envisioned that a large number of additional types for use in service templates will be defined by a community over time. Therefore, template authors in many cases will not have to define types themselves but can simply start writing service templates that use existing types. In addition, TOSCA will provide means for easily customizing and extending existing types, for example by providing a customized 'create' script for some software.

2.3 Implementations

Different kinds of processors and artifacts qualify as implementations of TOSCA. Those that this specification is explicitly mentioning or referring to fall into the following categories:

- TOSCA YAML service template (or "service template"): A YAML document artifact containing a (TOSCA) topology template (see sections 3.9 "Service template definition") that represents a Cloud application. (see sections 3.8 "Topology template definition")

- TOSCA processor (or "processor"): An engine or tool that is capable of parsing and interpreting a TOSCA service template for a particular purpose. For example, the purpose could be validation, translation or visual rendering.
- TOSCA orchestrator (also called orchestration engine): A TOSCA processor that interprets a TOSCA service template or a TOSCA CSAR in order to instantiate, deploy, and manage the described application in a Cloud.
- TOSCA generator: A tool that generates a TOSCA service template. An example of generator is a modeling tool capable of generating or editing a TOSCA service template (often such a tool would also be a TOSCA processor).
- TOSCA archive (or TOSCA Cloud Service Archive, or "CSAR"): a package artifact that contains a TOSCA service template and other artifacts usable by a TOSCA orchestrator to deploy an application.

The above list is not exclusive. The above definitions should be understood as referring to and implementing TOSCA as described in this document.

2.4 Terminology

The TOSCA language introduces a YAML grammar for describing service templates by means of Topology Templates and towards enablement of interaction with a TOSCA instance model perhaps by external APIs or plans. The primary focus currently is on design time aspects, i.e. the description of services to ensure their exchange between Cloud providers, TOSCA Orchestrators and tooling.

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

2.5 Glossary

The following terms are used throughout this specification and have the following definitions when used in context of this document.

Term	Definition
Instance Model	A deployed service is a running 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 declarative workflow that is automatically generated based on the node templates and relationship templates defined in the Topology Template.
Node Template	A *Node Template* specifies the occurrence of a component node as part of a Topology Template. Each Node Template refers to a Node Type that defines the semantics of the node (e.g., properties, attributes, requirements, capabilities, interfaces). Node Types are defined separately for reuse purposes.
Relationship Template	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 relationship (e.g., properties, attributes, interfaces, etc.). Relationship Types are defined separately for reuse purposes.
Service Template	A *Service Template* is typically used to specify the "topology" (or structure) and "orchestration" (or invocation of management behavior) of IT services so that they can be provisioned and managed in accordance with constraints and policies.
	Specifically, TOSCA Service Templates optionally allow definitions of a TOSCA Topology Template, TOSCA types (e.g., Node, Relationship, Capability, Artifact, etc.), groupings, policies and constraints along with any input or output declarations.

Topology Model	The term Topology Model is often used synonymously with the term Topology Template with the use of "model" being prevalent when considering a Service Template's topology definition as an **abstract representation** of an application or service to facilitate understanding of its functional components and by eliminating unnecessary details.
Topology Template	A Topology Template defines the structure of a service in the context of a Service Template. A Topology Template consists of a set of Node Template and Relationship Template definitions that together define the topology model of a service as a (not necessarily connected) directed graph.The term Topology Template is often used synonymously with the term Topology Model. The distinction is that a topology template can be used to instantiate and orchestrate the model as a **reusable pattern** and includes all details necessary to accomplish it.
Abstract Node Template	An abstract node template is a node template that doesn't define any implementations for the TOSCA lifecycle management operations. Service designers explicitly mark node templates as abstract using the substitute directive. TOSCA orchestrators provide implementations for abstract node templates by finding substituting templates for those node templates.
No-Op Node Template	A No-Op node template is a node template that does not specify implementations for any of its operations, but is not marked as abstract. No-op templates only act as placeholders for information to be used by other node templates and do not need to be orchestrated.

3 TOSCA definitions in YAML

Except for the examples, this section is **normative** and describes all of the YAML grammar, definitions and block structure for all keys and mappings that are defined for the TOSCA Version 2.0 specification that are needed to describe a TOSCA Service Template (in YAML).

3.1 TOSCA Metamodel

This section defines the models and the modeling goals that comprise the TOSCA Version 2.0 specification.

3.1.1 Modeling concepts and goals

TBD. Here we should have selected core concepts of TOSCA 1.0 from section "3   Core Concepts and Usage Pattern" and this section should be a more in-depth section than section 2.1 in this document.

3.1.2 Modeling definitions and reuse

The TOSCA metamodel includes complex definitions used in types and templates. Reuse concepts simplify the design of TOSCA templates by allowing relevant TOSCA entities to use and/or modify definitions already specified during entity type design. The following four concepts are clarified next:

- **Definition**:
  - The TOSCA specification is based on defining modeling entities.
  - Entity definitions are based on different sets of keynames (with specific syntax and semantics) that are associated with values (of a specific format).
- **Derivation**:
  - Specific TOSCA entities support a type definition.
  - When defining a type, it can be derived from a parent type.
  - The derivation rules describe what (keyname) definitions are inherited from the parent type and further if and how they can be expanded or modified.
- **Refinement**:
  - Definitions within a type definition consist of the definition of keynames and other TOSCA entities (e.g. properties, requirements, capabilities, etc.).
  - The refinement rules pertaining to an entity describe how such entity definitions that are inherited from the parent type during a type derivation can be expanded or modified.
- **Assignment**:
  - When creating a topology template, we specify several entities that are part of the template (e.g. nodes, relationships, groups, etc.).
  - When adding such an entity in the topology template, for some definitions that appear in the corresponding entity type (e.g. properties, operations, requirements, etc.) we may (or must) assign a certain specification (or value).

3.1.3 Goal of the derivation and refinement rules

The main reason for derivation and refinement rules is to create a framework useful for a consistent TOSCA type profile creation. The intuitive idea is that a derived type follows to a large extent the structure and behavior of a parent type, otherwise it would be better to define a new "not derived" type.

The guideline regarding the derivation rules is that a node of a derived type should be usable instead of a node of the parent type during the selection and substitution mechanisms. These two mechanisms are used by TOSCA templates to connect to TOSCA nodes and services defined by other TOSCA templates:

- The selection mechanism allows a node instance created a-priori by another service template to be selected for usage (i.e. building relationships) to the current TOSCA template.
- The substitution mechanism allows a node instance to be represented by a service created simultaneously via a substitution template.

It is relevant to emphasize the cross-template usage, as only in this case we deal with templates defined at different design time-points, with potentially different editing and maintenance restrictions.

3.1.4 Required Keynames

The TOSCA metamodel includes complex definitions used in types (e.g., Node Types, Relationship Types, Capability Types, Data Types, etc.) and templates (e.g. Service Template, Topology Template, Node Template, etc.) each of which include their own list of reserved keynames that are sometimes marked as **required**. If a keyname is marked as required it **MUST** be defined in that particular definition context. Note that in the context of type definitions, types may be used to derive other types, and keyname definitions **MAY** be inherited from parent types (according to the derivation rules of that type entity). If a keyname definition is inherited, the derived type does not have to provide such definition.

3.2 TOSCA Service

A TOSCA Service is specified by a TOSCA Service Template.

3.2.1 Service Template definition

A TOSCA Service Template (YAML) document contains element definitions of building blocks for cloud application, or complete models of cloud applications. This section describes the top-level structural elements (TOSCA keynames) along with their grammars, which are allowed to appear in a TOSCA Service Template document.

3.2.1.1 Keynames

The following is the list of recognized keynames for a TOSCA Service Template definition:

Keyname	Required	Type	Description
tosca_definitions_version	yes	string	Defines the version of the TOSCA specification the template (grammar) complies with.
namespace	no	URI	The default (target) namespace for all unqualified Types defined within the Service Template.
metadata	no	map of string	Defines a section used to declare additional metadata information. Domain-specific TOSCA profile specifications may define keynames that are required for their implementations.
description	no	description	Declares a description for this Service Template and its contents.
dsl_definitions	no	N/A	Declares optional DSL-specific definitions and conventions. For example, in YAML, this allows defining reusable YAML macros (i.e., YAML alias anchors) for use throughout the TOSCA Service Template.

Keyname	Required	Type	Description
repositories	no	map of Repository definitions	Declares the map of external repositories which contain artifacts that are referenced in the service template along with their addresses and necessary credential information used to connect to them in order to retrieve the artifacts.
imports	no	list of Import Definitions	Declares a list import statements pointing to external TOSCA Definitions documents. For example, these may be file location or URIs relative to the service template file within the same TOSCA CSAR file.
artifact_types	no	map of Artifact Types	This section contains an optional map of artifact type definitions for use in the service template
data_types	no	map of Data Types	Declares a map of optional TOSCA Data Type definitions.
capability_types	no	map of Capability Types	This section contains an optional map of capability type definitions for use in the service template.
interface_types	no	map of Interface Types	This section contains an optional map of interface type definitions for use in the service template.
relationship_types	no	map of Relationship Types	This section contains a map of relationship type definitions for use in the service template.
node_types	no	map of Node Types	This section contains a map of node type definitions for use in the service template.
group_types	no	map of Group Types	This section contains a map of group type definitions for use in the service template.
policy_types	no	list of Policy Types	This section contains a list of policy type definitions for use in the service template.
topology_template	no	Topology Template definition	Defines the topology template of an application or service, consisting of node templates that represent the application's or service's components, as well as relationship templates representing relations between the components.

3.2.1.1.1 Metadata keynames

The following is the list of recognized metadata keynames for a TOSCA Service Template definition:

Keyname	Required	Type	Description
template_name	no	string	Declares a descriptive name for the template.
template_author	no	string	Declares the author(s) or owner of the template.
template_version	no	string	Declares the version string for the template.

3.2.1.2 Grammar

The overall structure of a TOSCA Service Template and its top-level key collations using TOSCA is shown below:

```
# Required TOSCA Definitions version string
tosca_definitions_version: <value>  # Required, see section 3.1 for usage
```

```
    namespace: <URI>                        # Optional, see section 3.2 for usage

    # Optional metadata keyname: value pairs
    metadata:
      template_name: <value>                # Optional, name of this service
    template
      template_author: <value>             # Optional, author of this service
    template
      template_version: <value>            # Optional, version of this service
    template
      #  More optional entries of domain or profile specific metadata keynames

    # Optional description of the definitions inside the file.
    description: <template_type_description>

    dsl_definitions:
      # map of YAML alias anchors (or macros)

    repositories:
      # map of external repository definitions which host TOSCA artifacts

    imports:
      # ordered list of import definitions

    artifact_types:
      # map of artifact type definitions

    data_types:
      # map of datatype definitions

    capability_types:
      # map of capability type definitions

    interface_types
      # map of interface type definitions

    relationship_types:
      # map of relationship type definitions

    node_types:
      # map of node type definitions

    group_types:
      # map of group type definitions

    policy_types:
      # map of policy type definitions

    topology_template:
      # topology template definition of the cloud application or service
```

3.2.1.2.1 Requirements

- The URI value "http://docs.oasis-open.org/tosca", as well as all (path) extensions to it, SHALL be reserved for TOSCA approved specifications and work. That means Service Templates that do not originate from a TOSCA approved work product MUST NOT use it, in any form, when declaring a (default) Namespace.
- The key "tosca_definitions_version" SHOULD be the first line of each Service Template.

3.2.1.2.2 Notes

- TOSCA Service Templates do not have to contain a topology_template and MAY contain simply type definitions (e.g., Artifact, Interface, Capability, Node, Relationship Types, etc.) and be imported for use as type definitions in other TOSCA Service Templates.

3.2.1.3 Top-level keyname definitions

3.2.1.3.1 tosca_definitions_version

This required element provides a means to include a reference to the TOSCA specification within the TOSCA Definitions YAML file. It is an indicator for the version of the TOSCA grammar that should be used to parse the remainder of the document.

3.2.1.3.1.1 Keyname

```
tosca_definitions_version
```

3.2.1.3.1.2 Grammar

Single-line form:

```
tosca_definitions_version: <tosca_simple_profile_version>
```

3.2.1.3.1.3 Examples:

TOSCA Version 2.0 specification using the defined namespace alias (see Section 3.2.2.1 TOSCA Namespace URI and alias):

```
tosca_definitions_version: tosca_2_0
```

TOSCA Version 2.0 specification using the fully defined (target) namespace (see Section 3.2.2.1 TOSCA Namespace URI and alias):

```
tosca_definitions_version: http://docs.oasis-open.org/tosca/ns/2.0
```

3.2.1.3.2 metadata

This keyname is used to associate domain-specific metadata with the Service Template. The metadata keyname allows a declaration of a map of keynames with string values.

3.2.1.3.2.1 Keyname

```
metadata
```

3.2.1.3.2.2 Grammar

```
metadata:
  <map_of_string_values>
```

3.2.1.3.2.3 Example

```
metadata:
  creation_date: 2015-04-14
  date_updated: 2015-05-01
  status: developmental
```

3.2.1.3.3 template_name

This optional metadata keyname can be used to declare the name of service template as a single-line string value.

3.2.1.3.3.1 Keyname

```
template_name
```

3.2.1.3.3.2 Grammar

```
template_name: <name string>
```

3.2.1.3.3.3 Example

```
template_name: My service template
```

3.2.1.3.3.4 Notes

- Some service templates are designed to be referenced and reused by other service templates. Therefore, in these cases, the template_name value SHOULD be designed to be used as a unique identifier through the use of namespacing techniques.

3.2.1.3.4 template_author

This optional metadata keyname can be used to declare the author(s) of the service template as a single-line string value.

3.2.1.3.4.1 Keyname

```
template_author
```

3.2.1.3.4.2 Grammar

```
template_author: <author string>
```

3.2.1.3.4.3 Example

```
template_author: My service template
```

3.2.1.3.5 template_version

This optional metadata keyname can be used to declare a domain specific version of the service template as a single-line string value.

3.2.1.3.5.1 Keyname

```
template_version
```

3.2.1.3.5.2 Grammar

```
template_version: <version>
```

3.2.1.3.5.3   Example

```
template_version: 2.0.17
```

3.2.1.3.5.4   Notes:

- Some service templates are designed to be referenced and reused by other service templates and have a lifecycle of their own. Therefore, in these cases, a template_version value SHOULD be included and used in conjunction with a unique template_name value to enable lifecycle management of the service template and its contents.

3.2.1.3.6 description

This optional keyname provides a means to include single or multiline descriptions within a TOSCA template as a scalar string value.

3.2.1.3.6.1   Keyname

```
description
```

3.2.1.3.7 dsl_definitions

This optional keyname provides a section to define macros (e.g., YAML-style macros when using the TOSCA specification).

3.2.1.3.7.1   Keyname

```
dsl_definitions
```

3.2.1.3.7.2   Grammar

```
dsl_definitions:
   <dsl_definition_1>
   ...
   <dsl_definition_n>
```

3.2.1.3.7.3   Example

```
dsl_definitions:
   ubuntu_image_props: &ubuntu_image_props
      architecture: x86_64
      type: linux
      distribution: ubuntu
      os_version: 14.04

   redhat_image_props: &redhat_image_props
      architecture: x86_64
      type: linux
      distribution: rhel
      os_version: 6.6
```

3.2.1.3.8 repositories

This optional keyname provides a section to define external repositories which may contain artifacts or other TOSCA Service Templates which might be referenced or imported by the TOSCA Service Template definition.

3.2.1.3.8.1 Keyname

```
repositories
```

3.2.1.3.8.2 Grammar

```
repositories:
   <repository_definition_1>
   ...
   <repository_definition_n>
```

3.2.1.3.8.3 Example

```
repositories:
  my_project_artifact_repo:
    description: development repository for TAR archives and Bash scripts
    url: http://mycompany.com/repository/myproject/
```

3.2.1.3.9 imports

This optional keyname provides a way to import a _block sequence_ of one or more TOSCA Definitions documents. TOSCA Definitions documents can contain reusable TOSCA type definitions (e.g., Node Types, Relationship Types, Artifact Types, etc.) defined by other authors. This mechanism provides an effective way for companies and organizations to define normative types and/or describe their software applications for reuse in other TOSCA Service Templates.

3.2.1.3.9.1 Keyname

```
imports
```

3.2.1.3.9.2 Grammar

```
imports:
   - <import_definition_1>
   - ...
   - <import_definition_n>
```

3.2.1.3.9.3 Example

```
# An example import of definitions files from a location relative to the
# file location of the service template declaring the import.
imports:
  - some_definitions: relative_path/my_defns/my_typesdefs_1.yaml
  - file: my_defns/my_typesdefs_n.yaml
    repository: my_company_repo
    namespace_prefix: mycompany
```

3.2.1.3.10 artifact_types

This optional keyname lists the Artifact Types that are defined by this Service Template.

3.2.1.3.10.1 Keyname

```
artifact_types
```

3.2.1.3.10.2 Grammar

```
artifact_types:
  <artifact_type_defn_1>
  ...
  <artifact type_defn_n>
```

3.2.1.3.10.3 Example

```
artifact_types:
  mycompany.artifacttypes.myFileType:
    derived_from: tosca.artifacts.File
```

3.2.1.3.11 data_types

This optional keyname provides a section to define new data types in TOSCA.

3.2.1.3.11.1 Keyname

```
data_types
```

3.2.1.3.11.2 Grammar

```
data_types:
  <tosca_datatype_def_1>
  ...
  <tosca_datatype_def_n>
```

3.2.1.3.11.3 Example

```
data_types:
  # A complex datatype definition
  simple_contactinfo_type:
    properties:
      name:
        type: string
      email:
        type: string
      phone:
        type: string

  # datatype definition derived from an existing type
  full_contact_info:
    derived_from: simple_contact_info
    properties:
      street_address:
        type: string
      city:
        type: string
      state:
        type: string
      postalcode:
        type: string
```

3.2.1.3.12 capability_types

This optional keyname lists the Capability Types that provide the reusable type definitions that can be used to describe features Node Templates or Node Types can declare they support.

3.2.1.3.12.1 Keyname

```
capability_types
```

3.2.1.3.12.2 Grammar

```
capability_types:
  <capability_type_defn_1>
  ...
  <capability type_defn_n>
```

3.2.1.3.12.3 Example

capability_types:

```
mycompany.mytypes.myCustomEndpoint:
  derived_from: tosca.capabilities.Endpoint
  properties:
    # more details ...

mycompany.mytypes.myCustomFeature:
  derived_from: tosca.capabilities.Feature
  properties:
    # more details ...
```

3.2.1.3.13 interface_types

This optional keyname lists the Interface Types that provide the reusable type definitions that can be used to describe operations for on TOSCA entities such as Relationship Types and Node Types.

3.2.1.3.13.1 Keyname

```
interface_types
```

3.2.1.3.13.2 Grammar

```
interface_types:
  <interface_type_defn_1>
  ...
  <interface type_defn_n>
```

3.2.1.3.13.3 Example

```
interface_types:
  mycompany.interfaces.service.Signal:
    signal_begin_receive:
      description: Operation to signal start of some message processing.
    signal_end_receive:
      description: Operation to signal end of some message processed.
```

3.2.1.3.14 relationship_types

This optional keyname lists the Relationship Types that provide the reusable type definitions that can be used to describe dependent relationships between Node Templates or Node Types.

3.2.1.3.14.1 Keyname

```
relationship_types
```

3.2.1.3.14.2 Grammar

```
relationship_types:
  <relationship_type_defn_1>
  ...
  <relationship type_defn_n>
```

3.2.1.3.14.3 Example

```
relationship_types:
  mycompany.mytypes.myCustomClientServerType:
    derived_from: tosca.relationships.HostedOn
    properties:
      # more details ...

  mycompany.mytypes.myCustomConnectionType:
    derived_from: tosca.relationships.ConnectsTo
    properties:
      # more details ...
```

3.2.1.3.15 node_types

This optional keyname lists the Node Types that provide the reusable type definitions for software components that Node Templates can be based upon.

3.2.1.3.15.1 Keyname

```
node_types
```

3.2.1.3.15.2 Grammar

```
node_types:
  <node_type_defn_1>
  ...
  <node_type_defn_n>
```

3.2.1.3.15.3 Example

```
node_types:
  my_webapp_node_type:
    derived_from: WebApplication
    properties:
      my_port:
        type: integer

  my_database_node_type:
    derived_from: Database
    capabilities:
      mytypes.myfeatures.transactSQL
```

3.2.1.3.15.4 Notes

- The node types that are part of the node_types block can be mapped to the NodeType definitions as described by the TOSCA v1.0 specification.

3.2.1.3.16 group_types

This optional keyname lists the Group Types that are defined by this Service Template.

3.2.1.3.16.1 Keyname

```
group_types
```

3.2.1.3.16.2 Grammar

```
group_types:
  <group_type_defn_1>
  ...
  <group type_defn_n>
```

3.2.1.3.16.3 Example

```
group_types:
  mycompany.mytypes.myScalingGroup:
    derived_from: tosca.groups.Root
```

3.2.1.3.17 policy_types

This optional keyname lists the Policy Types that are defined by this Service Template.

3.2.1.3.17.1 Keyname

```
policy_types
```

3.2.1.3.17.2 Grammar

```
policy_types:
  <policy_type_defn_1>
  ...
  <policy type_defn_n>
```

3.2.1.3.17.3 Example

```
policy_types:
  mycompany.mytypes.myScalingPolicy:
    derived_from: tosca.policies.Scaling
```

3.2.2 Version and Namespace

3.2.2.1 TOSCA Namespace URI and alias

The following TOSCA Namespace URI alias and TOSCA Namespace Alias are reserved values which SHALL be used when identifying the TOSCA Version 2.0 specification.

Namespace Alias	Namespace URI	Specification Description
tosca_2_0	http://docs.oasis-open.org/tosca/ns/2.0	The TOSCA v2.0 (YAML) target namespace and namespace alias.

3.2.2.1.1 TOSCA Namespace prefix

The following TOSCA Namespace prefix is a reserved value and SHALL be used to reference the default TOSCA Namespace URI as declared in TOSCA Service Templates.

Namespace Prefix	Specification Description
tosca	The reserved TOSCA Specification prefix that can be associated with the default TOSCA Namespace URI

3.2.2.1.2 TOSCA Namespacing in TOSCA Service Templates

In the TOSCA , TOSCA Service Templates MUST always have, as the first line of YAML, the keyname "tosca_definitions_version" with an associated TOSCA Namespace Alias value. This single line accomplishes the following:

Establishes the TOSCA Specification version whose grammar MUST be used to parse and interpret the contents for the remainder of the TOSCA Service Template.

Establishes the default TOSCA Namespace URI and Namespace Prefix for all types found in the document that are not explicitly namespaced.

Automatically imports (without the use of an explicit import statement) the normative type definitions (e.g., Node, Relationship, Capability, Artifact, etc.) that are associated with the TOSCA Specification the TOSCA Namespace Alias value identifies.

Associates the TOSCA Namespace URI and Namespace Prefix to the automatically imported TOSCA type definitions.

3.2.2.1.3 Rules to avoid namespace collisions

TOSCA s allows template authors to declare their own types and templates and assign them simple names with no apparent namespaces. Since TOSCA Service Templates can import other service templates to introduce new types and topologies of templates that can be used to provide concrete implementations (or substitute) for abstract nodes. Rules are needed so that TOSCA Orchestrators know how to avoid collisions and apply their own namespaces when import and nesting occur.

3.2.2.1.3.1   Additional Requirements

- The URI value "http://docs.oasis-open.org/tosca", as well as all (path) extensions to it, SHALL be reserved for TOSCA approved specifications and work. That means Service Templates that do not originate from a TOSCA approved work product MUST NOT use it, in any form, when declaring a (default) Namespace.

- Since TOSCA Service Templates can import (or substitute in) other Service Templates, TOSCA Orchestrators and tooling will encounter the "tosca_definitions_version" statement for each imported template. In these cases, the following additional requirements apply:

  – Imported type definitions with the same Namespace URI, local name and version SHALL be equivalent.

  – If different values of the "tosca_definitions_version" are encountered, their corresponding type definitions MUST be uniquely identifiable using their corresponding Namespace URI using a different Namespace prefix.

- Duplicate local names (i.e., within the same Service Template SHALL be considered an error. These include, but are not limited to duplicate names found for the following definitions:

- – Repositories (repositories)

  - – Data Types (data_types)

  - – Node Types (node_types)

  - – Relationship Types (relationship_types)

  - – Capability Types (capability_types)

  - – Artifact Types (artifact_types)

  - – Interface Types (interface_types)

- Duplicate Template names within a Service Template's Topology Template SHALL be considered an error. These include, but are not limited to duplicate names found for the following template types:

  - – Node Templates (node_templates)

  - – Relationship Templates (relationship_templates)

  - – Inputs (inputs)

  - – Outputs (outputs)

- Duplicate names for the following keynames within Types or Templates SHALL be considered an error. These include, but are not limited to duplicate names found for the following keynames:

  - – Properties (properties)

  - – Attributes (attributes)

  - – Artifacts (artifacts)

  - – Requirements (requirements)

  - – Capabilities (capabilities)

  - – Interfaces (interfaces)

  - – Policies (policies)

  - – Groups (groups)

3.2.2.2 Using Namespaces

As of TOSCA version 1.2, Service template authors may declare a namespace within a Service Template that will be used as the default namespace for any types (e.g., Node Type, Relationship Type, Data Type, etc.) defined within the same Service template.

Specifically, a Service Template's namespace declaration's URI will be used to form a unique, fully qualified Type name when combined with the locally defined, unqualified name of any Type in the same Service Template. The resulting, fully qualified Type name will be used by TOSCA Orchestrators, Processors and tooling when that Service Template was imported into another Service Template to avoid Type name collision.

If a default namespace for the Service Template is declared, then it should be declared immediately after the "tosca_definitions_version" declaration, to ensure that the namespace is clearly visible.

3.2.2.2.1 Example – Importing a Service Template and Namespaces

For example, let say we have two Service Templates, A and B, both of which define Types and a Namespace. Service Template B contains a Node Type definition for "MyNode" and declares its (default) Namespace to be "http://companyB.com/service/namespace/":

Service Template B

```
tosca_definitions_version: tosca_simple_yaml_1_2
description: Service Template B
namespace: http://companyB.com/service/namespace/

node_types:
  MyNode:
    derived_from: SoftwareComponent
    properties:
      # omitted here for brevity
    capabilities:
      # omitted here for brevity
```

Service Template A has its own, completely different, Node Type definition also named "MyNode".

Service Template A

```
tosca_definitions_version: tosca_simple_yaml_1_2
description: Service Template A
namespace: http://companyA.com/product/ns/

imports:
  - file: csar/templates/ServiceTemplateB.yaml
    namespace_prefix: templateB

node_types:
  MyNode:
    derived_from: Root
    properties:
      # omitted here for brevity
    capabilities:
      # omitted here for brevity
```

As you can see, Service Template A also "imports" Service Template B (i.e., "ServiceTemplateB.yaml")
bringing in its Type defintions to the global namespace using the Namespace URI declared in Service
Template B to fully qualify all of its imported types.

In addition, the import includes a "namespace_prefix" value (i.e., "templateB"), that can be used to qualify
and disambiguate any Type reference from from Service Template B within Service Template A. This
prefix is effectively the local alias for the corresponding Namespace URI declared within Service
Template B (i.e., "http://companyB.com/service/namespace/").

To illustrate conceptually what a TOSCA Orchestrator, for example, would track for their global
namespace upon processing Service Template A (and by import Service Template B) would be a list of
global Namespace URIs and their associated Namespace prefixes, as well as a list of fully qualified Type
names that comprises the overall global namespace.

3.2.2.2.1.1 Conceptual Global Namespace URI and Namespace Prefix tracking

Entry#	Namespace URI	Namespace Prefix	Added by Key (Source file)
1	http://docs.oasis-open.org/tosca/ns/2.0	tosca	• tosca_definitions_version: - *from Service Template A*
2	http://companyA.com/product/ns/	\<None\>	• namespace: - *from Service Template A*

3	http://companyB.com/service/namespace/	templateB	• namespace:
			  - *from Service Template B*
			• namespace_prefix:
			  - *from Service Template A*, during import

In the above table,

- **Entry 1**: is an entry for the default TOSCA namespace, which is required to exist for it to be a valid Service template. It is established by the "tosca_definitions_version" key's value. By default, it also gets assigned the "tosca" Namespace prefix.
- **Entry 2**: is the entry for the local default namespace for Service Template A as declared by the "namespace" key.
    - *Note that no Namespace prefix is needed; any locally defined types that are not qualified (i.e., not a full URI or using a Namespace Prefix) will default to this namespace if not found first in the TOSCA namespace.*
- **Entry 3**: is the entry for default Namespace URI for any type imported from Service Template B. The author of Service Template A has assigned the local Namespace Prefix "templateB" that can be used to qualify reference to any Type from Service Template B.

As per TOSCA specification, any Type, that is not qualified with the 'tosca' prefix or full URI name, should be first resolved by its unqualified name within the TOSCA namespace. If it not found there, then it may be resolved within the local Service Template's default namespace.

3.2.2.2.1.2   Conceptual Global Namespace and Type tracking

Entry#	Namespace URI	Unqualified Full Name	Unqualified Short Name	Type Classification
1	http://docs.oasis-open.org/tosca/ns/2.0	tosca.nodes.Compute	Compute	node
2	http://docs.oasis-open.org/tosca/ns/2.0	tosca.nodes.SoftwareComponent	SoftwareComponent	
3	http://docs.oasis-open.org/tosca/ns/2.0	tosca.relationships.ConnectsTo	ConnectsTo	relationship
…	…			
100	http://companyA.com/product/ns/	N/A	MyNode	node
…	…			
200	http://companyB.com/service/namespace/	N/A	MyNode	node
…	…			

In the above table,

- **Entry 1**: is an example of one of the TOSCA standard Node Types (i.e., "Compute") that is brought into the global namespace via the "tosca_definitions_version" key.
    - It also has two forms, full and short that are unique to TOSCA types for historical reasons. Reference to a TOSCA type by either its unqualified short or full names is viewed as equivalent as a reference to the same fully qualified Type name (i.e., its full URI).
    - In this example, use of either "tosca.nodes.Compute" or "Compute" (i.e., an unqualified full and short name Type) in a Service Template will be treated as its fully qualified URI equivalent of:
        - "http://docs.oasis-open.org/tosca/ns/2.0/tosca.nodes.Compute".
- **Entry 2**: is an example of a standard TOSCA Relationship Type
- **Entry 100**: contains the unique Type identifer for the Node Type "MyNode" from Service Template A.
- **Entry 200**: contains the unique Type identifer for the Node Type "MyNode" from Service Template B.

As you can see, although both templates defined a NodeType with an unqualified name of "MyNode", the TOSCA Orchestrator, processor or tool tracks them by their unique fully qualified Type Name (URI).

The classification column is included as an example on how to logically differentiate a "Compute" Node Type and "Compute" capability type if the table will be used to "search" for a match based upon context in a Service Template.

For example, if the short name "Compute" were used in a template on a Requirements clause, then the matching type will not be the Compute Node Type, but instead the Compute Capability Type based upon the Requirement clause being the context for Type reference.

3.2.3 Imports and References

3.2.3.1 Import definition

An import definition is used within a TOSCA Service Template to locate and uniquely name another TOSCA Service Template file which has type and template definitions to be imported (included) and referenced within another Service Template.

3.2.3.1.1 Keynames

The following is the list of recognized keynames for a TOSCA import definition:

Keyname	Required	Type	Constraints	Description
file	yes	string	None	The required symbolic name for the imported file.
repository	no	string	None	The optional symbolic name of the repository definition where the imported file can be found as a string.
namespace_prefix	no	string	None	The optional namespace prefix (alias) that will be used to indicate the namespace_uri when forming a qualified name (i.e., qname) when referencing type definitions from the imported file.
namespace_uri	no	string	Deprecated	The optional, deprecated namespace URI to that will be applied to type definitions found within the imported file as a string.

3.2.3.1.2 Grammar

Import definitions have one the following grammars:

3.2.3.1.2.1 Single-line grammar:

```
imports:
  - <URI_1>
  - <URI_2>
```

3.2.3.1.2.2 Multi-line grammar

```
imports:
  - file: <file_URI>
    repository: <repository_name>
    namespace_uri: <definition_namespace_uri>  # deprecated
    namespace_prefix: <definition_namespace_prefix>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- file_uri: contains the required name (i.e., URI) of the file to be imported as a string.
- repository_name: represents the optional symbolic name of the repository definition where the imported file can be found as a string.
- namespace_uri: represents the optional namespace URI to that will be applied to type definitions found within the imported file as a string.
- namespace_prefix: represents the optional namespace prefix (alias) that will be used to indicate the default namespace as declared in the imported Service Template when forming a qualified name (i.e., qname) when referencing type definitions from the imported file as a string.

3.2.3.1.2.3　Requirements

- The "file" keyname's vlue MAY be an approved TOSCA Namespace alias.
- The namespace prefix "tosca" Is reserved and SHALL NOT be used to as a value for "namespace_prefix" on import.
- The imports key "namespace_uri" is now deprecated. It was intended to be able to define a default namespace for any types that were defined within the Service Template being imported; however, with version 1.2, Service Templates MAY now declare their own default Namespace which SHALL be used in place of this key's value.
  - Please note that TOSCA Orchestrators and Processors MAY still use the"namespace_uri" value if provided, if the imported Service Template has no declared default Namespace value. Regardless it is up to the TOSCA Orchestrator or Processor to resolve Namespace collisions caused by imports as they see fit, for example, they may treat it as an error or dynamically generate a unique namepspace themselves on import.

3.2.3.1.2.4　Import URI processing requirements

TOSCA Orchestrators, Processors and tooling SHOULD treat the <file_URI> of an import as follows:

- **URI**: If the <file_URI> is a known namespace URI (identifier), such as a well-known URI defined by a TOSCA specification, then it SHOULD cause the corresponding Type defintions to be imported.
  - This implies that there may or may not be an actual Service Template, perhaps it is a known set Types identified by the well-known URI.
  - This also implies that internet access is NOT needed to import.
- **Alias** – If the <file_URI> is a reserved TOSCA Namespace alias, then it SHOULD cause the corresponding Type defintions to be imported, using the associated full, Namespace URI to uniquely identify the imported types.
- **URL** - If the <file_URI> is a valid URL (i.e., network accessible as a remote resource) and the location contains a valid TOSCA Service Template, then it SHOULD cause the remote Service Template to be imported.
- **Relative path** - If the <file_URI> is a relative path URL, perhaps pointing to a Service Template located in the same CSAR file, then it SHOULD cause the locally accessible Service Template to be imported.
  - If the "repository" key is supplied, this could also mean relative to the repository's URL in a remote file system;
  - If the importing file located in a CSAR file, it should be treated as relative to the current document's location within a CSAR file's directory structure.
- Otherwise, the import SHOULD be considered a failure.

3.2.3.1.3 Example

The following represents how import definitions will be used for the imports keyname within a TOSCA Service Template:

```
imports:
  - path1/path2/some_defs.yaml
  - file: path1/path2/file2.yaml
    repository: my_service_catalog
    namespace_uri: http://mycompany.com/tosca/1.0/platform
    namespace_prefix: mycompany
```

3.2.3.2 Repository definition

A repository definition defines a named external repository which contains deployment and implementation artifacts that are referenced within the TOSCA Service Template.

3.2.3.2.1 Keynames

The following is the list of recognized keynames for a TOSCA repository definition:

Keyname	Required	Type	Constraints	Description
description	no	description	None	The optional description for the repository.
url	yes	string	None	The required URL or network address used to access the repository.
credential	no	Credential	None	The optional Credential used to authorize access to the repository.

3.2.3.2.2 Grammar

Repository definitions have one the following grammars:

3.2.3.2.2.1 Single-line grammar (no credential):

```
<repository_name>: <repository_address>
```

3.2.3.2.2.2 Multi-line grammar

```
<repository_name>:
  description: <repository_description>
  url: <repository_address>
  credential: <authorization_credential>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- repository_name: represents the required symbolic name of the repository as a string.
- repository_description: contains an optional description of the repository.
- repository_address: represents the required URL of the repository as a string.
- authorization_credential: represents the optional credentials (e.g., user ID and password) used to authorize access to the repository.

3.2.3.2.3 Example

The following represents a repository definition:

```
repositories:
  my_code_repo:
    description: My project's code repository in GitHub
    url: https://github.com/my-project/
```

3.2.4 Additional information definitions

3.2.4.1 Description definition

This optional element provides a means include single or multiline descriptions within a TOSCA template as a scalar string value.

3.2.4.1.1 Keyname

The following keyname is used to provide a description within the TOSCA specification:

```
description
```

3.2.4.1.2 Grammar

Description definitions have the following grammar:

```
description: <string>
```

3.2.4.1.3 Examples

Simple descriptions are treated as a single literal that includes the entire contents of the line that immediately follows the description key:

```
description: This is an example of a single line description (no folding).
```

The YAML "folded" style may also be used for multi-line descriptions which "folds" line breaks as space characters.

```
description: >
  This is an example of a multi-line description using YAML. It permits for
line
  breaks for easier readability...

  if needed.  However, (multiple) line breaks are folded into a single
space
  character when processed into a single string value.
```

3.2.4.1.4 Notes

- Use of "folded" style is discouraged for the YAML string type apart from when used with the description keyname.

3.2.4.2 Metadata

This optional element provides a means to include optional metadata as a map of strings.

3.2.4.2.1 Keyname

The following keyname is used to provide metadata within the TOSCA specification:

```
metadata
```

3.2.4.2.2 Grammar

Metadata definitions have the following grammar:

```
metadata:
  map of <string>
```

3.2.4.2.3 Examples

```
metadata:
  foo1: bar1
  foo2: bar2
```

3.2.4.2.4 Notes

- Data provided within metadata, wherever it appears, MAY be ignored by TOSCA Orchestrators and SHOULD NOT affect runtime behavior.

3.2.4.3 DSL Definitions

TBD.

3.2.5 Type definitions

TOSCA provides a type system to describe possible building blocks to construct a topology template (i.e. for the nodes, relationship, group and policy templates, and the data, capabilities, interfaces, and artifacts used in the node and relationship templates). TOSCA types are reusable TOSCA entities and are defined in their specific sections in the service template, see Section 3.2.1 Service Template definition.

Next, in Section 3.2.5.2 Common keynames in type definitions we present the definitions of common keynames that are used by all TOSCA types. Type-specific definitions for the different TOSCA type entities are presented further in the document:

- Node Type in Section 3.3.1 Node Type.
- Relationship Type in Section 3.3.3 Relationship Type.
- Interface Type in Section 3.3.6.1 Interface Type.
- Capability Type in Section 3.3.5.1 Capability Type.
- Requirement Type in Section 3.3.5.4 Requirement Type.
- Data Type in Section 3.4.2 Data Type.
- Artifact Type in Section 3.3.7.1 Artifact Type.
- Group Type in Section 3.6.1 Group Type.
- Policy Type in Section 3.6.3 Policy Type.

3.2.5.1 General derivation and refinement rules

To simplify type creation and to promote type extensibility TOSCA allows the definition of a new type (the derived type) based on another type (the parent type). The derivation process can be applied recursively, where a type may be derived from a long list of ancestor types (the parent, the parent of the parent, etc).

Unless specifically stated in the derivation rules, when deriving new types from parent types the keyname definitions are inherited from the parent type. Moreover, the inherited definitions may be refined according to the derivation rules of that particular type entity.

For definitions that are not inherited, a new definition **MUST** be provided (if the keyname is required) or **MAY** be provided (if the keyname is not required). If not provided, the keyname remains undefined. For definitions that are inherited, a refinement of the inherited definition is not mandatory even for required keynames (since it has been inherited). A definition refinement that is exactly the same as the definition in the parent type does not change in any way the inherited definition. While unnecessary, it is not wrong.

The following are some generic derivation rules used during type derivation (the specific rules of each TOSCA type entity are presented in their respective sections):

- If not refined, usually a keyname/entity definition, is inherited unchanged from the parent type, unless explicitly specified in the rules that it is "not inherited".
- New entities (such as properties, attributes, capabilities, requirements, interfaces, operations, notification, parameters) may be added during derivation.
- Already defined entities that have a type may be redefined to have a type derived from the original type.

- New constraints are added to already defined keynames/entities (i.e. the defined constraints do not replace the constraints defined in the parent type but are added to them).
- Some definitions must be totally flexible, so they will overwrite the definition in the parent type.
- Some definitions must not be changed at all once defined (i.e. they represent some sort of "signature").

3.2.5.2 Common keynames in type definitions

The following keynames are used by all TOSCA type entities in the same way. This section serves to define them at once.

3.2.5.2.1 Keynames

The following is the list of recognized keynames used by all TOSCA type definitions:

Keyname	Required	Type	Description
derived_from	no	string	An optional parent type name from which this type derives.
version	no	version	An optional version for the type definition.
metadata	no	map of string	Defines a section used to declare additional metadata information.
description	no	description	An optional description for the type.

3.2.5.2.2 Grammar

The common keynames in type definitions have the following grammar:

```
<type_name>:
  derived_from: <parent_type_name>
  version: <version_number>
  metadata:
    <metadata_map>
  description: <type_description>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- parent_type_name: represents the optional parent type name.
- version_number: represents the optional TOSCA version number for the type.
- entity_description: represents the optional description string for the type.
- metadata_map: represents the optional metadata map of string.

3.2.5.2.3 Derivation rules

During type derivation the common keyname definitions use the following rules:

- derived_from: obviously, the definition is not inherited from the parent type. If not defined, it remains undefined and this type does not derive from another type. If defined, then this type derives from another type, and all its keyname definitions must respect the derivation rules of the type entity.
- version: the definition is not inherited from the parent type. If undefined, it remains undefined.
- metadata: the definition is not inherited from the parent type. If undefined, it remains undefined.
- description: the definition is not inherited from the parent type. If undefined, it remains undefined.

3.2.6 Topology Template definition

This section defines the topology template of a cloud application. The main ingredients of the topology template are node templates representing components of the application and relationship templates representing links between the components. These elements are defined in the nested node_templates section and the nested relationship_templates sections, respectively. Furthermore, a topology template allows for defining input parameters, output parameters as well as grouping of node templates.

3.2.6.1 Keynames

The following is the list of recognized keynames for a TOSCA Topology Template:

Keyname	Required	Type	Description
description	no	description	The optional description for the Topology Template.
inputs	no	map of parameter definitions	An optional map of input parameters (i.e., as parameter definitions) for the Topology Template.
node_templates	no	map of node templates	An optional map of node template definitions for the Topology Template.
relationship_templates	no	map of relationship templates	An optional map of relationship templates for the Topology Template.
groups	no	map of group definitions	An optional map of Group definitions whose members are node templates defined within this same Topology Template.
policies	no	list of policy definitions	An optional list of Policy definitions for the Topology Template.
outputs	no	map of parameter definitions	An optional map of output parameters (i.e., as parameter definitions) for the Topology Template.
substitution_mappings	no	substitution_mapping	An optional declaration that exports the topology template as an implementation of a Node type. This also includes the mappings between the external Node Types capabilities and requirements to existing implementations of those capabilities and requirements on Node templates declared within the topology template.
workflows	no	map of imperative workflow definitions	An optional map of imperative workflow definition for the Topology Template.

3.2.6.2 Grammar

The overall grammar of the topology_template section is shown below.–Detailed grammar definitions of the each sub-sections are provided in subsequent subsections.

```
topology_template:
  description: <template_description>
```

```
        inputs: <input_parameters>
        outputs: <output_parameters>
        node_templates: <node_templates>
        relationship_templates: <relationship_templates>
        groups: <group_definitions>
        policies:
          - <policy_definition_list>
        workflows: <workflows>
        # Optional declaration that exports the Topology Template
        # as an implementation of a Node Type.
        substitution_mappings:
          <substitution_mappings>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- template_description: represents the optional description string for Topology Template.
- input_parameters: represents the optional map of input parameter definitions for the Topology Template.
- output_parameters: represents the optional map of output parameter definitions for the Topology Template.
- group_definitions: represents the optional map of group definitions whose members are node templates that also are defined within this Topology Template.
- policy_definition_list: represents the optional list of sequenced policy definitions for the Topology Template.
- workflows: represents the optional map of imperative workflow definitions for the Topology Template.
- node_templates: represents the optional map of node template definitions for the Topology Template.
- relationship_templates: represents the optional map of relationship templates for the Topology Template.
- node_type_name: represents the optional name of a Node Type that the Topology Template implements as part of the substitution_mappings.
- map_of_capability_mappings_to_expose: represents the mappings that expose internal capabilities from node templates (within the topology template) as capabilities of the Node Type definition that is declared as part of the substitution_mappings.
- map_of_requirement_mappings_to_expose: represents the mappings of link requirements of the Node Type definition that is declared as part of the substitution_mappings to internal requirements implementations within node templates (declared within the topology template).

More detailed explanations for each of the Topology Template grammar's keynames appears in the sections below.

3.2.6.2.1 inputs

The inputs section provides a means to define parameters using TOSCA parameter definitions, their allowed values via constraints and default values within a TOSCA template. Input parameters defined in the inputs section of a topology template can be mapped to properties of node templates or relationship templates within the same topology template and can thus be used for parameterizing the instantiation of the topology template.

This section defines topology template-level input parameter section.

- Inputs here would ideally be mapped to BoundaryDefinitions in TOSCA v1.0.
- Treat input parameters as fixed global variables (not settable within template)
- If not in input take default (nodes use default)

3.2.6.2.1.1   Grammar

The grammar of the inputs section is as follows:

```
inputs:
  <parameter_definitions>
```

3.2.6.2.1.2   Examples

This section provides a set of examples for the single elements of a topology template.

Simple inputs example without any constraints:

```
inputs:
  fooName:
    type: string
    description: Simple string typed parameter definition with no
constraints.
    default: bar
```

Example of inputs with constraints:

```
inputs:
  SiteName:
    type: string
    description: string typed parameter definition with constraints
    default: My Site
    constraints:
      - min_length: 9
```

3.2.6.2.2 node_templates

The node_templates section lists the Node Templates that describe the (software) components that are used to compose cloud applications.

3.2.6.2.2.1   grammar

The grammar of the node_templates section is a follows:

```
node_templates:
  <node_template_defn_1>
  ...
  <node_template_defn_n>
```

3.2.6.2.2.2   Example

Example of node_templates section:

```
node_templates:
  my_webapp_node_template:
    type: WebApplication

  my_database_node_template:
    type: Database
```

3.2.6.2.3 relationship_templates

The relationship_templates section lists the Relationship Templates that describe the relations between components that are used to compose cloud applications.

Note that in TOSCA, the explicit definition of relationship templates as it was required in TOSCA v1.0 is optional, since relationships between nodes get implicitly defined by referencing other node templates in the requirements sections of node templates.

3.2.6.2.3.1 Grammar

The grammar of the relationship_templates section is as follows:

```
relationship_templates:
  <relationship_template_defn_1>
  ...
  <relationship_template_defn_n>
```

3.2.6.2.3.2 Example

Example of relationship_templates section:

```
relationship_templates:
  my_connectsto_relationship:
    type: tosca.relationships.ConnectsTo
    interfaces:
      Configure:
        inputs:
          speed: { get_attribute: [ SOURCE, connect_speed ] }
```

3.2.6.2.4 outputs

The outputs section provides a means to define the output parameters that are available from a TOSCA service template. It allows for exposing attributes of node templates or relationship templates within the containing topology_template to users of a service.

3.2.6.2.4.1 Grammar

The grammar of the outputs section is as follows:

```
outputs:
  <parameter_definitions>
```

3.2.6.2.4.2 Example

Example of the outputs section:

```
outputs:
  server_address:
    description: The first private IP address for the provisioned server.
    value: { get_attribute: [ HOST, networks, private, addresses, 0 ] }
```

3.2.6.2.5 groups

The groups section allows for grouping one or more node templates within a TOSCA Service Template and for assigning special attributes like policies to the group.

3.2.6.2.5.1 Grammar

The grammar of the groups section is as follows:

```
groups:
  <group_defn_1>
  ...
  <group_defn_n>
```

3.2.6.2.5.2   Example

The following example shows the definition of three Compute nodes in the node_templates section of a topology_template as well as the grouping of two of the Compute nodes in a group server_group_1.

```
node_templates:
  server1:
    type: tosca.nodes.Compute
    # more details ...

  server2:
    type: tosca.nodes.Compute
    # more details ...

  server3:
    type: tosca.nodes.Compute
    # more details ...

groups:
  # server2 and server3 are part of the same group
  server_group_1:
    type: tosca.groups.Root
    members: [ server2, server3 ]
```

3.2.6.2.6 policies

The policies section allows for declaring policies that can be applied to entities in the topology template.

3.2.6.2.6.1   Grammar

The grammar of the policies section is as follows:

```
policies:
  - <policy_defn_1>
  - ...
  - <policy_defn_n>
```

3.2.6.2.6.2   Example

The following example shows the definition of a placement policy.

```
policies:
  - my_placement_policy:
      type: mycompany.mytypes.policy.placement
```

3.2.6.2.7 substitution_mapping

3.2.6.2.7.1   requirement_mapping

The grammar of a requirement_mapping is as follows:

```
<requirement_name>: [ <node_template_name>,
<node_template_requirement_name> ]
```

The multi-line grammar is as follows :

```
<requirement_name>:
  mapping: [ <node_template_name>, <node_template_capability_name> ]
  properties:
    <property_name>: <property_value>
```

- requirement_name: represents the name of the requirement as it appears in the Node Type definition for the Node Type (name) that is declared as the value for on the substitution_mappings' "node_type" key.
- node_template_name: represents a valid name of a Node Template definition (within the same topology_template declaration as the substitution_mapping is declared).
- node_template_requirement_name: represents a valid name of a requirement definition within the <node_template_name> declared in this mapping.

3.2.6.2.7.2 Example

The following example shows the definition of a placement policy.

```
topology_template:

inputs:
   cpus:
     type: integer
     constraints:
       less_than: 2 # OR use "defaults" key

substitution_mappings:
    node_type: MyService
    properties:  # Do not care if running or matching (e.g., Compute node)
      # get from outside?  Get from contsraint?
      num_cpus: cpus # Implied "PUSH"
      # get from some node in the topology…
      num_cpus: [ <node>, <cap>, <property> ]
      # 1) Running
      architecture:
        # a) Explicit
        value: { get_property: [some_service, architecture] }
        # b) implicit
        value: [ some_service, <req | cap name>, <property name>
architecture ]
        default: "amd"
        # c) INPUT mapping?
        ???
      # 2) Catalog (Matching)
      architecture:
         contraints: equals: "x86"

    capabilities:
      bar: [ some_service, bar ]
    requirements:
      foo: [ some_service, foo ]

  node_templates:
    some_service:
      type: MyService
      properties:
        rate: 100
      capabilities:
        bar:
           ...
      requirements:
        - foo:
             ...
```

3.2.6.2.8 Notes

- The parameters (properties) that are part of the inputs block can be mapped to PropertyMappings provided as part of BoundaryDefinitions as described by the TOSCA v1.0 specification.
- The node templates that are part of the node_templates block can be mapped to the NodeTemplate definitions provided as part of TopologyTemplate of a ServiceTemplate as described by the TOSCA v1.0 specification.
- The relationship templates that are part of the relationship_templates block can be mapped to the RelationshipTemplate definitions provided as part of TopologyTemplate of a ServiceTemplate as described by the TOSCA v1.0 specification.
- The output parameters that are part of the outputs section of a topology template can be mapped to PropertyMappings provided as part of BoundaryDefinitions as described by the TOSCA v1.0 specification.
  - Note, however, that TOSCA v1.0 does not define a direction (input vs. output) for those mappings, i.e. TOSCA v1.0 PropertyMappings are underspecified in that respect and TOSCA 's inputs and outputs provide a more concrete definition of input and output parameters.

3.3 Nodes and Relationships

3.3.1 Node Type

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 and attributes, the capabilities and requirements of the node as well as its supported interfaces and the artifacts it uses.

3.3.1.1 Keynames

The Node Type is a TOSCA type entity and has the common keynames listed Section 3.2.5.2 Common keynames in type definitions. In addition, the Node Type has the following recognized keynames:

Keyname	Required	Type	Description
properties	no	map of property definitions	An optional map of property definitions for the Node Type.
attributes	no	map of attribute definitions	An optional map of attribute definitions for the Node Type.
capabilities	no	map of capability definitions	An optional map of capability definitions for the Node Type.
requirements	no	list of requirement definitions	An optional list of requirement definitions for the Node Type.
interfaces	no	map of interface definitions	An optional map of interface definitions supported by the Node Type.
artifacts	no	map of artifact definitions	An optional map of artifact definitions for the Node Type.

3.3.1.2 Grammar

Node Types have following grammar:

```
<node_type_name>:
  derived_from: <parent_node_type_name>
  version: <version_number>
  metadata:
    <map of string>
  description: <node_type_description>
  properties:
    <property_definitions>
  attributes:
    <attribute_definitions>
  capabilities:
    <capability_definitions>
  requirements:
    - <requirement_definitions>
  interfaces:
    <interface_definitions>
  artifacts:
    <artifact_definitions>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- node_type_name: represents the required symbolic name of the Node Type being declared.
- parent_node_type_name: represents the name (string) of the Node Type this Node Type definition derives from (i.e. its parent type).
- version_number: represents the optional TOSCA version number for the Node Type.
- node_type_description: represents the optional description string for the corresponding node_type_name.
- property_definitions: represents the optional map of property definitions for the Node Type.
- attribute_definitions: represents the optional map of attribute definitions for the Node Type.
- capability_definitions: represents the optional map of capability definitions for the Node Type.
- requirement_definitions: represents the optional list of requirement definitions for the Node Type.
- interface_definitions: represents the optional map of one or more interface definitions supported by the Node Type.
- artifact_definitions: represents the optional map of artifact definitions for the Node Type.

3.3.1.3 Derivation rules

During Node Type derivation the keyname definitions follow these rules:

- properties: existing property definitions may be refined; new property definitions may be added.
- attributes: existing attribute definitions may be refined; new attribute definitions may be added.
- capabilities: existing capability definitions may be refined; new capability definitions may be added.
- requirements: existing requirement definitions may be refined; new requirement definitions may be added.
- interfaces: existing interface definitions may be refined; new interface definitions may be added.
- artifacts: existing artifact definitions may not be changed; new artifact definitions may be added.
  - an artifact is created for a specific purpose, if it cannot meet its purpose in a derived type then a new artifact should be defined and used.

3.3.1.4 Additional Requirements

- Requirements are intentionally expressed as a list of TOSCA Requirement definitions which **SHOULD** be resolved (processed) in sequence by TOSCA Orchestrators.

3.3.1.5 Example

```
my_company.my_types.my_app_node_type:
  derived_from: tosca.nodes.SoftwareComponent
  description: My company's custom applicaton
  properties:
    my_app_password:
      type: string
      description: application password
      constraints:
        - min_length: 6
        - max_length: 10
  attributes:
    my_app_port:
      type: integer
      description: application port number
  requirements:
    - some_database:
        capability: EndPoint.Database
        node: Database
        relationship: ConnectsTo
```

3.3.2 Node Template

A Node Template specifies the occurrence of a manageable component as part of an application's topology model which is defined in a TOSCA Service Template. A Node Template is an instance of a specified Node Type and can provide customized properties, constraints, relationships or interfaces which complement and change the defaults provided by its Node Type.

3.3.2.1 Keynames

The following is the list of recognized keynames for a TOSCA Node Template definition:

Keyname	Required	Type	Description
type	yes	string	The required name of the Node Type the Node Template is based upon.
description	no	description	An optional description for the Node Template.
metadata	no	map of string	Defines a section used to declare additional metadata information.
directives	no	list of string	An optional list of directive values to provide processing instructions to orchestrators and tooling.
properties	no	map of property assignments	An optional map of property value assignments for the Node Template.
attributes	no	map of attribute assignments	An optional map of attribute value assignments for the Node Template.
requirements	no	list of requirement assignments	An optional list of requirement assignments for the Node Template.

Keyname	Required	Type	Description
capabilities	no	map of capability assignments	An optional map of capability assignments for the Node Template.
interfaces	no	map of interface assignments	An optional map of interface assignments for the Node Template.
artifacts	no	map of artifact definitions	An optional map of artifact definitions for the Node Template.
node_filter	no	node filter	The optional filter definition that TOSCA orchestrators will use to select the correct target node.
copy	no	string	The optional (symbolic) name of another node template to copy into (all keynames and values) and use as a basis for this node template.

3.3.2.2 Grammar

```
<node_template_name>:
  type: <node_type_name>
  description: <node_template_description>
  directives: [<directives>]
  metadata:
    <map of string>
  properties:
    <property_assignments>
  attributes:
    <attribute_assignments>
  requirements:
    - <requirement_assignments>
  capabilities:
    <capability_assignments>
  interfaces:
    <interface_assignments>
  artifacts:
    <artifact_definitions>
  node_filter:
    <node_filter_definition>
  copy: <source_node_template_name>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- node_template_name: represents the required symbolic name of the Node Template being declared.
- node_type_name: represents the name of the Node Type the Node Template is based upon.
- node_template_description: represents the optional description string for Node Template.
- directives: represents the optional list of processing instruction keywords (as strings) for use by tooling and orchestrators.
- property_assignments: represents the optional map of property assignments for the Node Template that provide values for properties defined in its declared Node Type.
- attribute_assignments: represents the optional map of attribute assignments for the Node Template that provide values for attributes defined in its declared Node Type.
- requirement_assignments: represents the optional list of requirement assignments for the Node Template for requirement definitions provided in its declared Node Type.
- capability_assignments: represents the optional map of capability assignments for the Node Template for capability definitions provided in its declared Node Type.

- interface_assignments: represents the optional map of interface assignments for the Node Template interface definitions provided in its declared Node Type.
- artifact_definitions: represents the optional map of artifact definitions for the Node Template that augment those provided by its declared Node Type.
- node_filter_definition: represents the optional node filter TOSCA orchestrators will use for selecting a matching node template.
- source_node_template_name: represents the optional (symbolic) name of another node template to copy into (all keynames and values) and use as a basis for this node template.

3.3.2.3 Additional requirements

- The source node template provided as a value on the copy keyname **MUST NOT** itself use the copy keyname (i.e., it must itself be a complete node template description and not copied from another node template).

3.3.2.4 Example

```
node_templates:
  mysql:
    type: tosca.nodes.DBMS.MySQL
    properties:
      root_password: { get_input: my_mysql_rootpw }
      port: { get_input: my_mysql_port }
    requirements:
      - host: db_server
    interfaces:
      Standard:
        configure: scripts/my_own_configure.sh
```

3.3.3 Relationship Type

A Relationship Type is a reusable entity that defines the type of one or more relationships between Node Types or Node Templates.

3.3.3.1 Keynames

The Relationship Type is a TOSCA type entity and has the common keynames listed in Section 3.2.5.2 Common keynames in type definitions. In addition, the Relationship Type has the following recognized keynames:

Keyname	Required	Definition/Type	Description
properties	no	map of property definitions	An optional map of property definitions for the Relationship Type.
attributes	no	map of attribute definitions	An optional map of attribute definitions for the Relationship Type.
interfaces	no	map of interface definitions	An optional map of interface definitions supported by the Relationship Type.
valid_target_types	no	list of string	An optional list of one or more names of Capability Types that are valid targets for this relationship. If undefined, all Capability Types are valid target targets.

3.3.3.2 Grammar

Relationship Types have following grammar:

```
<relationship_type_name>:
  derived_from: <parent_relationship_type_name>
  version: <version_number>
  metadata:
    <map of string>
  description: <relationship_description>
  properties:
    <property_definitions>
  attributes:
    <attribute_definitions>
  interfaces:
    <interface_definitions>
  valid_target_types: [ <capability_type_names> ]
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- relationship_type_name: represents the required symbolic name of the Relationship Type being declared as a string.
- parent_relationship_type_name: represents the name (string) of the Relationship Type this Relationship Type definition derives from (i.e., its "parent" type).
- relationship_description: represents the optional description string for the corresponding relationship_type_name.
- version_number: represents the optional TOSCA version number for the Relationship Type.
- property_definitions: represents the optional map of property definitions for the Relationship Type.
- attribute_definitions: represents the optional map of attribute definitions for the Relationship Type.
- interface_definitions: represents the optional map of interface definitions supported by the Relationship Type.
- capability_type_names: represents the optional list of valid target Capability Types for the relationship; if undefined, the valid target types are not restricted at all (i.e. all Capability Types are valid).

3.3.3.3 Derivation rules

During Relationship Type derivation the keyname definitions follow these rules:

- properties: existing property definitions may be refined; new property definitions may be added.
- attributes: existing attribute definitions may be refined; new attribute definitions may be added.
- interfaces: existing interface definitions may be refined; new interface definitions may be added.
- valid_target_types: if valid_target_types is defined in the parent type, each element in this list must either be in the parent type list or derived from an element in the parent type list; if valid_target_types is not defined in the parent type then no restrictions are applied.

3.3.3.4 Examples

```
mycompanytypes.myrelationships.AppDependency:
  derived_from: tosca.relationships.DependsOn
  valid_target_types: [ mycompanytypes.mycapabilities.SomeAppCapability ]
```

3.3.4 Relationship Template

A Relationship Template specifies the occurrence of a manageable relationship between node templates as part of an application's topology model that is defined in a TOSCA Service Template. A Relationship template is an instance of a specified Relationship Type and can provide customized properties, constraints or operations which complement and change the defaults provided by its Relationship Type and its implementations.

3.3.4.1 Keynames

The following is the list of recognized keynames for a TOSCA Relationship Template definition:

Keyname	Required	Type	Description
type	yes	string	The required name of the Relationship Type the Relationship Template is based upon.
description	no	description	An optional description for the Relationship Template.
metadata	no	map of string	Defines a section used to declare additional metadata information.
properties	no	map of property assignments	An optional map of property assignments for the Relationship Template.
attributes	no	map of attribute assignments	An optional map of attribute assignments for the Relationship Template.
interfaces	no	map of interface assignments	An optional map of interface assignments for the relationship template.
copy	no	string	The optional (symbolic) name of another relationship template to copy into (all keynames and values) and use as a basis for this relationship template.

3.3.4.2 Grammar

```
<relationship_template_name>:
  type: <relationship_type_name>
  description: <relationship_type_description>
  metadata:
    <map of string>
  properties:
    <property_assignments>
  attributes:
    <attribute_assignments>
  interfaces:
    <interface_assignments>
  copy:
    <source_relationship_template_name>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- relationship_template_name: represents the required symbolic name of the Relationship Template being declared.
- relationship_type_name: represents the name of the Relationship Type the Relationship Template is based upon.
- relationship_template_description: represents the optional description string for the Relationship Template.

- property_assignments: represents the optional map of property assignments for the Relationship Template that provide values for properties defined in its declared Relationship Type.
- attribute_assignments: represents the optional map of attribute assignments for the Relationship Template that provide values for attributes defined in its declared Relationship Type.
- interface_assignments: represents the optional map of interface assignments for the Relationship Template for interface definitions provided by its declared Relationship Type.
- source_relationship_template_name: represents the optional (symbolic) name of another relationship template to copy into (all keynames and values) and use as a basis for this relationship template.

3.3.4.3 Additional requirements

- The source relationship template provided as a value on the copy keyname MUST NOT itself use the copy keyname (i.e., it must itself be a complete relationship template description and not copied from another relationship template).

3.3.4.4 Example

```
relationship_templates:
  storage_attachment:
    type: AttachesTo
    properties:
      location: /my_mount_point
```

3.3.5 Capabilities and Requirements

3.3.5.1 Capability Type

A Capability Type is a reusable entity that describes a kind of capability that a Node Type can declare to expose. Requirements (implicit or explicit) that are declared as part of one node can be matched to (i.e., fulfilled by) the Capabilities declared by another node.

3.3.5.1.1 Keynames

The Capability Type is a TOSCA type entity and has the common keynames listed in Section 3.2.5.2 Common keynames in type definitions. In addition, the Capability Type has the following recognized keynames:

Keyname	Required	Type	Description
properties	no	map of property definitions	An optional map of property definitions for the Capability Type.
attributes	no	map of attribute definitions	An optional map of attribute definitions for the Capability Type.
valid_source_types	no	list of string	An optional list of one or more valid names of Node Types that are supported as valid sources of any relationship established to the declared Capability Type. If undefined, all Node Types are valid sources.

3.3.5.1.2 Grammar

Capability Types have following grammar:

```
<capability_type_name>:
```

```
      derived_from: <parent_capability_type_name>
      version: <version_number>
      description: <capability_description>
      properties:
        <property_definitions>
      attributes:
        <attribute_definitions>
      valid_source_types: [ <node type_names> ]
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- capability_type_name: represents the required name of the Capability Type being declared as a string.
- parent_capability_type_name: represents the name of the Capability Type this Capability Type definition derives from (i.e., its "parent" type).
- version_number: represents the optional TOSCA version number for the Capability Type.
- capability_description: represents the optional description string for the Capability Type.
- property_definitions: represents the optional map of property definitions for the Capability Type.
- attribute_definitions: represents the optional map of attribute definitions for the Capability Type.
- node_type_names: represents the optional list of one or more names of Node Types that the Capability Type supports as valid sources for a successful relationship to be established to itself; if undefined, the valid source types are not restricted at all (i.e. all Node Types are valid)

3.3.5.1.3 Derivation rules

During Capability Type derivation the keyname definitions follow these rules:

- properties: existing property definitions may be refined; new property definitions may be added.
- attributes: existing attribute definitions may be refined; new attribute definitions may be added.
- valid_source_types: if valid_source_types is defined in the parent type, each element in this list must either be in the parent type list or derived from an element in the parent type list; if valid_source_types is not defined in the parent type then no restrictions are applied.

3.3.5.1.4 Example

```
mycompany.mytypes.myapplication.MyFeature:
  derived_from: tosca.capabilities.Root
  description: a custom feature of my company's application
  properties:
    my_feature_setting:
      type: string
    my_feature_value:
      type: integer
```

3.3.5.2 Capability definition

A Capability definition defines a named, typed set of data that a node can expose and is used to describe a relevant feature of the component described by the node. A Capability is defined part of a Node Type definition and may be refined during Node Type derivation.

3.3.5.2.1 Keynames

The following is the list of recognized keynames for a TOSCA capability definition:

Keyname	Required	Type	Constraints	Description
type	yes	string	N/A	The required name of the Capability Type this capability definition is based upon.
description	no	description	N/A	The optional description of the Capability definition.
properties	no	map of property refinements	- refinements apply to the definitions in the Capability Type - new properties may not be added	An optional map of property refinements for the Capability definition. The referred properties must have been defined in the Capability Type definition referred by the type keyword.
attributes	no	map of attribute refinements	- refinements apply to the definitions in the Capability Type - new attributes may not be added	An optional map of attribute refinements for the Capability definition. The referred attributes must have been defined in the Capability Type definition referred by the type keyword.
valid_source_types	no	list of string	if valid_source_types is defined in the Capability Type, each element in this list must either be in or derived from an element in the list defined in the type	An optional list of one or more valid names of Node Types that are supported as valid sources of any relationship established to the declared Capability Type. If undefined, all node types are valid sources.
occurrences	no	range of integer	if not defined the implied default is [1,UNBOUNDED]	The optional minimum and maximum of occurrences for the capability. The occurrence represents the maximum number of relationships that are allowed by the Capability. By default, an exported Capability should allow minimum one relationship to be formed with it and maximum a UNBOUNDED number of relationships.

3.3.5.2.2 Grammar

Capability definitions have one of the following grammars:

3.3.5.2.2.1   Short notation

The following single-line grammar may be used when only the capability type needs to be declared, without further refinement of the definitions in the capability type:

```
<capability_definition_name>: <capability_type>
```

3.3.5.2.2.2   Extended notation

The following multi-line grammar may be used when additional information on the capability definition is needed:

```
<capability_definition_name>:
  type: <capability_type>
  description: <capability_description>
  properties:
    <property_refinements>
  attributes:
    <attribute_refinements>
```

```
        valid_source_types: [ <node type_names> ]
        occurrences : <range_of_occurrences>
```

In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

- capability_definition_name**:** represents the symbolic name of the capability as a string.
- capability_type: represents the required name of a capability type the capability definition is based upon.
- capability_description: represents the optional description of the capability definition.
- property_refinements: represents the optional map of property definitions refinements for properties already defined in the capability type; new properties may not be added.
- attribute_refinements: represents the optional map of attribute definitions refinements for attributes already defined in the capability type; new attributes may not be added.
- node_type_names: represents the optional list of one or more names of Node Types that the Capability definition supports as valid sources for a successful relationship to be established to itself
    - if valid_source_types is defined in the capability type, each element in this list MUST either be in the capability type list or derived from an element in the capability type list; if valid_source_types is not defined in the capability type then no restrictions are applied.
- range_of_occurrences: represents he optional minimum and maximum occurrences for the capability
    - the occurrence represents the maximum number of relationships that are allowed by the capability; however, it does not restrict a lower number of relationships than the occurrence to be established.
    - in a node template, the occurrences keyname may be assigned to any number within the range_of_occurrences defined here.
    - if the occurrences is not assigned in the node template the TOSCA orchestrator may automatically set the occurrences to a number in the defined range (e.g. the maximum in the range).
    - the minimum in the range prevents the occurrences (during subsequent refinements or during assignment) to be set below this minimum.
    - by default (i.e. if occurrences is undefined here), a capability should allow at least one (1), and at most an unrestricted number (UNBOUNDED) of relationships to be formed to it.

3.3.5.2.3 Refinement rules

A capability definition within a node type uses the following definition refinement rules when the containing node type is derived:

- type: must be derived from (or the same as) the type in the capability definition in the parent node type definition.
- description: a new definition is unrestricted and will overwrite the one inherited from the capability definition in the parent node type definition.
- occurrences: the new range MUST be within the range defined in the capability definition in the parent node type definition.

- properties: not applicable to the definitions in the parent node type but to the definitions in the capability type referred by the type keyname (see grammar above for the rules).
- attributes: not applicable to the definitions in the parent node type but to the definitions in the capability type referred by the type keyname (see grammar above for the rules).
- valid_source_types: not applicable to the definitions in the parent node type but to the definitions in the capability type referred by the type keyname (see grammar above for the rules).

3.3.5.2.4 Examples

The following examples show capability definitions in both simple and full forms:

3.3.5.2.4.1 Simple notation example

```
# Simple notation, no properties defined or augmented
some_capability: mytypes.mycapabilities.MyCapabilityTypeName
```

3.3.5.2.4.2 Full notation example

```
# Full notation, augmenting properties of the referenced capability type
some_capability:
  type: mytypes.mycapabilities.MyCapabilityTypeName
  properties:
    limit:
      default: 100
```

3.3.5.2.5 Additional requirements

- Capability symbolic names SHALL be unique; it is an error if a capability name is found to occur more than once.
- If the occurrences keyname is not present, then a default declaration as follows will be assumed:
  - occurrences: [1, UNBOUNDED]

3.3.5.3 Capability assignment

A capability assignment allows node template authors to assign values to properties and attributes for a named capability definition that is part of a the node templates' respective type definition, and also to set the capability occurrences.

3.3.5.3.1 Keynames

The following is the list of recognized keynames for a TOSCA capability assignment:

Keyname	Required	Type	Description
properties	no	map of property assignments	An optional map of property assignments for the Capability definition.
attributes	no	map of attribute assignments	An optional map of attribute assignments for the Capability definition.
occurrences	no	integer	An optional integer that sets the number of occurrences. It defines the maximum number of allowed relationships to this capability. Must be within the range specified in the corresponding capability definition. If not defined, the orchestrator uses a suitable value from the range defined in the corresponding capability definition (e.g. the maximum in the range).

3.3.5.3.2 Grammar

Capability assignments have one of the following grammars:

```
<capability_definition_name>:
  properties:
```

```
      <property_assignments>
    attributes:
      <attribute_assignments>
    occurrences: <occurrences_value>
```

In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

- capability_definition_name: represents the symbolic name of the capability as a string.
- property_assignments: represents the optional map of property assignments that provide values for properties defined in the Capability definition.
- attribute_assignments: represents the optional map of attribute assignments that provide values for attributes defined in the Capability definition.
- occurrences_value: represents the optional integer that sets the number of occurrences
  - it represents the maximum number of relationships that are allowed by the capability; note that it does not restrict a lower number of relationships to be established.
  - must be within the range specified in the corresponding capability definition.
  - if not defined, the orchestrator uses a suitable value from the range defined in the corresponding capability definition (e.g. the maximum in the range).

3.3.5.3.3 Example

The following example shows a capability assignment:

3.3.5.3.3.1   Notation example

```
node_templates:
  some_node_template:
    capabilities:
      some_capability:
        properties:
          limit: 100
```

3.3.5.4 Requirement Type

A Requirement Type is a reusable entity that describes a kind of requirement that a Node Type can declare to expose.

TOSCA seeks to simplify the modeling by replacing the need for nodes declaring specific Requirement Types with nodes declaring their features sets using TOSCA Capability Types. So, it suffices that capabilites are advertised a-priory by Capability Types, while requirement definitions can be directly created during Node Type design.

Thus, currently the TOSCA specification does not utilize an independently defined Requirement Type. This is a desired effect as part of the simplification of the TOSCA v1.0 specification.

3.3.5.5 Requirement definition

The Requirement definition describes a named requirement (dependency) of a TOSCA node which needs to be fulfilled by a matching Capability definition declared by another TOSCA node. A Requirement is defined part of a Node Type definition and may be refined during Node Type derivation.

The requirement definition may itself include the specific name of the fulfilling entity (explicitly) or provide an abstract type, along with additional filtering characteristics, that a TOSCA orchestrator can use to fulfill the capability at runtime (implicitly).

3.3.5.5.1 Keynames

The following is the list of recognized keynames for a TOSCA requirement definition:

Keyname	Required	Type	Constraints	Description
description	no	string	N/A	The optional description of the Requirement definition.
capability	yes	string	N/A	The required keyname used to provide either the: • symbolic name of a **Capability definition** within a *target* Node Type that can fulfill the requirement. • name of a **Capability Type** that the TOSCA orchestrator will use to select a type-compatible *target* node to fulfill the requirement at runtime.
node	no	string	N/A	The optional keyname used to provide the name of a valid Node Type that contains the capability definition that can be used to fulfill the requirement. If a symbolic name of a **Capability definition** has been used for the capability keyname, then the node keyname is mandatory.
relationship	no	string	N/A	The optional keyname used to provide the name of a valid Relationship Type to construct a relationship when fulfilling the requirement.
node_filter	no	node filter	N/A	The optional filter definition that TOSCA orchestrators will use to select a type-compatible *target* node that can fulfill the associated abstract requirement at runtime.
occurrences	no	range of integer	implied default of [1,1]	The optional minimum and maximum occurrences for the requirement. Note: the keyword UNBOUNDED is also supported to represent any positive integer.

3.3.5.5.1.1 Additional Keynames for multi-line relationship grammar

The Requirement definition contains the Relationship Type information needed by TOSCA Orchestrators to construct relationships to other TOSCA nodes with matching capabilities; however, it is sometimes recognized that additional parameters may need to be passed to the relationship (perhaps for configuration). In these cases, additional grammar is provided so that the requirement definition may declare interface refinements (e.g. changing the implementation definition or declaring additional parameter definitions to be used as inputs/outputs).

Keyname	Required	Type	Constraints	Description
type	yes	string	N/A	The optional keyname used to provide the name of the Relationship Type as part of the relationship keyname definition.
interfaces	no	map of interface refinements	N/A	The optional keyname used to reference declared interface definitions on the corresponding Relationship Type for refinement.

3.3.5.5.2 Grammar

Requirement definitions have one of the following grammars:

3.3.5.5.2.1 Simple grammar (Capability Type only)

```
<requirement_definition_name>: <capability_type_name>
```

3.3.5.5.2.2 Extended grammar (with Node and Relationship Types)

```
<requirement_definition_name>:
  description: <requirement_description>
  capability: <capability_symbolic_name> | <capability_type_name>
  node: <node_type_name>
  relationship: <relationship_type_name>
  node_filter:
    <node_filter_definition>
  occurrences: [ <min_occurrences>, <max_occurrences> ]
```

3.3.5.5.2.3 Extended grammar for declaring Parameter Definitions on the relationship's Interfaces

The following additional multi-line grammar is provided for the relationship keyname in order to declare new parameter definitions for inputs/outputs of known Interface definitions of the declared Relationship Type.

```
<requirement_definition_name>:
  # Other keynames omitted for brevity
  relationship:
    type: <relationship_type_name>
    interfaces:
      <interface_refinements>
```

In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

- requirement_definition_name: represents the required symbolic name of the requirement definition as a string.

- requirement_description: represents the optional description of the requirement definition.

- capability_symbolic_name: represents the required symbolic name of the Capability definition within the target Node Type;

- capability_type_name: represents the required name of a Capability Type that can be used to fulfill the requirement.

- node_type_name: represents the name of a Node Type that contains the Capability Type definition the requirement can be fulfilled by; the node_type_name is required if the capability_symbolic_name was used, and is optional if the capability_type_name was used.

- relationship_type_name: represents the optional name of a Relationship Type to be used to construct a relationship between this requirement definition (i.e. in the source node) to a matching capability definition (in a target node).

- node_filter_definition: represents the optional node filter TOSCA orchestrators will use to fulfill the requirement when selecting a target node, or to verify that the specified node template fulfills the requirement (if a node template was specified during requirement assignment).

- min_occurrences, max_occurrences: represents the optional minimum and maximum range for the occurrences of the requirement (i.e. its cardinality)

  - the requirement occurrences define how many relationships are created from this requirement towards target capabilities, and its value is set during requirement assignment time to an integer in the range specified here.

  - by default (i.e. if occurrences is undefined here), a requirement shall form exactly one relationship (i.e. at least one, and at most one).

- interface_refinements: represents refinements for one or more already declared interface definitions in the Relationship Type (as declared on the type keyname)

  - allowing for the declaration of new parameter definitions for these interfaces or for specific operation or notification definitions of these interfaces or for the change of the description or implementation definitions.

3.3.5.5.3 Refinement rules

A requirement definition within a node type uses the following definition refinement rules when the containing node type is derived:

- description: a new definition is unrestricted and will overwrite the one inherited from the requirement definition in the parent node type definition.
- capability: the type of the capability must be derived from (or the same as) the capability type in the requirement definition in the parent node type definition
    - if the capability was specified using the symbolic name of a capability definition in the target node type, then the capability keyname definition MUST remain unchanged in any subsequent refinements or during assignment.
- node: must be derived from (or the same as) the node type in the requirement definition in the parent node type definition; if node is not defined in the parent type then no restrictions are applied
- relationship: must be derived from (or the same as) the relationship type in the requirement definition in the parent node type definition; if relationship is not defined in the parent type then no restrictions are applied
- node_filter: a new definition is unrestricted and will be considered in addition (i.e. logical and) to the node_filter definition in the parent node type definition; further refinements may add further node filtering.
- occurrences: the new range MUST be within the range defined in the requirement definition in the parent node type definition.

3.3.5.5.4 Additional requirements

- Requirement symbolic names SHALL be unique; it is an error if a requirement name is found to occur more than once.
- If the occurrences keyname is not present, then a default declaration as follows will be assumed:
    - occurrences: [1,1]

3.3.5.5.5 Notes

- The requirement symbolic name is used for identification of the requirement definition only and not relied upon for establishing any relationships in the topology.

3.3.5.5.6 Requirement definition is a tuple with a filter

A requirement definition allows type designers to govern which types are allowed (valid) for fulfillment using three levels of specificity with only the Capability definition or Capability Type being required.

1. Node Type (required/optional)
2. Relationship Type (optional)
3. Capability definition or Capability Type (required)

The first level allows selection, as shown in both the simple or complex grammar, simply providing the node's type using the node keyname. The second level allows specification of the relationship type to use when connecting the requirement to the capability using the relationship keyname. Finally, the specific Capability definition or Capability Type on the target node is provided using the capability keyname. Note that if a Capability definition is used, the Node Type definition is required (as it refers to a Capability definition in that Node Type).

In addition to the node, relationship and capability types, a filter, with the keyname node_filter, may be provided to constrain the allowed set of potential target nodes based upon their properties and their capabilities' properties. This allows TOSCA orchestrators to help find the "best fit" when selecting among multiple potential target nodes for the expressed requirements. Also, if a Node Template was specified during requirement assignment it allows TOSCA orchestrators to verify that the specified node template fulfills the requirement.

3.3.5.6 Requirement assignment

A Requirement assignment allows Node Template authors to provide assignments for individual and/or subsets of occurrences of the corresponding Requirement definition (i.e. having the same symbolic name) in the Node Type definition.

A Requirement assignment provides either names of Node Templates or selection criteria for TOSCA orchestrators to find matching TOSCA nodes that are used to fulfill the named requirement's declared Capability Type and/or Node Type. A Requirement assignment also provides either names of Relationship Templates (to use) or the name of Relationship Types (to create relationships) for relating the source node (containing the Requirement) to the target node (containing the Capability).

Note that several Requirement assignments in the Node Template definition can have the same symbolic name, each referring to different occurrences of the Requirement definition. To how many occurrences a particular assignment refers to is set via the occurrences keyname. Nevertheless, the sum of the occurrences' values for all of the Requirement assignments with the same symbolic name MUST be within the range of occurrences specified by the corresponding Requirement definition.

3.3.5.6.1 Keynames

The following is the list of recognized keynames for a TOSCA requirement assignment:

Keyname	Required	Type	Description
capability	no	string	The optional keyname used to provide either the: • symbolic name of a **Capability definition** within a *target* node that can fulfill the requirement. • name of a **Capability Type** that the TOSCA orchestrator will use to select a type-compatible *target* node to fulfill the requirement at runtime.
node	no	string	The optional keyname used to identify the target node of a relationship; specifically, it is used to provide either the: • name of a **Node Template** that can fulfill the *target* node requirement. • name of a **Node Type** that the TOSCA orchestrator will use to select a type-compatible *target* node to fulfill the requirement at runtime.
relationship	no	string	The optional keyname used to provide either the: • name of a **Relationship Template** to use to relate *this* node to the *target* node when fulfilling the requirement. • name of a **Relationship Type** that the TOSCA orchestrator will use to create a relationship to relate *this* node to the *target* node when fulfilling the requirement.
node_filter	no	node filter	The optional filter definition that TOSCA orchestrators will use to select a type-compatible *target* node that can fulfill the requirement at runtime.
occurrences	no	integer	An optional keyname that sets the occurrences for this requirement. The sum of all occurrences' values for all Requirement assignments with the same symbolic name must be within the range specified in the corresponding Requirement definition. If not defined, the assumed occurrences for an assignment is one (1).

The following is the list of recognized keynames for a TOSCA requirement assignment's relationship keyname which is used when property assignments or interface assignments (for e.g. changing the implementation keyname or declare additional parameter definitions to be used as inputs/outputs) need to be provided:

Keyname	Required	Type	Description
type	no	string	The optional keyname used to provide the name of the Relationship Type for the Requirement assignment's relationship.
properties	no	map of property assignments	An optional keyname providing property assignments for the relationship.
interfaces	no	map of interface assignments	The optional keyname providing Interface assignments for the corresponding Interface definitions in the Relationship Type.

3.3.5.6.2 Grammar

Requirement assignments have one of the following grammars:

3.3.5.6.2.1   Short notation:

The following single-line grammar may be used if only a concrete Node Template for the target node needs to be declared in the requirement:

```
<requirement_name>: <node_template_name>
```

This notation is only valid if the corresponding Requirement definition in the Node Template's parent Node Type declares a target capability which can be found in the Node Type of the node identified by node_template_name. A valid capability definition always needs to be provided in the requirement declaration of the *source* node to identify a specific capability definition in the *target* node the requirement will form a TOSCA relationship with.

3.3.5.6.2.2   Extended notation:

The following grammar should be used if the requirement assignment needs to provide more information than just the Node Template name:

```
<requirement_name>:
  capability: <capability_symbolic_name> | <capability_type_name>
  node: <node_template_name> | <node_type_name>
  relationship: <relationship_template_name> | <relationship_type_name>
  node_filter:
    <node_filter_definition>
  occurrences: <occurrences_value>
```

3.3.5.6.2.3   Extended grammar with Property Assignments and Interface Assignments for the relationship

The following additional multi-line grammar is provided for the relationship keyname in order to provide new Property assignments and Interface assignments for the created relationship of the declared Relationship.

```
<requirement_name>:
  # Other keynames omitted for brevity
  relationship:
    type: <relationship_template_name> | <relationship_type_name>
    properties:
      <property_assignments>
    interfaces:
      <interface_assignments>
```

In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

- requirement_name: represents the symbolic name of a requirement assignment as a string.
- capability_symbolic_name: represents the optional name of the Capability definition within the target Node Type or Node Template;
  - if the capability in the Requirement definition was specified using the symbolic name of a capability definition in a target node type, then the capability keyname definition MUST remain unchanged in any subsequent refinements or during assignment.
  - if the capability in the Requirement definition was specified using the name of a Capabiility Type, then the Capability definition referred here by capability_symbolic_name must be of a type that is the same as or derived from the said Capabiility Type in the Requirement definition.
- capability_type_name: represents the optional name of a Capability Type definition within the target Node Type or Node Template this requirement needs to form a relationship with;
  - may not be used if the capability in the Requirement definition was specified using the symbolic name of a capability definition in a target node type.
  - otherwise the capability_type_name must be of a type that is the same as or derived from the type defined by the capability keyname in the Requirement definition.
- node_template_name: represents the optional name of a Node Template that contains the capability this requirement will be fulfilled by;
  - in addition, the Node Type of the Node Template must be of a type that is the same as or derived from the type defined by the node keyname (if the node keyname is defined) in the Requirement definition.
- node_type_name: represents the optional name of a Node Type that contains the capability this Requirement will be fulfilled by;
  - in addition, the node_type_name must be of a type that is the same as or derived from the type defined by the node keyname (if the node keyname is defined) in the Requirement definition.
- relationship_template_name: represents the optional name of a Relationship Template to be used when relating the Requirement to the Capability in the target node.
  - in addition, the Relationship Type of the Relationship Template must be of a type that is the same as or derived from the type defined by the relationship keyname (if the capability keyname is defined) in the Requirement definition.
- relationship_type_name: represents the optional name of a Relationship Type that is compatible with the Capability Type in the target node; the TOSCA orchestrator will create a relationship of the named Relationship Type when relating the Requirement to the Capability in the target node.
  - in addition, the relationship_type_name must be of a type that is the same as or derived from the type defined by the relationship keyname (if the relationship keyname is defined) in the Requirement definition.
- property_assignments: represents the optional map of property assignments for the declared relationship.
- interface_assignments: represents the optional map of interface assignments for the declared relationship used to provide parameter assignments on inputs and outputs of interfaces, operations and notifications or changing the implementation definition.
- node_filter_definition: represents the optional node filter TOSCA orchestrators will use to fulfill the requirement for selecting a target node; if a node template was specified during requirement assignment, the TOSCA orchestrator will verify that the specified node template fulfills the node filter.
  - this node_filter does not replace the node_filter definition in the Requirement definition, it is applied in addition to that.
- occurrences_value: represents the optional occurrences number that specifies to how many occurrences within the Requirement definition this particular assignment refers to.
  - in addition, the sum of all occurrences_value for all Requirement assignments with the same symbolic name must be within the range specified in the Requirement definition.

- if not defined, the assumed occurrences_value for an assignment is one; i.e. the following default declaration will be assumed:
  - occurrences: 1

3.3.5.6.3 Examples

Examples of uses for the extended requirement assignment grammar include:

- The need to allow runtime selection of the target node a Node Type rather than a Node Template. This may include use of the node_filter keyname to provide node and capability filtering information to find the "best match" of a node at runtime.
- The need to further specify the Relationship Template or Relationship Type to use when relating the source node's requirement to the target node's capability.
- The need to further specify the capability (symbolic) name or Capability Type in the target node to form a relationship between.
- The need to specify the number of occurrences the requirement assigns (when greater than 1).

3.3.5.6.3.1   Example 1 – Hosting requirement on a Node Type

A web application node template named 'my_application_node_template' of type WebApplication declares a requirement named 'host' that needs to be fulfilled by any node that derives from the node type WebServer.

```
# Example of a requirement fulfilled by a specific web server node template
node_templates:
  my_application_node_template:
    type: tosca.nodes.WebApplication
    ...
    requirements:
      - host:
          node: tosca.nodes.WebServer
```

In this case, the node template's type is WebApplication which already declares the Relationship Type HostedOn to use to relate to the target node and the Capability Type of Container to be the specific target of the requirement in the target node.

3.3.5.6.3.2   Example 2 - Requirement with Node Template and a custom Relationship Type

This example is similar to the previous example; however, the requirement named 'database' describes a requirement for a connection to a database endpoint (Endpoint.Database) Capability Type in a named node template (my_database). However, the connection requires a custom Relationship Type (my.types.CustomDbConnection') declared on the keyname 'relationship'.

```
# Example of a (database) requirement that is fulfilled by a node template
named
# "my_database", but also requires a custom database connection
relationship
my_application_node_template:
  requirements:
    - database:
        node: my_database
        capability: Endpoint.Database
        relationship: my.types.CustomDbConnection
```

3.3.5.6.3.3   Example 3 - Requirement for a Compute node with additional selection criteria (filter)

This example shows how to extend an abstract 'host' requirement for a Compute node with a filter definition that further constrains TOSCA orchestrators to include additional properties and capabilities on the target node when fulfilling the requirement.

```
node_templates:
  mysql:
   type: tosca.nodes.DBMS.MySQL
    properties:
      # omitted here for brevity
    requirements:
     - host:
         node: tosca.nodes.Compute
         node_filter:
           capabilities:
             - host:
                 properties:
                   - num_cpus: { in_range: [ 1, 4 ] }
                   - mem_size: { greater_or_equal: 512 MB }
             - os:
                 properties:
                   - architecture: { equal: x86_64 }
                   - type: { equal: linux }
                   - distribution: { equal: ubuntu }
             - mytypes.capabilities.compute.encryption:
                 properties:
                   - algorithm: { equal: aes }
                   - keylength: { valid_values: [ 128, 256 ] }
```

3.3.5.6.3.4 Example 4 - Requirement assignment for definition with occurrences: [2,2]

This example shows how the assignments can look if the Requirement definition has the occurrences range different from the default [1,1]. In this case the redundant_database requirement has occurrences: [2,2]. The Requirement definition is not presented here for brevity. In the Requirement assignment we use the short notation. Note that the occurrences keyname for each assignment is not declared (i.e. the default value of 1 is used) and that the sum of the occurrences values of both assignments is 2 which is in the range of [2,2] as specified in the Requirement definition.

```
# Example of a (redundant_database) requirement that is fulfilled by
# two node templates named "database1" and "database1
my_critical_application_node_template:
  requirements:
    - redundant_database: database1
    - redundant_database: database2
```

3.3.5.7 Node Filter definition

A node filter defines criteria for selection of a target node based upon its property values, capabilities and capability properties.

3.3.5.7.1 Keynames

The following is the list of recognized keynames for a TOSCA node filter definition:

Keyname	Required	Type	Description
properties	no	list of property filter definition	An optional list of property filters that will be used to select (filter) matching TOSCA entities (e.g., Node Template, Node Type, Capability Types, etc.) based upon their property definitions' values.

Keyname	Required	Type	Description
capabilities	no	list of capability names or capability type names	An optional list of capability names or types that will be used to select (filter) matching TOSCA entities based upon their existence.

3.3.5.7.2 Additional filtering on capability properties

Capabilities used as filters often have their own sets of properties which also can be used to construct a filter.

Keyname	Required	Type	Description
properties (within a capability name or type name)	no	list of property filter definitions	An optional list of property filters that will be used to select (filter) matching TOSCA entities (e.g., Node Template, Node Type, Capability Types, etc.) based upon their capabilities' property definitions' values.

3.3.5.7.3 Grammar

Node filter definitions have following grammar:

```
node_filter:
  properties:
    - <property_filter_def_1>
    - ...
    - <property_filter_def_n>
  capabilities:
    - <capability_name_or_type_1>:
        properties:
          - <cap_1_property_filter_def_1>
          - ...
          - <cap_1_property_filter_def_n>
    - ...
    - <capability_name_or_type_m>:
        properties:
          - <cap_m_property_filter_def_1>
          - ...
          - <cap_m_property_filter_def_n>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- property_filter_def_*: represents a property filter definition that will be used to select (filter) matching TOSCA entities (e.g., Node Template, Node Type, Capability Types, etc.) based upon their property definitions' values.

- capability_name_or_type_*: represents the type or name of a capability that will be used to select (filter) matching TOSCA entities based upon their existence.

- cap_*_property_def_*: represents a property filter definition that will be used to select (filter) matching TOSCA entities (e.g., Node Template, Node Type, Capability Types, etc.) based upon their capabilities' property definitions' values.

3.3.5.7.4 Additional requirements

- TOSCA orchestrators **SHALL** search for matching capabilities listed on a target filter by assuming the capability name is first a symbolic name and secondly it is a type name (in order to avoid namespace collisions).

3.3.5.7.5 Example

The following example is a filter that will be used to select a Compute node based upon the values of its defined capabilities. Specifically, this filter will select Compute nodes that support a specific range of CPUs (i.e., num_cpus value between 1 and 4) and memory size (i.e., mem_size of 2 or greater) from its declared "host" capability.

```
my_node_template:
  # other details omitted for brevity
  requirements:
    - host:
        node_filter:
          capabilities:
            # My "host" Compute node needs these properties:
            - host:
                properties:
                  - num_cpus: { in_range: [ 1, 4 ] }
                  - mem_size: { greater_or_equal: 512 MB }
```

3.3.5.8 Property Filter definition

A property filter definition defines criteria, using constraint clauses, for selection of a TOSCA entity based on its property values. Constraint clauses are further defined in Section **Error! Reference source not f ound. Error! Reference source not found.**.

3.3.5.8.1 Grammar

Property filter definitions have one of the following grammars:

3.3.5.8.1.1 Short notation:

The following single-line grammar may be used when only a single constraint is needed on a property:

```
<property_name>: <property_constraint_clause>
```

3.3.5.8.1.2 Extended notation:

The following multi-line grammar may be used when multiple constraints are needed on a property:

```
<property_name>:
  - <property_constraint_clause_1>
  - ...
  - <property_constraint_clause_n>
```

In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

* property_name: represents the name of property that will be used to select a property definition with the same name (property_name) on a TOSCA entity (e.g., a Node Type, Node Template, Capability Type, etc.).
* property_constraint_clause_*: represents constraint clause(s) that will be used to filter entities based upon the named property's value(s).

3.3.5.8.2 Additional Requirements

* Property constraint clauses must be type compatible with the property definitions (of the same name) as defined on the target TOSCA entity that the clause will be applied against.

3.3.6 Interfaces

3.3.6.1 Interface Type

An Interface Type is a reusable entity that describes a set of operations that can be used to interact with or to manage a node or relationship in a TOSCA topology.

3.3.6.1.1 Keynames

The Interface Type is a TOSCA type entity and has the common keynames listed in Section 3.2.5.2 Common keynames in type definitions. In addition, the Interface Type has the following recognized keynames:

Keyname	Required	Type	Description
inputs	no	map of parameter definitions	The optional map of input parameter definitions available to all operations defined for this interface.
operations	no	map of operation definitions	The optional map of operations defined for this interface.
notifications	no	map of notification definitions	The optional map of notifications defined for this interface.

3.3.6.1.2 Grammar

Interface Types have following grammar:

```
<interface_type_name>:
  derived_from: <parent_interface_type_name>
  version: <version_number>
  metadata:
    <map of string>
  description: <interface_description>
  inputs:
    <parameter_definitions>
  operations:
    <operation_definitions>
  notifications:
    <Error! Reference source not found.Error! Reference source not fo
und.notification_definitions>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- interface_type_name: represents the required name of the interface as a string.
- parent_interface_type_name: represents the name of the Interface Type this Interface Type definition derives from (i.e. its "parent" type).
- version_number: represents the optional TOSCA version number for the Interface Type.
- interface_description: represents the optional description for the Interface Type.
- parameter_definitions: represents the optional map of parameter definitions which the TOSCA orchestrator will make available (i.e., or pass) to all implementation artifacts for operations declared on the interface during their execution.
- operation_definitions: represents the optional map of one or more operation definitions.
- notification_definitions: represents the optional map of one or more notification definitions.

3.3.6.1.3 Derivation rules

During Interface Type derivation the keyname definitions follow these rules:

- inputs: existing parameter definitions may be refined; new parameter definitions may be added.
- operations: existing operation definitions may be refined; new operation definitions may be added.
- notifications: existing notification definitions may be refined; new notification definitions may be added.

3.3.6.1.4 Example

The following example shows a custom interface used to define multiple configure operations.

```
mycompany.mytypes.myinterfaces.MyConfigure:
  derived_from: tosca.interfaces.relationship.Root
  description: My custom configure Interface Type
  inputs:
    mode:
      type: string
  operations:
    pre_configure_service:
      description: pre-configure operation for my service
    post_configure_service:
      description: post-configure operation for my service
```

3.3.6.1.5 Additional Requirements

- Interface Types **MUST NOT** include any implementations for defined operations or notifications; that is, the implementation keyname is invalid in this context.

3.3.6.1.6 Notes

- Starting with Version 1.3 of this specification, interface type definition grammar was changed to support notifications as well as operations. As a result, operations must now be specified under the newly introduced operations keyname and the notifications under the new notifications keyname. For backward compatibility if neither the operations nor notifications are specified then we assume the symbolic names in the interface definition to mean operations, but this use is deprecated. Operations and notifications names should not overlap.

3.3.6.2 Interface definition

An Interface definition defines a named interface (containing operations and notifications definitions) that can be associated with (i.e. defined within) a Node or Relationship Type definition (including Interface definitions in Requirements definitions). An Interface definition may be refined in subsequent Node or Relationship Type derivations.

3.3.6.2.1 Keynames

The following is the list of recognized keynames for a TOSCA interface definition:

Keyname	Required	Type	Description
type	yes	string	The required name of the Interface Type this interface definition is based upon.
description	no	description	The optional description for this interface definition.
inputs	no	map of parameter definitions and refinements	The optional map of input parameter refinements and new definitions available to all operations defined for this interface. The inputs to be refined have been defined in the Interface Type definition.

Keyname	Required	Type	Description
operations	no	map of operation refinements	The optional map of operations refinements for this interface. The referred operations must have been defined in the Interface Type definition.
notifications	no	map of notification refinements	The optional map of notifications refinements for this interface. The referred operations must have been defined in the Interface Type definition.

3.3.6.2.2 Grammar

Interface definitions in Node or Relationship Type definitions have the following grammar:

```
<interface_definition_name>:
  type: <interface_type_name>
  description: <interface_description>
  inputs:
    <parameter_definitions_and_refinements>
  operations:
    <operation_refinements>
  notifications:
    <Error! Reference source not found.Error! Reference source not fo
und.notification_refinements>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- interface_definition_name: represents the required symbolic name of the interface as a string.
- interface_type_name: represents the required name of the Interface Type for the interface definition.
- interface_description: represents the optional description string for the interface.
- parameter_definitions_and_refinements: represents the optional map of input parameters which the TOSCA orchestrator will make available (i.e. pass) to all defined operations. This means these parameters and their values will be accessible to the implementation artifacts (e.g., scripts) associated to each operation during their execution
    - the map represents a mix of parameter refinements (for parameters already defined in the Interface Type) and new parameter definitions.
    - with the new parameter definitions, we can flexibly add new parameters when changing the implementation of operations and notifications during refinements or assignments.
- operation_refinements: represents the optional map of operation definition refinements for this interface; the referred operations must have been previously defined in the Interface Type.
- notification_refinements: represents the optional map of notification definition refinements for this interface; the referred notifications must have been previously defined in the Interface Type.

3.3.6.2.3 Refinement rules

An interface definition within a node or relationship type (including interface definitions in requirements definitions) uses the following definition refinement rules when the containing entity type is derived:

- type: must be derived from (or the same as) the type in the interface definition in the parent entity type definition.
- description: a new definition is unrestricted and will overwrite the one inherited from the interface definition in the parent entity type definition.

- inputs: not applicable to the definitions in the parent entity type but to the definitions in the interface type referred by the type keyname (see grammar above for the rules).
- operations: not applicable to the definitions in the parent entity type but to the definitions in the interface type referred by the type keyname (see grammar above for the rules).

- notifications: not applicable to the definitions in the parent entity type but to the definitions in the interface type referred by the type keyname (see grammar above for the rules).

3.3.6.3 Interface assignment

An Interface assignment is used to specify assignments for the inputs, operations and notifications defined in the Interface. Interface assignments may be used within a Node or Relationship Template definition (including when Interface assignments are referenced as part of a Requirement assignment in a Node Template).

3.3.6.3.1 Keynames

The following is the list of recognized keynames for a TOSCA interface definition:

Keyname	Required	Type	Description
inputs	no	map of parameter value assignments	The optional map of input parameter assignments. Template authors MAY provide parameter assignments for interface inputs that are not defined in their corresponding Interface Type.
operations	no	map of operation assignments	The optional map of operations assignments specified for this interface.
notifications	no	map of notification assignments	The optional map of notifications assignments specified for this interface.

3.3.6.3.2 Grammar

Interface assignments have the following grammar:

```
<interface_definition_name>:
  inputs:
    <parameter_value_assignments>
  operations:
    <operation_assignments>
  notifications:
    <notification_assignments>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- interface_definition_name: represents the required symbolic name of the interface as a string.
- parameter_value_assignments: represents the optional map of parameter value assignments for passing input parameter values to all interface operations
  - template authors MAY provide new parameter assignments for interface inputs that are not defined in the Interface definition.
- operation_assignments: represents the optional map of operation assignments for operations defined in the Interface definition.
- notification_assignments: represents the optional map of notification assignments for notifications defined in the Interface definition.

3.3.6.4 Operation definition

An operation definition defines a named function or procedure to which an operation implementation can be bound.

A new operation definition may be declared only inside interface type definitions (this is the only place where new operations can be defined). In interface type, node type, or relationship type definitions

(including operation definitions as part of a requirement definition) we may further refine operations already defined in an interface type.

An operation definition or refinement inside an interface type definition may not contain an operation implementation definition and it may not contain an attribute mapping as part of its output definition (as both these keynames are node/relationship specific).

3.3.6.4.1 Keynames

The following is the list of recognized keynames for a TOSCA operation definition (including definition refinement)

Keyname	Required	Type	Description
description	no	description	The optional description string for the associated operation.
implementation	no	operation implementation definition	The optional definition of the operation implementation. May not be used in an interface type definition (i.e. where an operation is initially defined), but only during refinements.
inputs	no	map of parameter definitions	The optional map of parameter definitions for operation input values.
outputs	no	map of parameter definitions	The optional map of parameter definitions for operation output values. Only as part of node and relationship type definitions, the output definitions may include mappings onto attributes of the node or relationship type that contains the definition.

3.3.6.4.2 Grammar

Operation definitions have the following grammar:

3.3.6.4.2.1   Short notation

The following single-line grammar may be used when the operation's implementation definition is the only keyname that is needed, and when the operation implementation definition itself can be specified using a single line grammar:

```
<operation_name>: <operation_implementation_definition>
```

3.3.6.4.2.2   Extended notation

The following multi-line grammar may be used when additional information about the operation is needed:

```
<operation_name>:
  description: <operation_description>
  implementation: <operation_implementation_definition>
  inputs:
    <parameter_definitions>
  outputs:
    <parameter_definitions>
```

In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

- operation_name: represents the required symbolic name of the operation as a string.
- operation_description: represents the optional description string for the operation.

- operation_implementation_definition: represents the optional specification of the operation's implementation).
- parameter_definitions: represents the optional map of parameter definitions which the TOSCA orchestrator will make available as inputs to or receive as outputs from the corresponding implementation artifact during its execution.

3.3.6.4.3 Refinement rules

An operation definition within an interface, node, or relationship type (including interface definitions in requirements definitions) uses the following refinement rules when the containing entity type is derived:

- description: a new definition is unrestricted and will overwrite the one inherited from the operation definition in the parent entity type definition.
- implementation: a new definition is unrestricted and will overwrite the one inherited from the operation definition in the parent entity type definition.
- inputs: parameter definitions inherited from the parent entity type may be refined; new parameter definitions may be added.
- outputs: parameter definitions inherited from the parent entity type may be refined; new parameter definitions may be added.

3.3.6.4.4 Additional requirements

- The definition of implementation is not allowed in interface type definitions (as a node or node type context is missing at that point). Thus, it can be part only of an operation refinement and not of the original operation definition.
- The default refinement behavior for implementations SHALL be overwrite. That is, implementation definitions in a derived type overwrite any defined in its parent type.
- Defining a fixed value for an input parameter (as part of its definition) may only use a `parameter_value_expression` that is meaningful in the scope of the context. For example, within the context of an Interface Type definition functions such as get_propery or get_attribute cannot be used. Within the context of Node or Relationship Type definitions, these functions may only reference properties and attributes of the same node (i.e. SELF), respectively same relationship or its target (i.e. SELF or TARGET). For example, `value: { get_property: [SELF, property1] }`
- Defining attribute mapping as part of the output parameter definition is not allowed in interface type definitions (i.e. as part of operation definitions). It is allowed only in node and relationship type definitions (as part of operation refinements) and has to be meaningful in the scope of the context (i.e. SELF in node types and SELF or TARGET in relationship types).
- Implementation artifact file names (e.g., script filenames) may include file directory path names that are relative to the TOSCA service template file itself when packaged within a TOSCA Cloud Service Archive (CSAR) file.

3.3.6.4.5 Examples

3.3.6.4.5.1 Single-line example

```
interfaces:
  Standard:
    start: scripts/start_server.sh
```

3.3.6.4.5.2 Multi-line example with shorthand implementation definitions

```
interfaces:
  Configure:
    pre_configure_source:
      implementation:
```

```
                primary: scripts/pre_configure_source.sh
                dependencies:
                  - scripts/setup.sh
                  - binaries/library.rpm
                  - scripts/register.py
```

3.3.6.4.5.3   Multi-line example with extended implementation definitions

```
interfaces:
  Configure:
    pre_configure_source:
      implementation:
        primary:
          file: scripts/pre_configure_source.sh
          type: tosca.artifacts.Implementation.Bash
          repository: my_service_catalog
        dependencies:
          - file : scripts/setup.sh
            type : tosca.artifacts.Implementation.Bash
            repository : my_service_catalog
```

3.3.6.5 Operation assignment

An operation assignment may be used to assign values for input parameters, specify attribute mappings for output parameters, and define/redefine the implementation definition of an already defined operation in the interface definition. An operation assignment may be used inside interface assignments inside node template or relationship template definitions (this includes when operation assignments are part of a requirement assignment in a node template).

An operation assignment may add or change the implementation and description definition of the operation. Assigning a value to an input parameter that had a fixed value specified during operation definition or refinement is not allowed. Providing an attribute mapping for an output parameter that was mapped during an operation refinement is also not allowed.

Note also that in the operation assignment we can use inputs and outputs that have not been previously defined in the operation definition. This is equivalent to an ad-hoc definition of a parameter, where the type is inferred from the assigned value (for input parameters) or from the attribute to map to (for output parameters).

3.3.6.5.1 Keynames

The following is the list of recognized keynames for an operation assignment:

Keyname	Required	Type	Description
implementation	no	operation implementation definition	The optional definition of the operation implementation. Overrides implementation provided at operation definition.
inputs	no	map of parameter value assignments	The optional map of parameter value assignments for assigning values to operation inputs.
outputs	no	map of parameter mapping assignments	The optional map of parameter mapping assignments that specify how operation outputs are mapped onto attributes of the node or relationship that contains the operation definition.

3.3.6.5.2 Grammar

Operation assignments have the following grammar:

3.3.6.5.2.1 Short notation

The following single-line grammar may be used when the operation's implementation definition is the only keyname that is needed, and when the operation implementation definition itself can be specified using a single line grammar:

```
<operation_name>: <operation_implementation_definition>
```

3.3.6.5.2.2 Extended notation

The following multi-line grammar may be used in Node or Relationship Template definitions when additional information about the operation is needed:

```
<operation_name>:
    implementation: <operation_implementation_definition>
    inputs:
      <parameter_value_assignments>
    outputs:
      <parameter_mapping_assignments>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- operation_name: represents the required symbolic name of the operation as a string.
- operation_implementation_definition: represents the optional specification of the operation's implementation
  - the implementation declared here overrides the implementation provided at operation definition.
- parameter_value_assignments: represents the optional map of parameter value assignments for passing input parameter values to operations.
  - assignments for operation inputs that are not defined in the operation definition may be provided
- parameter_mapping_assignments: represents the optional map of parameter mapping assignments that consists of named output values returned by operation implementations (i.e. artifacts) and associated attributes into which this output value must be stored
  - assignments for operation outputs that are not defined in the operation definition may be provided.

3.3.6.5.3 Additional requirements

- The behavior for implementation of operations SHALL be override. That is, implementation definitions assigned in an operation assignment override any defined in the operation definition.
- Template authors MAY provide parameter assignments for operation inputs that are not defined in the operation definition.
- Template authors MAY provide attribute mappings for operation outputs that are not defined in the operation definition.
- Implementation artifact file names (e.g., script filenames) may include file directory path names that are relative to the TOSCA service template file itself when packaged within a TOSCA Cloud Service Archive (CSAR) file.

3.3.6.5.4 Examples

TBD

3.3.6.6 Notification definition

A notification definition defines a named asynchronous notification or incoming message that can be associated with an interface. The notification is a way for an external event to be transmitted to the TOSCA orchestrator. Values can be sent with a notification as notification outputs and we can map them

to node/relationship attributes similarly to the way operation outputs are mapped to attributes. The artifact that the orchestrator is registering with in order to receive the notification is specified using the implementation keyname in a similar way to operations. As opposed to an operation definition, a notification definition does not include an inputs keyname since notifications are not invoked from the orchestrator.

When the notification is received an event is generated within the orchestrator that can be associated to triggers in policies to call other internal operations and workflows. The notification name (the unqualified full name) itself identifies the event type that is generated and can be textually used when defining the associated triggers.

A notification definition may be used only inside interface type definitions (this is the only place where new notifications can be defined). Inside interface type, node type, or relationship type definitions (including notifications definitions as part of a requirement definition) we may further refine a notification already defined in the interface type.

A notification definition or refinement inside an interface type definition may not contain a notification implementation definition and it may not contain an attribute mapping as part of its output definition (as both these keynames are node/relationship specific).

3.3.6.6.1 Keynames

The following is the list of recognized keynames for a TOSCA notification definition:

Keyname	Required	Type	Description
description	no	description	The optional description string for the associated notification.
implementation	no	notification implementation definition	The optional definition of the notification implementation.
outputs	no	map of parameter definitions	The optional map of parameter definitions that specify notification output values. Only as part of node and relationship type definitions, the output definitions may include their mappings onto attributes of the node type or relationship type that contains the definition.

3.3.6.6.2 Grammar

Notification definitions have the following grammar:

3.3.6.6.2.1 Short notation

The following single-line grammar may be used when the notification's implementation definition is the only keyname that is needed and when the notification implementation definition itself can be specified using a single line grammar:

```
<notification_name>: <notification_implementation_definition>
```

3.3.6.6.2.2 Extended notation

The following multi-line grammar may be used when additional information about the notification is needed:

```
<notification_name>:
  description: <notification_description>
  implementation: <notification_implementation_definition>
  outputs:
    <parameter_definitions>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- notification_name: represents the required symbolic name of the notification as a string.
- notification_description: represents the optional description string for the notification.
- notification_implementation_definition: represents the optional specification of the notification implementation (i.e. the external artifact that may send notifications)
- parameter_definitions: represents the optional map of parameter definitions for parameters that the orchestrator will receive as outputs from the corresponding implementation artifact during its execution.

3.3.6.6.3 Refinement rules

A notification definition within an interface, node, or relationship type (including interface definitions in requirements definitions) uses the following refinement rules when the containing entity type is derived:

- description: a new definition is unrestricted and will overwrite the one inherited from the notification definition in the parent entity type definition.
- implementation: a new definition is unrestricted and will overwrite the one inherited from the notification definition in the parent entity type definition.
- outputs: parameter definitions inherited from the parent entity type may be refined; new parameter definitions may be added.

3.3.6.6.4 Additional requirements

- The definition of implementation is not allowed in interface type definitions (as a node or node type context is missing at that point). Thus, it can be part only of a notification refinement and not of the original notification definition.
- The default sub-classing (i.e. refinement) behavior for implementations of notifications SHALL be overwrite. That is, implementation artifacts definitions in a derived type overwrite any defined in its parent type.
- Defining attribute mapping as part of the output parameter definition is not allowed in interface type definitions (i.e. as part of operation definitions). It is allowed only in node and relationship type definitions (as part of operation refinements).
- Defining a mapping in an output parameter definition may use an attribute target that is meaningful in the scope of the context. Within the context of Node Type definitions these functions may only reference attributes of the same node (i.e. SELF). Within the context of Relationship Type definitions, they may reference attributes of the relationship itself or its target node (i.e. SELF or TARGET).
- Implementation artifact file names (e.g., script filenames) may include file directory path names that are relative to the TOSCA service template file itself when packaged within a TOSCA Cloud Service Archive (CSAR) file.

3.3.6.6.5 Examples

TBD

3.3.6.7 Notification assignment

A notification assignment may be used to specify attribute mappings for output parameters and to define/redefine the implementation definition and description definition of an already defined notification in the interface definition. A notification assignment may be used inside interface assignments inside node or relationship template definitions (this includes when notification assignments are part of a requirement assignment in a node template).

Providing an attribute mapping for an output parameter that was mapped during a previous refinement is not allowed. Note also that in the notification assignment we can use outputs that have not been previously defined in the operation definition. This is equivalent to an ad-hoc definition of an output parameter, where the type is inferred from the attribute to map to.

3.3.6.7.1 Keynames

The following is the list of recognized keynames for a TOSCA notification assignment:

Keyname	Required	Type	Description
implementation	no	notification implementation definition	The optional definition of the notification implementation. Overrides implementation provided at notification definition.
outputs	no	map of parameter mapping assignments	The optional map of parameter mapping assignments that specify how notification outputs values are mapped onto attributes of the node or relationship type that contains the notification definition.

3.3.6.7.2 Grammar

Notification assignments have the following grammar:

3.3.6.7.2.1   Short notation

The following single-line grammar may be used when the notification's implementation definition is the only keyname that is needed, and when the notification implementation definition itself can be specified using a single line grammar:

```
<notification_name>: <notification_implementation_definition>
```

3.3.6.7.2.2   Extended notation

The following multi-line grammar may be used in Node or Relationship Template definitions when additional information about the notification is needed:

```
<notification_name>:
  implementation: <notification_implementation_definition>
  outputs:
    <parameter_mapping_assignments>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- notification_name: represents the required symbolic name of the notification as a string.
- notification_implementation_definition: represents the optional specification of the notification implementation (i.e. the external artifact that is may send notifications)
  - the implementation declared here overrides the implementation provided at notification definition.
- parameter_mapping_assignments: represents the optional map of parameter_mapping_assignments that consists of named output values returned by operation implementations (i.e. artifacts) and associated attributes into which this output value must be stored
  - assignments for notification outputs that are not defined in the operation definition may be provided.

3.3.6.7.3 Additional requirements

- The behavior for implementation of notifications SHALL be override. That is, implementation definitions assigned in a notification assignment override any defined in the notification definition.
- Template authors MAY provide attribute mappings for notification outputs that are not defined in the corresponding notification definition.

- Implementation artifact file names (e.g., script filenames) may include file directory path names that are relative to the TOSCA service template file itself when packaged within a TOSCA Cloud Service Archive (CSAR) file.

3.3.6.7.4 Examples

TBD

3.3.6.8 Operation and notification implementation definition

An operation implementation definition specifies one or more artifacts (e.g. scripts) to be used as the implementation for an operation in an interface.

A notification implementation definition specifies one or more artifacts to be used by the orchestrator to subscribe and receive a particular notification (i.e. the artifact implements the notification).

The operation implementation definition and the notification implementation definition share the same keynames and grammar, with the exception of the timeout keyname that has no meaning in the context of a notification implementation definition and should not be used in such.

3.3.6.8.1 Keynames

The following is the list of recognized keynames for an operation implementation definition or a notification implementation definition:

Keyname	Required	Type	Description
primary	no	artifact definition	The optional implementation artifact (i.e., the primary script file within a TOSCA CSAR file).
dependencies	no	list of artifact definition	The optional list of one or more dependent or secondary implementation artifacts which are referenced by the primary implementation artifact (e.g., a library the script installs or a secondary script).
timeout	no	integer	Timeout value in seconds. Has no meaning and should not be used within a notification implementation definition.

3.3.6.8.2 Grammar

Operation implementation definitions and notification implementation definitions have the following grammar:

3.3.6.8.2.1   Short notation for use with single artifact

The following single-line grammar may be used when only a primary implementation artifact name is needed:

```
implementation: <primary_artifact_name>
```

This notation can be used when the primary artifact name uniquely identifies the artifact, either because it refers to an artifact specified in the artifacts section of a type or template, or because it represents the name of a script in the CSAR file that contains the definition.

3.3.6.8.2.2   Short notation for use with multiple artifacts

The following multi-line short-hand grammar may be used when multiple artifacts are needed, but each of the artifacts can be uniquely identified by name as before:

```
implementation:
  primary: <primary_artifact_name>
```

```
      dependencies:
        - <list_of_dependent_artifact_names>
      timeout : 60
```

3.3.6.8.2.3   Extended notation for use with single artifact

The following multi-line grammar may be used in Node or Relationship Type or Template definitions when only a single artifact is used but additional information about the primary artifact is needed (e.g. to specify the repository from which to obtain the artifact, or to specify the artifact type when it cannot be derived from the artifact file extension):

```
implementation:
  primary:
    <primary_artifact_definition>
  timeout : 100
```

3.3.6.8.2.4   Extended notation for use with multiple artifacts

The following multi-line grammar may be used in Node or Relationship Type or Template definitions when there are multiple artifacts that may be needed for the operation to be implemented and additional information about each of the artifacts is required:

```
implementation:
  primary:
    <primary_artifact_definition>
  dependencies:
    - <list_of_dependent_artifact definitions>
  timeout: 120
```

In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

- primary_artifact_name: represents the optional name (string) of an implementation artifact definition (defined elsewhere), or the direct name of an implementation artifact's relative filename (e.g., a service template-relative, path-inclusive filename or absolute file location using a URL).

- primary_artifact_definition: represents a full inline definition of an implementation artifact.

- list_of_dependent_artifact_names: represents the optional ordered list of one or more dependent or secondary implementation artifact names (as strings) which are referenced by the primary implementation artifact. TOSCA orchestrators will copy these files to the same location as the primary artifact on the target node so as to make them accessible to the primary implementation artifact when it is executed.

- list_of_dependent_artifact_definitions: represents the ordered list of one or more inline definitions of dependent or secondary implementation artifacts. TOSCA orchestrators will copy these artifacts to the same location as the primary artifact on the target node so as to make them accessible to the primary implementation artifact when it is executed.

3.3.7 Artifacts

3.3.7.1 Artifact Type

An Artifact Type is a reusable entity that defines the type of one or more files that are used to define implementation or deployment artifacts that are referenced by nodes or relationships.

3.3.7.1.1 Keynames

The Artifact Type is a TOSCA type entity and has the common keynames listed in Section 3.2.5.2 Common keynames in type definitions. In addition, the Artifact Type has the following recognized keynames:

Keyname	Required	Type	Constraints	Description
mime_type	no	string	None	The required mime type property for the Artifact Type.
file_ext	no	list of string	None	The required file extension property for the Artifact Type.
properties	no	map of property definitions	No	An optional map of property definitions for the Artifact Type.

3.3.7.1.2 Grammar

Artifact Types have following grammar:

```
<artifact_type_name>:
  derived_from: <parent_artifact_type_name>
  version: <version_number>
  metadata:
    <map of string>
  description: <artifact_description>
  mime_type: <mime_type_string>
  file_ext: [ <file_extensions> ]
  properties:
    <property_definitions>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- artifact_type_name: represents the name of the Artifact Type being declared as a string.
- parent_artifact_type_name: represents the name of the Artifact Type this Artifact Type definition derives from (i.e., its "parent" type).
- version_number: represents the optional TOSCA version number for the Artifact Type.
- artifact_description: represents the optional description string for the Artifact Type.
- mime_type_string: represents the optional Multipurpose Internet Mail Extensions (MIME) standard string value that describes the file contents for this type of Artifact Type as a string.
- file_extensions: represents the optional list of one or more recognized file extensions for this type of artifact type as strings.
- property_definitions: represents the optional map of property definitions for the artifact type.

3.3.7.1.3 Examples

```
my_artifact_type:
  description: Java Archive artifact type
  derived_from: tosca.artifact.Root
  mime_type: application/java-archive
  file_ext: [ jar ]
  properties:
    id:
      description: Identifier of the jar
      type: string
      required: true
    creator:
      description: Vendor of the java implementation on which the jar is
based
      type: string
      required: false
```

3.3.7.1.4 Additional Requirements

- The 'mime_type' keyname is meant to have values that are Apache mime types such as those defined here: http://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types

3.3.7.1.5 Notes

Information about artifacts can be broadly classified in two categories that serve different purposes:

Selection of artifact processor – This category includes informational elements such as artifact version, checksum, checksum algorithm etc. and s used by TOSCA Orchestrator to select the correct artifact processor for the artifact. These informational elements are captured in TOSCA as keywords for the artifact.

Properties processed by artifact processor - Some properties are not processed by the Orchestrator, but passed on to the artifact processor to assist with proper processing of the artifact. These informational elements are described through artifact properties.

3.3.7.2 Artifact definition

An artifact definition defines a named, typed file that can be associated with Node Type or Node Template and used by orchestration engine to facilitate deployment and implementation of interface operations.

3.3.7.2.1 Keynames

The following is the list of recognized keynames for a TOSCA artifact definition when using the extended notation:

Keyname	Required	Type	Description
type	yes	string	The required artifact type for the artifact definition.
file	yes	string	The required URI string (relative or absolute) which can be used to locate the artifact's file.
repository	no	string	The optional name of the repository definition which contains the location of the external repository that contains the artifact. The artifact is expected to be referenceable by its file URI within the repository.
description	no	description	The optional description for the artifact definition.
deploy_path	no	string	The file path the associated file will be deployed on within the target node's container.
artifact_version	no	string	The version of this artifact. One use of this artifact_version is to declare the particular version of this artifact type, in addition to its mime_type (that is declared in the artifact type definition). Together with the mime_type it may be used to select a particular artifact processor for this artifact. For example a python interpreter that can interpret python version 2.7.0
checksum	no	string	The checksum used to validate the integrity of the artifact.
checksum_algorithm	no	string	Algorithm used to calculate the artifact checksum (e.g. MD5, SHA [Ref]). Shall be specified if checksum is specified for an artifact.
properties	no	map of property assignments	The optional map of property assignments associated with the artifact.

3.3.7.2.2 Grammar

Artifact definitions have one of the following grammars:

3.3.7.2.2.1 Short notation

The following single-line grammar may be used when the artifact's type and mime type can be inferred from the file URI:

```
<artifact_name>: <artifact_file_URI>
```

3.3.7.2.2.2 Extended notation:

The following multi-line grammar may be used when the artifact's definition's type and mime type need to be explicitly declared:

```
<artifact_name>:
  description: <artifact_description>
  type: <artifact_type_name>
  file: <artifact_file_URI>
  repository: <artifact_repository_name>
  deploy_path: <file_deployment_path>
  version: <artifact _version>
  checksum: <artifact_checksum>
  checksum_algorithm: <artifact_checksum_algorithm>
  properties: <property assignments>
```

In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

- artifact_name: represents the required symbolic name of the artifact as a string.
- artifact_description: represents the optional description for the artifact.
- artifact_type_name: represents the required artifact type the artifact definition is based upon.
- artifact_file_URI: represents the required URI string (relative or absolute) which can be used to locate the artifact's file.
- artifact_repository_name: represents the optional name of the repository definition to use to retrieve the associated artifact (file) from.
- file_deployement_path: represents the optional path the artifact_file_URI will be copied into within the target node's container.
- artifact_version: represents the version of artifact
- artifact_checksum: represents the checksum of the Artifact
- artifact_checksum_algorithm:represents the algorithm for verifying the checksum. Shall be specified if checksum is specified
- properties: represents an optional map of property assignments associated with the artifact

3.3.7.2.3 Examples

The following represents an artifact definition:

```
my_file_artifact: ../my_apps_files/operation_artifact.txt
```

The following example represents an artifact definition with property assignments:

```
artifacts:
  sw_image:
    description: Image for virtual machine
    type: tosca.artifacts.Deployment.Image.VM
```

```
        file:
http://10.10.86.141/images/Juniper_vSRX_15.1x49_D80_preconfigured.qcow2
        checksum: ba411cafee2f0f702572369da0b765e2
        version: 3.2
        checksum_algorithm: MD5
        properties:
          name: vSRX
          container_format: BARE
          disk_format: QCOW2
          min_disk: 1 GB
          size: 649 MB
```

3.4 Properties, Attributes, and Parameters

This section presents data handling in TOSCA via properties, attributes and parameters. The type of the values they contain can be divided in primitive types (either referenced from YAML or defined in TOSCA) or complex data types that can be defined themselves in the TOSCA service template.

3.4.1 Primitive types

This clause describes the primitive types that are used for declaring normative properties, parameters and grammar elements throughout this specification.

3.4.1.1 Referenced YAML Types

Many of the types we use in this profile are built-in types from the YAML 1.2 specification (i.e., those identified by the "tag:yaml.org,2002" version tag) [YAML-1.2].

The following table declares the valid YAML type URIs and aliases that SHALL be used when possible when defining parameters or properties within TOSCA Service Templates using this specification:

Valid aliases	Type URI
string	tag:yaml.org,2002:str (default)
integer	tag:yaml.org,2002:int
float	tag:yaml.org,2002:float
boolean	tag:yaml.org,2002:bool (i.e., a value either 'true' or 'false')
timestamp	tag:yaml.org,2002:timestamp [YAML-TS-1.1]
null	`tag:yaml.org,2002:null`

3.4.1.1.1 Notes

- The "string" type is the default type when not specified on a parameter or property declaration.
- While YAML supports further type aliases, such as "str" for "string", the TOSCA specification promotes the fully expressed alias name for clarity.

3.4.1.2 TOSCA version

TOSCA supports the concept of "reuse" of type definitions, as well as template definitions which could be version and change over time. It is important to provide a reliable, normative means to represent a version string which enables the comparison and management of types and templates over time. Therefore, the TOSCA TC intends to provide a normative version type (string) for this purpose in future Working Drafts of this specification.

Shorthand Name	version
Type Qualified Name	tosca:version

3.4.1.2.1 Grammar

TOSCA version strings have the following grammar:

<major_version>.<minor_version>[.<fix_version>[.<qualifier>[-<build_version]]]

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- major_version: is a required integer value greater than or equal to 0 (zero)
- minor_version: is a required integer value greater than or equal to 0 (zero).
- fix_version: is an optional integer value greater than or equal to 0 (zero).
- qualifier: is an optional string that indicates a named, pre-release version of the associated code that has been derived from the version of the code identified by the combination major_version, minor_version and fix_version numbers.
- build_version: is an optional integer value greater than or equal to 0 (zero) that can be used to further qualify different build versions of the code that has the same qualifer_string.

3.4.1.2.2 Version Comparison

- When comparing TOSCA versions, all component versions (i.e., *major*, *minor* and *fix*) are compared in sequence from left to right.
- TOSCA versions that include the optional qualifier are considered older than those without a qualifier.
- TOSCA versions with the same major, minor, and fix versions and have the same qualifier string, but with different build versions can be compared based upon the build version.
- Qualifier strings are considered domain-specific. Therefore, this specification makes no recommendation on how to compare TOSCA versions with the same major, minor and fix versions, but with different qualifiers strings and simply considers them different named branches derived from the same code.

3.4.1.2.3 Examples

Examples of valid TOSCA version strings:

```
# basic version strings
6.1
2.0.1

# version string with optional qualifier
3.1.0.beta

# version string with optional qualifier and build version
1.0.0.alpha-10
```

3.4.1.2.4 Notes

- [Maven-Version] The TOSCA version type is compatible with the Apache Maven versioning policy.

3.4.1.2.5 Additional Requirements

- A version value of zero (i.e., '0', '0.0', or '0.0.0') SHALL indicate there no version provided.
- A version value of zero used with any qualifiers SHALL NOT be valid.

3.4.1.3 TOSCA range type

The range type can be used to define numeric ranges with a lower and upper boundary. For example, this allows for specifying a range of ports to be opened in a firewall.

Shorthand Name	range
Type Qualified Name	tosca:range

3.4.1.3.1 Grammar

TOSCA range values have the following grammar:

```
[<lower_bound>, <upper_bound>]
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- lower_bound: is a required integer value that denotes the lower boundary of the range.
- upper_bound: is a required integer value that denotes the upper boundary of the range. This value MUST be greater than or equal to lower_bound.

3.4.1.3.2 Keywords

The following Keywords may be used in the TOSCA range type:

Keyword	Applicable Types	Description
UNBOUNDED	scalar	Used to represent an unbounded upper bounds (positive) value in a set for a scalar type.

3.4.1.3.3 Examples

Example of a node template property with a range value:

```
# numeric range between 1 and 100
a_range_property: [ 1, 100 ]

# a property that has allows any number 0 or greater
num_connections: [ 0, UNBOUNDED ]
```

3.4.1.4 TOSCA list type

The list type allows for specifying multiple values for a parameter of property. For example, if an application allows for being configured to listen on multiple ports, a list of ports could be configured using the list data type.

Note that entries in a list for one property or parameter must be of the same type. The type (for simple entries) or schema (for complex entries) is defined by the entry_schema attribute of the respective property definition, attribute definitions, or input or output parameter definitions.

Shorthand Name	list
Type Qualified Name	tosca:list

3.4.1.4.1 Grammar

TOSCA lists are essentially normal YAML lists with the following grammars:

3.4.1.4.1.1   Square bracket notation

```
[ <list_entry_1>, <list_entry_2>, ... ]
```

3.4.1.4.1.2   Bulleted list notation

```
- <list_entry_1>
- ...
- <list_entry_n>
```

In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

- <list_entry_*>: represents one entry of the list.

3.4.1.4.2 Declaration Examples

3.4.1.4.2.1   List declaration using a simple type

The following example shows a list declaration with an entry schema based upon a simple integer type (which has additional constraints):

```
<some_entity>:
  ...
  properties:
    listen_ports:
      type: list
      entry_schema:
        description: listen port entry (simple integer type)
        type: integer
        constraints:
          - max_length: 128
```

3.4.1.4.2.2   List declaration using a complex type

The following example shows a list declaration with an entry schema based upon a complex type:

```
<some_entity>:
  ...
  properties:
    products:
      type: list
      entry_schema:
        description: Product information entry (complex type) defined
elsewhere
        type: ProductInfo
```

3.4.1.4.3 Definition Examples

These examples show two notation options for defining lists:

- A single-line option which is useful for only short lists with simple entries.
- A multi-line option where each list entry is on a separate line; this option is typically useful or more readable if there is a large number of entries, or if the entries are complex.

3.4.1.4.3.1   Square bracket notation

```
listen_ports: [ 80, 8080 ]
```

3.4.1.4.3.2   Bulleted list notation

```
listen_ports:
  - 80
  - 8080
```

3.4.1.5 TOSCA map type

The map type allows for specifying multiple values for a parameter of property as a map. In contrast to the list type, where each entry can only be addressed by its index in the list, entries in a map are named elements that can be addressed by their keys.

Note that entries in a map for one property or parameter must be of the same type. The type (for simple entries) or schema (for complex entries) is defined by the entry_schema attribute of the respective property definition, attribute definition, or input or output parameter definition. In addition, the keys that identify entries in a map must be of the same type as well. The type of these keys is defined by the key_schema attribute of the respective property_definition, attribute_definition, or input or output parameter_definition. If the key_schema is not specified, keys are assumed to be of type string.

Shorthand Name	map
Type Qualified Name	tosca:map

3.4.1.5.1 Grammar

TOSCA maps are normal YAML dictionaries with following grammar:

3.4.1.5.1.1   Single-line grammar

```
{ <entry_key_1>: <entry_value_1>, ..., <entry_key_n>: <entry_value_n> }
```

3.4.1.5.1.2   Multi-line grammar

```
<entry_key_1>: <entry_value_1>
...
<entry_key_n>: <entry_value_n>
```

In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

- entry_key_*: is the required key for an entry in the map
- entry_value_*: is the value of the respective entry in the map

3.4.1.5.2 Declaration Examples

3.4.1.5.2.1   Map declaration using a simple type

The following example shows a map with an entry schema definition based upon an existing string type (which has additional constraints):

```
<some_entity>:
  ...
  properties:
    emails:
      type: map
      entry_schema:
        description: basic email address
        type: string
        constraints:
```

```
                - max_length: 128
```

3.4.1.5.2.2   Map declaration using a complex type

The following example shows a map with an entry schema definition for contact information:

```
<some_entity>:
  ...
  properties:
    contacts:
       type: map
       entry_schema:
          description: simple contact information
          type: ContactInfo
```

3.4.1.5.3 Definition Examples

These examples show two notation options for defining maps:

- A single-line option which is useful for only short maps with simple entries.
- A multi-line option where each map entry is on a separate line; this option is typically useful or more readable if there is a large number of entries, or if the entries are complex.

3.4.1.5.3.1   Single-line notation

```
# notation option for shorter maps
user_name_to_id_map: { user1: 1001, user2: 1002 }
```

3.4.1.5.3.2   Multi-line notation

```
# notation for longer maps
user_name_to_id_map:
  user1: 1001
  user2: 1002
```

3.4.1.6 TOSCA scalar-unit type

The scalar-unit type can be used to define scalar values along with a unit from the list of recognized units provided below.

3.4.1.6.1 Grammar

TOSCA scalar-unit typed values have the following grammar:

```
<scalar> <unit>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- scalar: is a required scalar value.
- unit: is a required unit value. The unit value MUST be type-compatible with the scalar.

3.4.1.6.2 Additional requirements

- **Whitespace**: any number of spaces (including zero or none) SHALL be allowed between the scalar value and the unit value.
- It SHALL be considered an error if either the scalar or unit portion is missing on a property or attribute declaration derived from any scalar-unit type.

- When performing constraint clause evaluation on values of the scalar-unit type, both the scalar value portion and unit value portion **SHALL** be compared together (i.e., both are treated as a single value). For example, if we have a property called storage_size (which is of type scalar-unit) a valid range constraint would appear as follows:

  – storage_size: in_range [4 GB, 20 GB]

where storage_size's range will be evaluated using both the numeric and unit values (combined together), in this case '4 GB' and '20 GB'.

3.4.1.6.3 Concrete Types

Shorthand Names	scalar-unit.size, scalar-unit.time, scalar-unit.frequency, scalar-unit.bitrate
Type Qualified Names	tosca:scalar-unit.size, tosca:scalar-unit.time

The scalar-unit type grammar is abstract and has four recognized concrete types in TOSCA:

- **scalar-unit.size** – used to define properties that have scalar values measured in size units.
- **scalar-unit.time** – used to define properties that have scalar values measured in size units.
- **scalar-unit.frequency** – used to define properties that have scalar values measured in units per second.
- **scalar-unit.bitrate** – used to define properties that have scalar values measured in bits or bytes per second

These types and their allowed unit values are defined below.

3.4.1.6.4 scalar-unit.size

3.4.1.6.4.1 Recognized Units

Unit	Usage	Description
B	size	byte
kB	size	kilobyte (1000 bytes)
KiB	size	kibibytes (1024 bytes)
MB	size	megabyte (1000000 bytes)
MiB	size	mebibyte (1048576 bytes)
GB	size	gigabyte (1000000000 bytes)
GiB	size	gibibytes (1073741824 bytes)
TB	size	terabyte (1000000000000 bytes)
TiB	size	tebibyte (1099511627776 bytes)

3.4.1.6.4.2 Examples

```
# Storage size in Gigabytes
```

```
      properties:
        storage_size: 10 GB
```

3.4.1.6.4.3   Notes

- The unit values recognized by TOSCA for size-type units are based upon a subset of those defined by GNU at http://www.gnu.org/software/parted/manual/html_node/unit.html, which is a non-normative reference to this specification.

- TOSCA treats these unit values as case-insensitive (e.g., a value of 'kB', 'KB' or 'kb' is equivalent), but it is considered best practice to use the case of these units as prescribed by GNU.

- Some cloud providers may not support byte-level granularity for storage size allocations. In those cases, these values could be treated as desired sizes and actual allocations will be based upon individual provider capabilities.

3.4.1.6.5 scalar-unit.time

3.4.1.6.5.1   Recognized Units

Unit	Usage	Description
d	time	days
h	time	hours
m	time	minutes
s	time	seconds
ms	time	milliseconds
us	time	microseconds
ns	time	nanoseconds

3.4.1.6.5.2   Examples

```
# Response time in milliseconds
properties:
  respone_time: 10 ms
```

3.4.1.6.5.3   Notes

- The unit values recognized by TOSCA for time-type units are based upon a subset of those defined by International System of Units whose recognized abbreviations are defined within the following reference:

  – http://www.ewh.ieee.org/soc/ias/pub-dept/abbreviation.pdf

  – This document is a non-normative reference to this specification and intended for publications or grammars enabled for Latin characters which are not accessible in typical programming languages

3.4.1.6.6 scalar-unit.frequency

3.4.1.6.6.1 Recognized Units

Unit	Usage	Description
Hz	frequency	Hertz, or Hz. equals one cycle per second.
kHz	frequency	Kilohertz, or kHz, equals to 1,000 Hertz
MHz	frequency	Megahertz, or MHz, equals to 1,000,000 Hertz or 1,000 kHz
GHz	frequency	Gigahertz, or GHz, equals to 1,000,000,000 Hertz, or 1,000,000 kHz, or 1,000 MHz.

3.4.1.6.6.2 Examples

```
# Processor raw clock rate
properties:
   clock_rate: 2.4 GHz
```

3.4.1.6.6.3 Notes

- The value for Hertz (Hz) is the International Standard Unit (ISU) as described by the Bureau International des Poids et Mesures (BIPM) in the "*SI Brochure: The International System of Units (SI) [8th edition, 2006; updated in 2014]*", http://www.bipm.org/en/publications/si-brochure/

3.4.1.6.7 scalar-unit.bitrate

3.4.1.6.7.1 Recognized Units

Unit	Usage	Description
bps	bitrate	bit per second
Kbps	bitrate	kilobit (1000 bits) per second
Kibps	bitrate	kibibits (1024 bits) per second
Mbps	bitrate	megabit (1000000 bits) per second
Mibps	bitrate	mebibit (1048576 bits) per second
Gbps	bitrate	gigabit (1000000000 bits) per second
Gibps	bitrate	gibibits (1073741824 bits) per second
Tbps	bitrate	terabit (1000000000000 bits) per second
Tibps	bitrate	tebibits (1099511627776 bits) per second
Bps	bitrate	byte per second
KBps	bitrate	kilobyte (1000 bytes) per second
KiBps	bitrate	kibibytes (1024 bytes) per second
MBps	bitrate	megabyte (1000000 bytes) per second
MiBps	bitrate	mebibyte (1048576 bytes) per second
GBps	bitrate	gigabyte (1000000000 bytes) per second
GiBps	bitrate	gibibytes (1073741824 bytes) per second

Unit	Usage	Description
TBps	bitrate	terabytes (1000000000000 bits) per second
TiBps	bitrate	tebibytes (1099511627776 bits) per second

3.4.1.6.7.2 Examples

```
#### Somewhere in a node template definition
requirements:
    - link:
        node_filter:
           capabilities:
             - myLinkable
                 properties:
                   bitrate:
                     - greater_or_equal: 10 Kbps # 10 * 1000 bits per second
at least
```

3.4.1.6.7.3 Notes

- Unlike with the scalar-unit.size type, TOSCA treats scalar-unit.bitrate values as case-sensitive (e.g., a value of 'KBs' means kilobyte per second, whereas 'Kb' means kilobit per second).
- For comparison purposes, 1 byte is the same as 8 bits.

3.4.2 Data Type

A Data Type definition defines the schema for new named datatypes in TOSCA.

3.4.2.1 Keynames

The Data Type is a TOSCA type entity and has the common keynames listed in Section 3.2.5.2 Common keynames in type definitions. In addition, the Data Type has the following recognized keynames:

Keyname	Required	Type	Description
constraints	no	list of constraint clauses	The optional list of sequenced constraint clauses for the Data Type.
properties	no	map of property definitions	The optional map property definitions that comprise the schema for a complex Data Type in TOSCA.
key_schema	no	schema definition	For data types that derive from the TOSCA map data type, the optional schema definition for the keys used to identify entries in properties of this data type.
entry_schema	no	schema definition	For data types that derive from the TOSCA map or list data types, the optional schema definition for the entries in properties of this data type.

3.4.2.2 Grammar

Data Types have the following grammar:

```
<data_type_name>:
  derived_from: <existing_type_name>
  version: <version_number>
  metadata:
```

```
   <map of string>
description: <datatype_description>
constraints:
  - <type_constraints>
properties:
  <property_definitions>
key_schema: <key_schema_definition>
entry_schema: <entry_schema_definition>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- data_type_name: represents the required symbolic name of the data type as a string.
- version_number: represents the optional TOSCA version number for the data type.
- datatype_description: represents the optional description for the data type.
- existing_type_name: represents the optional name of a valid TOSCA primitive type or data type this new data type derives from.
- type_constraints: represents the optional list of one or more type-compatible constraint clauses that restrict the data type.
- property_definitions: represents the optional map of one or more property definitions that provide the schema for the data type
  - property_definitions may not be added to data types derived_from TOSCA primitive types.
- key_schema_definition: if the data type derives from the TOSCA map type (i.e existing_type_name is a map or derives from a map), it represents the optional schema definition for the keys used to identify entry properties of this type.
- entry_schema_definition: if the data type derives from the TOSCA map or list types (i.e. existing_type name is a map or list or derives from a map or list), it represents the optional schema definition for the entries in properties of this type.

3.4.2.3 Derivation rules

During data type derivation the keyname definitions follow these rules:

- constraints: new constraints may be defined; these constraints do not replace the constraints defined in the parent type but are considered in addition to them.
- properties: existing property definitions may be refined; new property definitions may be added.
- key_schema: the key_schema definition may be refined according to schema refinement rules.
- entry_schema: the entry_schema definition may be refined according to schema refinement rules.

3.4.2.4 Additional Requirements

- A valid datatype definition **MUST** have either a valid derived_from declaration or at least one valid property definition.
- Any constraint clauses **SHALL** be type-compatible with the type declared by the derived_from keyname.
- If a properties keyname is provided, it **SHALL** contain one or more valid property definitions.
- Property definitions may not be added to data types derived from TOSCA primitive types.

3.4.2.5 Examples

The following example represents a Data Type definition based upon an existing string type:

3.4.2.5.1 Defining a complex datatype

```
# define a new complex datatype
mytypes.phonenumber:
```

```
          description: my phone number datatype
        properties:
          countrycode:
            type: integer
          areacode:
            type: integer
          number:
            type: integer
```

3.4.2.5.2 Defining a datatype derived from an existing datatype

```
# define a new datatype that derives from existing type and extends it
mytypes.phonenumber.extended:
  derived_from: mytypes.phonenumber
  description: custom phone number type that extends the basic phonenumber
type
  properties:
    phone_description:
      type: string
      constraints:
        - max_length: 128
```

3.4.3 Schema definition

All entries in a map or list for one property or parameter must be of the same type. Similarly, all keys for map entries for one property or parameter must be of the same type as well. A TOSCA schema definition specifies the type (for simple entries) or schema (for complex entries) for keys and entries in TOSCA set types such as the TOSCA list or map.

If the schema definition specifies a map key, then the type of the schema must be derived originally from the string type (which basically ensures that the schema type is a string with additional constraints). As there is little need for complex keys this caters to more straight-forward and clear specifications.

Schema definitions appear in data type definitions when derived_from a map or list type or in parameter, property, or attribute definitions of a map or list type.

3.4.3.1 Keynames

The following is the list of recognized keynames for a TOSCA schema definition:

Keyname	Required	Type	Description
type	yes	string	The required data type for the key or entry. If this schema definition is for a map key, then the referred type must be derived originally from string.
description	no	description	The optional description for the schema.
constraints	no	list of constraint clauses	The optional list of sequenced constraint clauses for the property.
key_schema	no	schema definition	When the schema itself is of type map, the optional schema definition that is used to specify the type of the keys of that map's entries.
entry_schema	no	schema definition	When the schema itself is of type map or list, the optional schema definition that is used to specify the type of the entries in that map or list

3.4.3.2 Grammar

Schema definitions have the following grammar:

```
<schema_definition>:
  type: <schema_type>
  description: <schema_description>
  constraints:
    - <schema_constraints>
  key_schema : <key_schema_definition>
  entry_schema: <entry_schema_definition>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

* schema_description: represents the optional description of the schema definition
* schema_type: represents the required type name for entries of the specified schema.
* schema_constraints: represents the optional list of one or more constraint clauses on on entries of the specified schema.
* key_schema_definition: if the schema_type is map, it represents the optional schema definition for the keys of that map's entries.
* entry_schema_definition: if the schema_type is map or list, it represents the optional schema definition for the entries in that map or list.

3.4.3.3 Refinement rules

A schema definition uses the following definition refinement rules when the containing entity type is derived:

* type: must be derived from (or the same as) the type in the schema definition in the parent entity type definition.
* description: a new definition is unrestricted and will overwrite the one inherited from the schema definition in the parent entity type definition.
* constraints: a new definition is unrestricted; these constraints do not replace the constraints defined in the schema definition in the parent entity type but are considered in addition to them.

* key_schema: may be refined (recursively) according to schema refinement rules.
* entry_schema: may be refined (recursively) according to schema refinement rules.

3.4.4 Constraint clause definition

A constraint clause defines an operation along with one or more compatible values that can be used to define a constraint on a property or parameter's allowed values when it is defined in a TOSCA Service Template or one of its entities.

3.4.4.1 Operator keynames

The following is the list of recognized operators (keynames) when defining constraint clauses:

Operator	Type	Value Type	Description
equal	scalar	any	Constrains a property or parameter to a value equal to ('=') the value declared.
greater_than	scalar	comparable	Constrains a property or parameter to a value greater than ('>') the value declared.
greater_or_equal	scalar	comparable	Constrains a property or parameter to a value greater than or equal to ('>=') the value declared.

Operator	Type	Value Type	Description
less_than	scalar	comparable	Constrains a property or parameter to a value less than ('<') the value declared.
less_or_equal	scalar	comparable	Constrains a property or parameter to a value less than or equal to ('<=') the value declared.
in_range	dual scalar	comparable, range	Constrains a property or parameter to a value in range of (inclusive) the two values declared. Note: subclasses or templates of types that declare a property with the in_range constraint MAY only further restrict the range specified by the parent type.
valid_values	list	any	Constrains a property or parameter to a value that is in the list of declared values.
length	scalar	string, list, map	Constrains the property or parameter to a value of a given length.
min_length	scalar	string, list, map	Constrains the property or parameter to a value to a minimum length.
max_length	scalar	string, list, map	Constrains the property or parameter to a value to a maximum length.
pattern	regex	string	Constrains the property or parameter to a value that is allowed by the provided regular expression. **Note**: Future drafts of this specification will detail the use of regular expressions and reference an appropriate standardized grammar.
schema	string	string	Constrains the property or parameter to a value that is allowed by the referenced schema.

3.4.4.1.1 Comparable value types

In the Value Type column above, an entry of "comparable" includes integer, float, timestamp, string, version, and scalar-unit types while an entry of "*any*" refers to any type allowed in the TOSCA.

3.4.4.2 Schema Constraint purpose

TOSCA recognizes that there are external data-interchange formats that are widely used within Cloud service APIs and messaging (e.g., JSON, XML, etc.).

The 'schema' Constraint was added so that, when TOSCA types utilize types from these externally defined data (interchange) formats on Properties or Parameters, their corresponding Property definitions' values can be optionally validated by TOSCA Orchestrators using the schema string provided on this operator.

3.4.4.3 Additional Requirements

- If no operator is present for a simple scalar-value on a constraint clause, it **SHALL** be interpreted as being equivalent to having the "equal" operator provided; however, the "equal" operator may be used for clarity when expressing a constraint clause.
- The "length" operator **SHALL** be interpreted mean "size" for set types (i.e., list, map, etc.).
- Values provided by the operands (i.e., values and scalar values) **SHALL** be type-compatible with their associated operations.
- Future drafts of this specification will detail the use of regular expressions and reference an appropriate standardized grammar.

- The value for the keyname 'schema' SHOULD be a string that contains a valid external schema definition that matches the corresponding Property definitions type.
    - When a valid 'schema' value is provided on a Property definition, a TOSCA Orchestrator MAY choose to use the contained schema definition for validation.

3.4.4.4 Grammar

Constraint clauses have one of the following grammars:

```
# Scalar grammar
<operator>: <scalar_value>

# Dual scalar grammar
<operator>: [ <scalar_value_1>, <scalar_value_2> ]

# List grammar
<operator>: [ <value_1>, <value_2>, ..., <value_n> ]

# Regular expression (regex) grammar
pattern: <regular_expression_value>

# Schema grammar
schema: <schema_definition>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- operator: represents a required operator from the specified list shown above in section "Operator keynames".
- scalar_value, scalar_value_*: represents a required scalar (or atomic quantity) that can hold only one value at a time. This will be a value of a primitive type, such as an integer or string that is allowed by this specification.
- value_*: represents a required value of the operator that is not limited to scalars.
- reqular_expression_value: represents a regular expression (string) value.
- schema_definition: represents a schema definition as a string.

3.4.4.5 Examples

Constraint clauses used on parameter or property definitions:

```
# equal
equal: 2

# greater_than
greater_than: 1

# greater_or_equal
greater_or_equal: 2

# less_than
less_than: 5

# less_or_equal
less_or_equal: 4

# in_range
in_range: [ 1, 4 ]

# valid_values
valid_values: [ 1, 2, 4 ]
# specific length (in characters)
```

```
        length: 32

        # min_length (in characters)
        min_length: 8

        # max_length (in characters)
        max_length: 64

        # schema
        schema: <
          {
              # Some schema syntax that matches corresponding property or parameter.
          }
```

3.4.5 Property definition

A property definition defines a named, typed value and related data that can be associated with an entity defined in this specification (e.g., Node Types, Relationship Types, Capability Types, etc.). Properties are used by template authors to provide input values to TOSCA entities which indicate their "desired state" when they are instantiated. The value of a property can be retrieved using the get_property function within TOSCA Service Templates.

3.4.5.1 Attribute and Property reflection

The actual state of the entity, at any point in its lifecycle once instantiated, is reflected by an attribute. TOSCA orchestrators automatically create an attribute for every declared property (with the same symbolic name) to allow introspection of both the desired state (property) and actual state (attribute). If an attribute is reflected from a property, its initial value is the value of the reflected property.

3.4.5.2 Keynames

The following is the list of recognized keynames for a TOSCA property definition:

Keyname	Required	Type	Constraints	Description
type	yes	string	None	The required data type for the property.
description	no	description	None	The optional description for the property.
required	no	boolean	default: true	An optional key that declares a property as required (true) or not (false).
default	no	<any>	None	An optional key that may provide a value to be used as a default if not provided by another means.
value	no	<any>	None	An optional key that may provide a fixed value to be used. A property that has a fixed value provided (as part of a definition or refinement) cannot be subject to a further refinement or assignment. That is, a fixed value cannot be changed.
status	no	string	default: supported	The optional status of the property relative to the specification or implementation. See table below for valid values.
constraints	no	list of constraint clauses	None	The optional list of sequenced constraint clauses for the property.

Keyname	Required	Type	Constraints	Description
key_schema	no	schema definition	None	The optional schema definition for the keys used to identify entries in properties of type TOSCA map.
entry_schema	no	schema definition	None	The optional schema definition for the entries in properties of TOSCA set types such as list or map.
external-schema	no	string	None	The optional key that contains a schema definition that TOSCA Orchestrators MAY use for validation when the "type" key's value indicates an External schema (e.g., "json"). See section "External schema" below for further explanation and usage.
metadata	no	map of string	None	Defines a section used to declare additional metadata information.

3.4.5.3 Status values

The following property status values are supported:

Value	Description
supported	Indicates the property is supported. This is the **default** value for all property definitions.
unsupported	Indicates the property is not supported.
experimental	Indicates the property is experimental and has no official standing.
deprecated	Indicates the property has been deprecated by a new specification version.

3.4.5.4 Grammar

Property definitions have the following grammar:

```
<property_name>:
  type: <property_type>
  description: <property_description>
  required: <property_required>
  default: <default_value>
  value: <fixed_value>
  status: <status_value>
  constraints:
    - <property_constraints>
  key_schema : <key_schema_definition>
  entry_schema: <entry_schema_definition>
  metadata:
    <metadata_map>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- property_name: represents the required symbolic name of the property as a string.
- property_description: represents the optional description of the property.
- property_type: represents the required data type of the property.
- property_required: represents an optional boolean value (true or false) indicating whether or not the property is required. If this keyname is not present on a property definition, then the property SHALL be considered **required** (i.e., true) by **default**.
- default_value: contains a type-compatible value that is used as a default value if a value is not provided by another means (via the fixed_value definition or via property assignment);

- the default_value is irrelevant for properties that are not required (i.e. property_required is "false") as they will stay undefined.
- fixed_value: contains a type-compatible value that may be defined during property definition or refinement to set and fix the value of the property
  - note that a fixed value cannot be changed; once defined, the property cannot be further refined or assigned. Thus, fixed value definitions should be avoided in data_type definitions.
- status_value: a string that contains a keyword that indicates the status of the property relative to the specification or implementation.
- property_constraints: represents the optional list of one or more sequenced constraint clauses on the property definition.
- key_schema_definition: if the property_type is map, represents the optional schema definition for the keys used to identify entries in that map.
- entry_schema_definition: if the property_type is map or list, represents the optional schema definition for the entries in that map or list.
- metadata_map: represents the optional map of string.

3.4.5.5 Additional Requirements

- Implementations of TOSCA **SHALL** automatically reflect (i.e., make available) any property defined on an entity as an attribute of the entity with the same name as the property.
- A property **SHALL** be considered required by default (i.e., as if the required keyname on the definition is set to true) unless the definition's required keyname is explicitly set to false.
- The value provided on a property definition's default keyname SHALL be type compatible with the type declared on the definition's type keyname.
- Constraints of a property definition **SHALL** be type-compatible with the type defined for that definition.
- If a key_schema or entry_schema keyname is provided, its value (string) MUST represent a valid schema definition that matches the property type (i.e. the property type as defined by the type keyword must be the same as or derived originally from map (for key_schema) or map or list (for entry_schema).
- TOSCA Orchestrators MAY choose to validate the value of the 'schema' keyname in accordance with the corresponding schema specifcation for any recognized external types.

3.4.5.6 Refinement rules

A property definition within data, capability, node, relationship, group, policy, and artifact types (including capability definitions in node types) uses the following refinement rules when the containing entity type is derived:

- type: must be derived from (or the same as) the type in the property definition in the parent entity type definition.
- description: a new definition is unrestricted and will overwrite the one inherited from the property definition in the parent entity type definition.
- required: if defined to "false" in the property definition parent entity type it may be redefined to "true"; note that if undefined it is automatically considered as being defined to "true".
- default: a new definition is unrestricted and will overwrite the one inherited from the property definition in the parent entity type definition.
- value: if undefined in the property definition in the parent entity type, it may be defined to any type-compatible value; once defined, the property cannot be further refined or assigned.
- status: a new definition is unrestricted and will overwrite the one inherited from the property definition in the parent entity type definition.
- constraints: a new definition is unrestricted; these constraints do not replace the constraints defined in the property definition in the parent entity type but are considered in addition to them.

- key_schema: if defined in the property definition in the parent entity type it may be refined according to schema refinement rules.
- entry_schema: if defined in the property definition in the parent entity type it may be refined according to schema refinement rules.
- metadata: a new definition is unrestricted and will overwrite the one inherited from the property definition in the parent entity type definition.

3.4.5.7 Examples

The following represents an example of a property definition with constraints:

```
properties:
  num_cpus:
    type: integer
    description: Number of CPUs requested for a software node instance.
    default: 1
    required: true
    constraints:
      - valid_values: [ 1, 2, 4, 8 ]
```

The following shows an example of a property refinement. Consider the definition of an Endpoint capability type:

```
tosca.capabilities.Endpoint:
  derived_from: tosca.capabilities.Root
  properties:
    protocol:
      type: string
      required: true
      default: tcp
    port:
      type: PortDef
      required: false
    secure:
      type: boolean
      required: false
      default: false
    # Other property definitions omitted for brevity
```

The Endpoint.Admin capability type refines the secure property of the Endpoint capability type from which it derives by forcing its value to always be true:

```
tosca.capabilities.Endpoint.Admin:
  derived_from: tosca.capabilities.Endpoint
  # Change Endpoint secure indicator to true from its default of false
  properties:
    secure: true
```

3.4.6 Property assignment

This section defines the grammar for assigning values to named properties within TOSCA templates.

3.4.6.1 Keynames

The TOSCA property assignment has no keynames.

3.4.6.2 Grammar

Property assignments have the following grammar:

3.4.6.2.1 Short notation:

The following single-line grammar may be used when a simple value assignment is needed:

```
<property_name>: <property_value> | { <property_value_expression> }
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- property_name: represents the name of a property that will be used to select a property definition with the same name within on a TOSCA entity (e.g., Node Template, Relationship Template, etc.) which is declared in its declared type (e.g., a Node Type, Node Template, Capability Type, etc.).
- property_value, property_value_expression: represent the type-compatible value to assign to the property. Property values may be provided as the result from the evaluation of an expression or a function.

3.4.6.3 Additional Requirements

- Properties that have a (fixed) value defined during their definition or during a subsequent refinement may not be assigned (as their value is already set).
- If a required property has no value defined or assigned, its default value is assigned
- A non-required property that is not assigned it stays undefined, thus the default keyname is irrelevant for a non-required property.

3.4.7 Attribute definition

An attribute definition defines a named, typed value that can be associated with an entity defined in this specification (e.g., a Node, Relationship or Capability Type). Specifically, it is used to expose the "actual state" of some property of a TOSCA entity after it has been deployed and instantiated (as set by the TOSCA orchestrator). Attribute values can be retrieved via the get_attribute function from the instance model and used as values to other entities within TOSCA Service Templates.

3.4.7.1 Attribute and Property reflection

The actual state of the entity, at any point in its lifecycle once instantiated, is reflected by an attribute. TOSCA orchestrators automatically create an attribute for every declared property (with the same symbolic name) to allow introspection of both the desired state (property) and actual state (attribute). If an attribute is reflected from a property, its initial value is the value of the reflected property.

3.4.7.2 Keynames

The following is the list of recognized keynames for a TOSCA attribute definition:

Keyname	Required	Type	Constraints	Description
type	yes	string	None	The required data type for the attribute.
description	no	description	None	The optional description for the attribute.

Keyname	Required	Type	Constraints	Description
default	no	<any>	None	An optional key that may provide a value to be used as a default if not provided by another means. This value SHALL be type compatible with the type declared by the property definition's type keyname.
status	no	string	default: supported	The optional status of the attribute relative to the specification or implementation. See supported status values .
key_schema	No	schema definition	None	The optional schema definition for the keys used to identify entries in attributes of type TOSCA map.
entry_schema	no	schema definition	None	The optional schema definition for the entries in attributes of TOSCA set types such as list or map.
metadata	no	map of string	None	Defines a section used to declare additional metadata information.

3.4.7.3 Grammar

Attribute definitions have the following grammar:

```
attributes:
  <attribute_name>:
    type: <attribute_type>
    description: <attribute_description>
    default: <default_value>
    status: <status_value>
    key_schema : <key_schema_definition>
    entry_schema: <entry_schema_definition>
    metadata:
      <metadata_map>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- attribute_name: represents the required symbolic name of the attribute as a string.
- attribute_type: represents the required data type of the attribute.
- attribute_description: represents the optional description of the attribute.
- default_value: contains a type-compatible value that may be used as a default if not provided by another means.
- status_value: contains a value indicating the attribute's status relative to the specification version (e.g., supported, deprecated, etc.); supported status values for this keyname are defined in the property definition section.
- key_schema_definition: if the attribute_type is map, represents the optional schema definition for the keys used to identify entries in that map.
- entry_schema_definition: if the attribute_type is map or list, represents the optional schema definition for the entries in that map or list.
- metadata_map: represents the optional map of string.

3.4.7.4 Refinement rules

An attribute definition within data, capability, node, relationship, and group types (including capability definitions in node types) uses the following refinement rules when the containing entity type is derived:

- type: must be derived from (or the same as) the type in the property definition in the parent entity type definition.
- description: a new definition is unrestricted and will overwrite the one inherited from the property definition in the parent entity type definition.
- default: a new definition is unrestricted and will overwrite the one inherited from the property definition in the parent entity type definition.
- status: a new definition is unrestricted and will overwrite the one inherited from the property definition in the parent entity type definition.
- key_schema: if defined in the property definition in the parent entity type it may be refined according to schema refinement rules.
- entry_schema: if defined in the property definition in the parent entity type it may be refined according to schema refinement rules.
- metadata: a new definition is unrestricted and will overwrite the one inherited from the property definition in the parent entity type definition

3.4.7.5 Additional Requirements

- In addition to any explicitly defined attributes on a TOSCA entity (e.g., Node Type, Relationship Type, etc.), implementations of TOSCA **MUST** automatically reflect (i.e., make available) any property defined on an entity as an attribute of the entity with the same name as the property.
- Values for the default keyname **MUST** be derived or calculated from other attribute or operation output values (that reflect the actual state of the instance of the corresponding resource) and not hard-coded or derived from a property settings or inputs (i.e., desired state).

3.4.7.6 Notes

- Attribute definitions are very similar to Property definitions; however, properties of entities reflect an input that carries the template author's requested or desired value (i.e., desired state) which the orchestrator (attempts to) use when instantiating the entity whereas attributes reflect the actual value (i.e., actual state) that provides the actual instantiated value.
  - For example, a property can be used to request the IP address of a node using a property (setting); however, the actual IP address after the node is instantiated may by different and made available by an attribute.

3.4.7.7 Example

The following represents a required attribute definition:

```
actual_cpus:
  type: integer
  description: Actual number of CPUs allocated to the node instance.
```

3.4.8 Attribute assignment

This section defines the grammar for assigning values to named attributes within TOSCA templates.

3.4.8.1 Keynames

The TOSCA attribute assignment has no keynames.

3.4.8.2 Grammar

Attribute assignments have the following grammar:

3.4.8.2.1 Short notation:

The following single-line grammar may be used when a simple value assignment is needed:

```
<attribute_name>: <attribute_value> | { <attribute_value_expression> }
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- attribute_name: represents the name of an attribute that will be used to select an attribute definition with the same name within on a TOSCA entity (e.g., Node Template, Relationship Template, etc.) which is declared (or reflected from a Property definition) in its declared type (e.g., a Node Type, Node Template, Capability Type, etc.).
- attribute_value, attribute_value_expresssion: represent the type-compatible value to assign to the attribute. Attribute values may be provided as the result from the evaluation of an expression or a function.

3.4.8.3 Additional requirements

- Attribute values **MAY** be provided by the underlying implementation at runtime when requested by the get_attribute function or it **MAY** be provided through the evaluation of expressions and/or functions that derive the values from other TOSCA attributes (also at runtime).

3.4.9 Parameter definition

A parameter definition defines a named, typed value and related data and may be used to exchange values between the TOSCA orchestrator and the external world. Such values may be

- inputs and outputs of interface operations and notifications
- inputs and outputs of workflows
- inputs and outputs of service templates

From the perspective of the TOSCA orchestrator such parameters are either "incoming" (i.e. transferring a value from the external world to the orchestrator) or "outgoing" (transferring a value from the orchestrator to the external world). Thus:

- outgoing parameters are:
    - template outputs
    - internal workflow outputs
    - external workflow inputs
    - operation inputs
- incoming parameters are:
    - template inputs
    - internal workflow inputs
    - external workflow outputs
    - operation outputs
    - notification outputs

An "outgoing" parameter definition is essentially the same as a TOSCA property definition, however it may optionally inherit the data type of the value assigned to it rather than have an explicit data type defined.

An "incoming" parameter definition may define an attribute mapping of the parameter value to an attribute of a node. Optionally, it may inherit the data type of the attribute it is mapped to, rather than have an explicit data type defined for it.

3.4.9.1 Keynames

The TOSCA parameter definition has all the keynames of a TOSCA property definition with the following additional or changed keynames:

Keyname	Required	Type	Description
type	no	string	The data type of the parameter. **Note**: This keyname is required for a TOSCA Property definition but is not required for a TOSCA Parameter definition.
value	no	<any>	The type-compatible value to assign to the parameter. Parameter values may be provided as the result from the evaluation of an expression or a function. May only be defined for outgoing parameters. Mutually exclusive with the "mapping" keyname.
mapping	no	attribute selection format	A mapping that specifies the node or relationship attribute into which the returned output value must be stored. May only be defined for incoming parameters. Mutually exclusive with the "value" keyname.

3.4.9.2 Grammar

Parameter definitions have the following grammar:

```
<parameter_name>:
  type: <parameter_type>
  description: <parameter_description>
  value: <parameter_value> | { <parameter_value_expression> }
  required: <parameter_required>
  default: <parameter_default_value>
  status: <status_value>
  constraints:
    - <parameter_constraints>
  key_schema : <key_schema_definition>
  entry_schema: <entry_schema_definition>
  mapping: <attribute_selection_form>
```

In addition, the following single-line grammar is supported when only a fixed value needs to be provided:

```
<parameter_name>: <parameter_value> | { <parameter_value_expression> }
```

This single-line grammar is equivalent to the following:

```
<parameter_name>:
    value : <parameter_value> | { <parameter_value_expression> }
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- parameter_name: represents the required symbolic name of the parameter as a string.
- parameter_description: represents the optional description of the parameter.
- parameter_type: represents the optional data type of the parameter. Note, this keyname is required for a TOSCA Property definition, but is not for a TOSCA Parameter definition.

- parameter_value, parameter_value_expresssion: represent the type-compatible value to assign to the parameter. Parameter values may be provided as the result from the evaluation of an expression or a function.
  - once the value keyname is defined, the property cannot be further refined or assigned.
  - the value keyname is relevant only for "outgoing" parameter definitions and SHOULD NOT be defined in "incoming" parameter definitions.
- parameter_required: represents an optional boolean value (true or false) indicating whether or not the parameter is required. If this keyname is not present on a parameter definition, then the property SHALL be considered **required** (i.e., true) by **default**.
- default_value: contains a type-compatible value that may be used as a default if not provided by other means.
  - the default keyname is irrelevant for properties that are not required (i.e. property_required is "false") as they will stay undefined.
- status_value: a string that contains a keyword that indicates the status of the parameter relative to the specification or implementation.
- parameter_constraints: represents the optional list of one or more sequenced constraint clauses on the parameter definition.
- key_schema_definition: if the parameter_type is map, represents the optional schema definition for the keys used to identify entries in that map.
- entry_schema_definition: if the parameter_type is map or list, represents the optional schema definition for the entries in that map or list.
- attribute_selection_form: a list that corresponds to a valid attribute_selection_format; the parameter is mapped onto an attribute of the containing entity
  - the mapping keyname is relevant only for "incoming" parameter definitions and SHOULD NOT be defined in "outgoing" parameter definitions.

3.4.9.3 Additional Requirements

- A parameter **SHALL** be considered <u>required by default</u> (i.e., as if the required keyname on the definition is set to true) unless the definition's required keyname is explicitly set to false.
- The value provided on a parameter definition's default keyname **SHALL** be type compatible with the type declared on the definition's type keyname.
- Constraints of a parameter definition **SHALL** be type-compatible with the type defined for that definition.

3.4.9.4 Example

The following represents an example of an input parameter definition with constraints:

```
inputs:
  cpus:
    type: integer
    description: Number of CPUs for the server.
    constraints:
      - valid_values: [ 1, 2, 4, 8 ]
```

The following represents an example of an (untyped) output parameter definition:

```
outputs:
  server_ip:
    description: The private IP address of the provisioned server.
    value: { get_attribute: [ my_server, private_address ] }
```

3.4.9.5 Refinement rules

A parameter definition within interface types, interface definitions in node and relationship types, uses the following refinement rules when the containing entity type is derived:

- type: must be derived from (or the same as) the type in the parameter definition in the parent entity type definition.

- description: a new definition is unrestricted and will overwrite the one inherited from the parameter definition in the parent entity type definition.

- required: if defined to "false" in the parameter definition parent entity type it may be redefined to "true"; note that if undefined it is automatically considered as being defined to "true".

- default: a new definition is unrestricted and will overwrite the one inherited from the parameter definition in the parent entity type definition.

- value: if undefined in the parameter definition in the parent entity type, it may be defined to any type-compatible value; once defined, the parameter cannot be further refined or assigned
    - the value keyname should be defined only for "outgoing" parameters.

- mapping: if undefined in the parameter definition in the parent entity type, it may be defined to any type-compatible attribute mapping; once defined, the parameter cannot be further refined or mapped
    - the value keyname should be defined only for "incoming" parameters.

- status: a new definition is unrestricted and will overwrite the one inherited from the parameter definition in the parent entity type definition.

- constraints: a new definition is unrestricted; these constraints do not replace the constraints defined in the parameter definition in the parent entity type but are considered in addition to them.

- key_schema: if defined in the parameter definition in the parent entity type it may be refined according to schema refinement rules.

- entry_schema: if defined in the parameter definition in the parent entity type it may be refined according to schema refinement rules.

- metadata: a new definition is unrestricted and will overwrite the one inherited from the parameter definition in the parent entity type definition

3.4.10 Parameter value assignment

This section defines the grammar for assigning values to named "outgoing" parameters in TOSCA templates.

3.4.10.1 Keynames

The TOSCA parameter value assignment has no keynames.

3.4.10.2 Grammar

Parameter value assignments have the following grammar:

3.4.10.2.1 Short notation:

The following single-line grammar may be used when a simple value assignment is needed:

```
<parameter_name>: <parameter_value> | { <parameter_value_expression> }
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- parameter_name: represents the symbolic name of the parameter to assign; note that in some cases, even parameters that do not have a corresponding definition in the entity type of the entity containing them may be assigned (see interface inputs and outputs).

- parameter_value, parameter_value_expression: represent the type-compatible value to assign to the parameter. Parameter values may be provided as the result from the evaluation of an expression or a function.

3.4.10.3 Additional Requirements

- Parameters that have a (fixed) value defined during their definition or during a subsequent refinement may not be assigned (as their value is already set).
- If a required parameter has no value defined or assigned, its default value is assigned
- A non-required parameter that is not assigned it stays undefined, thus the default keyname is irrelevant for a non-required parameter.

3.4.11 Parameter mapping assignment

A parameter to attribute mapping defines a named "incoming" parameter value (e.g. an output value that is expected to be returned by an operation implementation) and a mapping that specifies the node or relationship attribute into which the returned "incoming" parameter value must be stored.

3.4.11.1 Keynames

The TOSCA parameter mapping assignment has no keynames.

3.4.11.2 Grammar

Parameter mapping assignments have the following grammar:

```
<parameter_name>: <attribute_selection_format>
```

3.4.11.3 Attribute selection format

The attribute_selection_format is a list of the following format:

```
[ <SELF | SOURCE | TARGET >, <optional_capability_name>,
<attribute_name>, <nested_attribute_name_or_index_1>, ...,
<nested_attribute_name_or_index_or_key_n> ]
```

The various entities in this grammar are defined as follows:

Parameter	Required	Type	Description
SELF | SOURCE | TARGET	yes	string	For operation outputs in interfaces on node templates, the only allowed keyname is SELF: output values must always be stored into attributes that belong to the node template that has the interface for which the output values are returned. For operation outputs in interfaces on relationship templates, allowable keynames are SELF, SOURCE, or TARGET.
<optional_capability_name>	no	string	The optional name of the capability within the specified node template that contains the attribute into which the output value must be stored.
<attribute_name>	yes	string	The name of the attribute into which the output value must be stored.

Parameter	Required	Type	Description
<nested_attribute_name_or_index _or_key_*>	no	string| integer	Some TOSCA attributes are complex (i.e., composed as nested structures). These parameters are used to dereference into the names of these nested structures when needed. Some attributes represent list or map types. In these cases, an index or key may be provided to reference a specific entry in the list or map (identified by the previous parameter).

Note that it is possible for multiple operations to define outputs that map onto the same attribute value. For example, a *create* operation could include an output value that sets an attribute to an initial value, and the subsequence *configure* operation could then update that same attribute to a new value.

It is also possible that a node template assigns a value to an attribute that has an operation output mapped to it (including a value that is the result of calling an intrinsic function). Orchestrators could use the assigned value for the attribute as its initial value. After the operation runs that maps an output value onto that attribute, the orchestrator must then use the updated value, and the value specified in the node template will no longer be used.

3.5 Substitution

3.5.1 Substitution mapping

A substitution mapping allows a given topology template to be used as an implementation of abstract node templates of a specific node type. This allows the consumption of complex systems using a simplified vision.

3.5.1.1 Keynames

Keyname	Required	Type	Description
node_type	yes	string	The required name of the Node Type the Topology Template is providing an implementation for.
substitution_filter	no	node filter	The optional filter that further constrains the abstract node templates for which this topology template can provide an implementation.
properties	no	map of property mappings	The optional map of properties mapping allowing to map properties of the node_type to inputs of the topology template.
attributes	no	map of attribute mappings	The optional map of attribute mappings allowing to map outputs from the topology template to attributes of the node_type.
capabilities	no	map of capability mappings	The optional map of capabilities mapping.
requirements	no	map of requirement mappings	The optional map of requirements mapping.
interfaces	no	map of interfaces mappings	The optional map of interface mapping allows to map an interface and operations of the node type to implementations that could be either workflows or node template interfaces/operations.

3.5.1.2 Grammar

The grammar of the substitution_mapping section is as follows:

```
node_type: <node_type_name>
substitution_filter : <node_filter>
properties:
  <property_mappings>
capabilities:
  <capability_mappings>
requirements:
  <requirement_mappings>
attributes:
  <attribute_mappings>
interfaces:
  <interface_mappings>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- node_type_name: represents the required Node Type name that the Service Template's topology is offering an implementation for.
- node_filter: represents the optional node filter that reduces the set of abstract node templates for which this topology template is an implementation by only substituting for those node templates whose properties and capabilities satisfy the constraints specified in the node filter.
- **properties**: represents the <optional> map of properties mappings.
- **capability_mappings**: represents the <optional> map of capability mappings.
- **requirement_mappings**: represents the <optional> map of requirement mappings.
- **attributes**: represents the <optional> map of attributes mappings.
- **interfaces:** represents the <optional> map of interfaces mappings.

3.5.1.3 Examples

3.5.1.4 Additional requirements

- The substitution mapping MUST provide mapping for every property, capability and requirement defined in the specified <node_type>

3.5.1.5 Notes

- The node_type specified in the substitution mapping SHOULD be abstract (does not provide implementation for normative operations).

3.5.2 Property mapping

A property mapping allows to map the property of a substituted node type an input of the topology template.

3.5.2.1 Keynames

The following is the list of recognized keynames for a TOSCA property mapping:

Keyname	Required	Type	Description
mapping	no	list of strings	An array with 1 string element that references an input of the topology.

Keyname	Required	Type	Description
value	no	matching the type of this property	This **deprecated** keyname allows to explicitly assigne a value to this property. This field is mutually exclusive with the mapping keyname.

3.5.2.2 Grammar

The single-line grammar of a property_mapping is as follows:

```
<property_name>: <property_value> # This use is deprecated
<property_name>: [ <input_name> ]
```

The multi-line grammar is as follows :

```
<property_name>:
  mapping: [ < input_name > ]
<property_name>:
  value: <property_value> # This use is deprecated
```

3.5.2.3 Notes

- Single line grammar for a property value assignment is not allowed for properties of   type in order to avoid collision with the mapping single line grammar.

- The property_value mapping grammar has been deprecated. The original intent of the *property-to-constant-value* mapping was not to provide a *mapping*, but rather to present a *matching* mechanism to drive selection of the appropriate substituting template when more than one template was available as a substitution for the abstract node. In that case, a topology template was only a valid candidate for substitution if the property value in the abstract node template matched the constant value specified in the property_value mapping for that property. With the introduction of substitution filter syntax to drive matching, there is no longer a need for the property-to-constant-value mapping functionality.

- The previous version of the specification allowed direct mappings from properties of the abstract node template to properties of node templates in the substituting topology template. Support for these mappings has been deprecated since they would have resulted in unpredictable behavior, for the following reason. If the substituting template is a valid TOSCA template, then all the (required) properties of all its node templates must have valid property assignments already defined. If the substitution mappings of the substituting template include direct property-to-property mappings, the the substituting template ends up with two conflicting property assignments: one defined in the substituting template itself, and one defined by the substitution mappings. These conflicting assignments lead to unpredictable behavior.

3.5.2.4 Additional constraints

- When Input mapping it may be referenced by multiple nodes in the topologies with resulting attributes values that may differ later on in the various nodes. In any situation, the attribute reflecting the property of the substituted type will remain a constant value set to the one of the input at deployment time.

3.5.3 Attribute mapping

An attribute mapping allows to map the attribute of a substituted node type an output of the topology template.

3.5.3.1 Keynames

The following is the list of recognized keynames for a TOSCA attribute mapping:

Keyname	Required	Type	Description
mapping	no	list of strings	An array with 1 string element that references an output of the topology..

3.5.3.2 Grammar

The single-line grammar of an attribute_mapping is as follows:

```
<attribute_name>: [ <output_name> ]
```

3.5.4 Capability mapping

A capability mapping allows to map the capability of one of the node of the topology template to the capability of the node type the service template offers an implementation for.

3.5.4.1 Keynames

The following is the list of recognized keynames for a TOSCA capability mapping:

Keyname	Required	Type	Description
mapping	no	list of strings (with 2 members)	A list of strings with 2 members, the first one being the name of a node template, the second the name of a capability of the specified node template.
properties	no	map of property assignments	This field is mutually exclusive with the mapping keyname and allows to provide a capability assignment for the template and specify it's related properties.
attributes	no	map of attributes assignments	This field is mutually exclusive with the mapping keyname and allows to provide a capability assignment for the template and specify it's related attributes.

3.5.4.2 Grammar

The single-line grammar of a capability_mapping is as follows:

```
<capability_name>: [ <node_template_name>, <node_template_capability_name>
]
```

The multi-line grammar is as follows :

```
<capability_name>:
  mapping: [ <node_template_name>, <node_template_capability_name> ]
  properties:
    <property_name>: <property_value>
  attributes:
    <attribute_name>: <attribute_value>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- capability_name: represents the name of the capability as it appears in the Node Type definition for the Node Type (name) that is declared as the value for on the substitution_mappings' "node_type" key.

- node_template_name: represents a valid name of a Node Template definition (within the same topology_template declaration as the substitution_mapping is declared).
- node_template_capability_name: represents a valid name of a capability definition within the <node_template_name> declared in this mapping.
- property_name: represents the name of a property of the capability.
- property_value: represents the value to assign to a property of the capability.
- attribute_name: represents the name a an attribute of the capability.
- attribute_value: represents the value to assign to an attribute of the capability.

3.5.4.3 Additional requirements

- Definition of capability assignment in a capability mapping (through properties and attribute keynames) SHOULD be prohibited for connectivity capabilities as tosca.capabilities.Endpoint.

3.5.5 Requirement mapping

A requirement mapping allows to map the requirement of one of the node of the topology template to the requirement of the node type the service template offers an implementation for.

3.5.5.1 Keynames

The following is the list of recognized keynames for a TOSCA requirement mapping:

Keyname	Required	Type	Description
mapping	no	list of strings (2 members)	A list of strings with 2 elements, the first one being the name of a node template, the second the name of a requirement of the specified node template.
properties	no	List of property assignment	This field is mutually exclusive with the mapping keyname and allow to provide a requirement for the template and specify it's related properties.
attributes	no	List of attributes assignment	This field is mutually exclusive with the mapping keyname and allow to provide a requirement for the template and specify it's related attributes.

3.5.5.2 Grammar

The single-line grammar of a requirement_mapping is as follows:

```
<requirement_name>: [ <node_template_name>,
<node_template_requirement_name> ]
```

The multi-line grammar is as follows :

```
<requirement_name>:
  mapping: [ <node_template_name>, <node_template_requirement_name> ]
  properties:
    <property_name>: <property_value>
  attributes:
    <attribute_name>: <attribute_value>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- requirement_name: represents the name of the requirement as it appears in the Node Type definition for the Node Type (name) that is declared as the value for on the substitution_mappings' "node_type" key.
- node_template_name: represents a valid name of a Node Template definition (within the same topology_template declaration as the substitution_mapping is declared).
- node_template_requirement_name: represents a valid name of a requirement definition within the <node_template_name> declared in this mapping.
- property_name: represents the name of a property of the requirement.
- property_value: represents the value to assign to a property of the requirement.
- attribute_name: represents the name of an attribute of the requirement.
- attribute_value: represents the value to assign to an attribute of the requirement.

3.5.5.3 Additional requirements

- Definition of capability assignment in a capability mapping (through properties and attribute keynames) SHOULD be prohibited for connectivity capabilities as tosca.capabilities.Endpoint.

3.5.6 Interface mapping

An interface mapping allows to map a workflow of the topology template to an operation of the node type the service template offers an implementation for.

3.5.6.1 Grammar

The grammar of an interface_mapping is as follows:

```
<interface_name>:
  <operation_name>: <workflow_name>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- **interface_name:** represents the name of the interface as it appears in the Node Type definition for the Node Type (name) that is declared as the value for on the substitution_mappings' "node_type" key. Or the name of a new management interface to add to the generated type.
- **operation_name:** represents the name of the operation as it appears in the interface type definition.
- **workflow_name:** represents the name of a workflow of the template to map to the specified operation.

3.5.6.2 Notes

- Declarative workflow generation will be applied by the TOSCA orchestrator after the topology template have been substituted. Unless one of the normative operation of the standard interface is mapped through an interface mapping. In that case the declarative workflow generation will consider the substitution node as any other node calling the create, configure and start mapped workflows as if they where single operations.
- Operation implementation being TOSCA workflows the TOSCA orchestrator replace the usual operation_call activity by an inline activity using the specified workflow.

3.6 Groups and Policies

3.6.1 Group Type

A Group Type defines logical grouping types for nodes, typically for different management purposes. Conceptually, group definitions allow the creation of logical "membership" relationships to nodes in a service template that are not a part of the application's explicit requirement dependencies in the topology template (i.e. those required to actually get the application deployed and running). Instead, such logical membership allows for the introduction of things such as group management and uniform application of policies (i.e., requirements that are also not bound to the application itself) to the group's members.

3.6.1.1 Keynames

The Group Type is a TOSCA type entity and has the common keynames listed in Section 3.2.5.2 Common keynames in type definitions. In addition, the Group Type has the following recognized keynames:

Keyname	Required	Type	Description
attributes	no	map of attribute definitions	An optional map of attribute definitions for the Group Type.
properties	no	map of property definitions	An optional map of property definitions for the Group Type.
members	no	list of string	An optional list of one or more names of Node Types that are valid (allowed) as members of the Group Type. Note: This can be viewed by TOSCA Orchestrators as an implied relationship from the listed members nodes to the group, but one that does not have operational lifecycle considerations. For example, if we were to name this as an explicit Relationship Type we might call this "MemberOf" (group).

3.6.1.2 Grammar

Group Types have one the following grammars:

```
<group_type_name>:
  derived_from: <parent_group_type_name>
  version: <version_number>
  metadata:
    <map of string>
  description: <group_description>
  attributes :
    <attribute_definitions>
  properties:
    <property_definitions>
  members: [ <list_of_valid_member_types> ]
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- group_type_name: represents the required symbolic name of the Group Type being declared as a string.
- parent_group_type_name: represents the name (string) of the Group Type this Group Type definition derives from (i.e., its "parent" type).
- version_number: represents the optional TOSCA version number for the Group Type.
- group_description: represents the optional description string for the corresponding group_type_name.
- attribute_definitions: represents the optional map of attribute_definitions for the Group Type.
- property_definitions: represents the optional map of property definitions for the Group Type.

- list_of_valid_member_types: represents the optional list of TOSCA types (e.g.,., Node, Capability or even other Group Types) that are valid member types for being added to (i.e., members of) the Group Type.

3.6.1.3 Notes

Note that earlier versions of this specification support interface definitions, capability definitions, and requirement definitions in group types. These definitions have been deprecated in this version based on the realization that groups in TOSCA only exist for purposes of uniform application of policies to collections of nodes. Consequently, groups do not have a lifecycle of their own that is independent of the lifeycle of their members.

3.6.1.4 Additional Requirements

- Group definitions **SHOULD NOT** be used to define or redefine relationships (dependencies) between nodes that can be expressed using normative TOSCA Relationships (e.g., HostedOn, ConnectsTo, etc.) within a TOSCA topology template.
- The list of values associated with the "members" keyname **MUST** only contain types that or homogenous (i.e., derive from the same type hierarchy).

3.6.1.5 Example

The following represents a Group Type definition:

```
group_types:
  mycompany.mytypes.groups.placement:
    description: My company's group type for placing nodes of type Compute
    members: [ tosca.nodes.Compute ]
```

3.6.2 Group definition

A group definition defines a logical grouping of node templates, typically for management purposes, but is separate from the application's topology template.

3.6.2.1 Keynames

The following is the list of recognized keynames for a TOSCA group definition:

Keyname	Required	Type	Description
type	yes	string	The required name of the group type the group definition is based upon.
description	no	description	The optional description for the group definition.
metadata	no	map of string	Defines a section used to declare additional metadata information.
properties	no	map of property assignments	An optional map of property value assignments for the group definition.
members	no	list of string	The optional list of one or more node template names that are members of this group definition.

3.6.2.2 Grammar

Group definitions have one the following grammars:

```
<group_name>:
  type: <group_type_name>
  description: <group_description>
  metadata:
    <map of string>
  attributes :
    <attribute_assignments>
  properties:
    <property_assignments>
  members: [ <list_of_node_templates> ]
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- group_name: represents the required symbolic name of the group as a string.
- group_type_name: represents the name of the Group Type the definition is based upon.
- group_description: contains an optional description of the group.
- attribute_assigments: represents the optional map of attribute_assignments for the group definition that provide values for attributes defined in its declared Group Type.
- property_assignments: represents the optional map of property assignments for the group definition that provide values for properties defined in its declared Group Type.
- list_of_node_templates: contains the required list of one or more node template names (within the same topology template) that are members of this logical group.

3.6.2.3 Notes

Note that earlier versions of this specification support interface definitions in group definitions. These definitions have been deprecated in this version based on the realization that groups in TOSCA only exist for purposes of uniform application of policies to collections of nodes. Consequently, groups do not have a lifecycle of their own that is independent of the lifeycle of their members.

3.6.2.4 Additional Requirements

- Group definitions **SHOULD NOT** be used to define or redefine relationships (dependencies) for an application that can be expressed using normative TOSCA Relationships within a TOSCA topology template.

3.6.2.5 Example

The following represents a group definition:

```
groups:
  my_app_placement_group:
    type: tosca.groups.Root
    description: My application's logical component grouping for placement
    members: [ my_web_server, my_sql_database ]
```

3.6.3 Policy Type

A Policy Type defines a type of requirement that affects or governs an application or service's topology at some stage of its lifecycle, but is not explicitly part of the topology itself (i.e., it does not prevent the application or service from being deployed or run if it did not exist).

3.6.3.1 Keynames

The Policy Type is a TOSCA type entity and has the common keynames listed in Section 3.2.5.2 Common keynames in type definitions. In addition, the Policy Type has the following recognized keynames:

Keyname	Required	Type	Description
properties	no	map of property definitions	An optional map of property definitions for the Policy Type.
targets	no	list of string	An optional list of valid Node Types or Group Types the Policy Type can be applied to. Note: This can be viewed by TOSCA Orchestrators as an implied relationship to the target nodes, but one that does not have operational lifecycle considerations. For example, if we were to name this as an explicit Relationship Type we might call this "AppliesTo" (node or group).
triggers	no	map of trigger definitions	An optional map of policy triggers for the Policy Type.

3.6.3.2 Grammar

Policy Types have the following grammar:

```
<policy_type_name>:
  derived_from: <parent_policy_type_name>
  version: <version_number>
  metadata:
    <map of string>
  description: <policy_description>
  properties:
    <property_definitions>
  targets: [ <list_of_valid_target_types> ]
  triggers:
    <trigger_definitions>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- policy_type_name: represents the required symbolic name of the Policy Type being declared as a string.
- parent_policy_type_name: represents the name (string) of the Policy Type this Policy Type definition derives from (i.e., its "parent" type).
- version_number: represents the optional TOSCA version number for the Policy Type.
- policy_description: represents the optional description string for the corresponding policy_type_name.
- property_definitions: represents the optional map of property definitions for the Policy Type.
- list_of_valid_target_types: represents the optional list of TOSCA types (i.e., Group or Node Types) that are valid targets for this Policy Type.
- trigger_definitions: represents the optional map of trigger definitions for the policy.

3.6.3.3 Example

The following represents a Policy Type definition:

```
policy_types:
  mycompany.mytypes.policies.placement.Container.Linux:
    description: My company's placement policy for linux
```

```
        derived_from: tosca.policies.Root
```

3.6.4 Policy definition

A policy definition defines a policy that can be associated with a TOSCA topology or top-level entity definition (e.g., group definition, node template, etc.).

3.6.4.1 Keynames

The following is the list of recognized keynames for a TOSCA policy definition:

Keyname	Required	Type	Description
type	yes	string	The required name of the policy type the policy definition is based upon.
description	no	description	The optional description for the policy definition.
metadata	no	map of string	Defines a section used to declare additional metadata information.
properties	no	map of property assignments	An optional map of property value assignments for the policy definition.
targets	no	list of string	An optional list of valid Node Templates or Groups the Policy can be applied to.
triggers	no	map of trigger definitions	An optional map of trigger definitions to invoke when the policy is applied by an orchestrator against the associated TOSCA entity.

3.6.4.2 Grammar

Policy definitions have one the following grammars:

```
<policy_name>:
  type: <policy_type_name>
  description: <policy_description>
  metadata:
    <map of string>
  properties:
    <property_assignments>
  targets: [<list_of_policy_targets>]
  triggers:
    <trigger_definitions>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

*   policy_name: represents the required symbolic name of the policy as a string.
*   policy_type_name: represents the name of the policy the definition is based upon.
*   policy_description: contains an optional description of the policy.
*   property_assignments: represents the optional map of property assignments for the policy definition that provide values for properties defined in its declared Policy Type.
*   list_of_policy_targets: represents the optional list of names of node templates or groups that the policy is to applied to.
*   trigger_definitions: represents the optional map of trigger definitions for the policy.

3.6.4.3 Example

The following represents a policy definition:

```
policies:
  - my_compute_placement_policy:
      type: tosca.policies.placement
      description: Apply my placement policy to my application's servers
      targets: [ my_server_1, my_server_2 ]
      # remainder of policy definition left off for brevity
```

3.6.5 Trigger definition

A trigger definition defines the event, condition and action that is used to "trigger" a policy it is associated with.

3.6.5.1 Keynames

The following is the list of recognized keynames for a TOSCA trigger definition:

Keyname	Required	Type	Description
description	no	description	The optional description string for the trigger.
event	yes	string	The required name of the event that activates the trigger's action. A deprecated form of this keyname is "event_type".
schedule	no	TimeInterval	The optional time interval during which the trigger is valid (i.e., during which the declared actions will be processed).
target_filter	no	event filter	The optional filter used to locate the attribute to monitor for the trigger's defined condition. This filter helps locate the TOSCA entity (i.e., node or relationship) or further a specific capability of that entity that contains the attribute to monitor.
condition	no	condition clause definition	The optional condition which contains a condition clause definition specifying one or multiple attribute constraint that can be monitored. Note: this is optional since sometimes the event occurrence itself is enough to trigger the action.
action	yes	list of activity definition	The list of sequential activities to be performed when the event is triggered, and the condition is met (i.e. evaluates to true).

3.6.5.2 Additional keynames for the extended condition notation

Keyname	Required	Type	Description
constraint	no	condition clause definition	The optional condition which contains a condition clause definition specifying one or multiple attribute constraint that can be monitored. Note: this is optional since sometimes the event occurrence itself is enough to trigger the action.
period	no	scalar-unit.time	The optional period to use to evaluate for the condition.
evaluations	no	integer	The optional number of evaluations that must be performed over the period to assert the condition exists.
method	no	string	The optional statistical method name to use to perform the evaluation of the condition.

3.6.5.3 Grammar

Trigger definitions have the following grammars:

3.6.5.3.1 Short notation

```
<trigger_name>:
  description: <trigger_description>
  event: <event _name>
  schedule: <time_interval_for_trigger>
  target_filter:
    <event_filter_definition>
  condition:
    <condition_clause_definition>
  action:
    - <list_of_activity_definition>
```

3.6.5.3.2 Extended notation:

```
<trigger_name>:
  description: <trigger_description>
  event: <event _name>
  schedule: <time_interval_for_trigger>
  target_filter:
    <event_filter_definition>
  condition:
    constraint: <condition_clause_definition>
    period: <scalar-unit.time> # e.g., 60 sec
    evaluations: <integer> # e.g., 1
    method: <string> # e.g., average
  action:
    - <list_of_activity_definition>
```

In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

- trigger_name: represents the required symbolic name of the trigger as a string.

- trigger_description: represents the optional description string for the corresponding trigger_name.

- event_name: represents the required name of an event associated with an interface notification on the identified resource (node).

- time_interval_for_trigger: represents the optional time interval that the trigger is valid for.

- event_filter_definition: represents the optional filter to use to locate the resource (node) or capability attribute to monitor.

- condition_clause_definition: represents one or multiple attribute constraints that can be monitored, and that used to test for a specific condition on the monitored resource.

- list_of_activity_definition: represents the list of activities that are performed if the event and (optionally) condition are met. The activity definitions are the same as the ones used in a workflow step. One could regard these activities as an anonymous workflow that is invoked by this trigger and is applied to the target(s) of this trigger's policy.

3.6.6 Event Filter definition

An event filter definition defines criteria for selection of an attribute, for the purpose of monitoring it, within a TOSCA entity, or one its capabilities.

3.6.6.1 Keynames

The following is the list of recognized keynames for a TOSCA event filter definition:

Keyname	Required	Type	Description
node	yes	string	The required name of the node type or template that contains either the attribute to be monitored or contains the requirement that references the node that contains the attribute to be monitored.
requirement	no	string	The optional name of the requirement within the filter's node that can be used to locate a referenced node that contains an attribute to monitor.
capability	no	string	The optional name of a capability within the filter's node or within the node referenced by its requirement that contains the attribute to monitor.

3.6.6.2 Grammar

Event filter definitions have following grammar:

```
node: <node_type_name> | <node_template_name>
requirement: <requirement_name>
capability: <capability_name>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- node_type_name: represents the required name of the node type that will be used to select (filter) the node that contains the attribute to monitor or contains the requirement that references another node that contains the attribute to monitor.

- node_template_name: represents the required name of the node template that will be used to select (filter) the node that contains the attribute to monitor or contains the requirement that references another node that contains the attribute to monitor.

- requirement_name: represents the optional name of the requirement that will be used to select (filter) a referenced node that contains the attribute to monitor.

- capability_name: represents the optional name of a capability that will be used to select (filter) the attribute to monitor. If a requirement_name is specified, then the capability_name refers to a capability of the node that is targeted by the requirement.

3.6.7 Condition clause definition

A workflow condition clause definition is used to specify a condition that can be used within a workflow precondition or workflow filter.

3.6.7.1 Keynames

The following is the list of recognized keynames for a TOSCA workflow condition definition:

Keyname	Required	Type	Description
and	no	list of condition clause definition	An and clause allows to define sub-filter clause definitions that must all be evaluated truly so the and clause is considered as true.
or	no	list of condition clause definition	An or clause allows to define sub-filter clause definitions where one of them must all be evaluated truly so the or clause is considered as true.

Keyname	Required	Type	Description
not	no	list of condition clause definition	A **not** clause allows to define sub-filter clause definitions where one or more of them must be evaluated as false.
assert (deprecated)	no	list of assertion definition	An **assert** clause defines a list of assertions that are evaluated on entity attributes. **Assert** acts as an **and** clause, i.e. every defined constraint clause must be true for the assertion to be true. Because **assert** and **and** (applied to several direct assertion clauses) are logically identical, the **assert** keyname has been deprecated.

Note: It is allowed to add direct assertion definitions directly to the condition clause definition without using any of the supported keynames. In that case, an *and* clause is performed for all direct assertion definition.

3.6.7.2 Grammar

Condition clause definitions have the following grammars:

3.6.7.2.1 And clause

```
and: <list_of_condition_clause_definition>
```

In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

- **list_of_condition_clause_definition:** represents the list of condition clauses. All condition clauses MUST be asserted to true so that the and clause is asserted to true.

3.6.7.2.2 Or clause

```
or: <list_of_condition_clause_definition>
```

In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

- **list_of_condition_clause_definition:** represents the list of condition clauses. One of the condition clause have to be asserted to true so that the or clause is asserted to true.

3.6.7.2.3 Not clause

```
not: <list_of_condition_clause_definition>
```

In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

- **list_of_condition_clause_definition:** represents the list of condition clauses. One of the condition clause have to be asserted to false so that the not clause is asserted to true.

3.6.7.3 Direct assertion definition

```
<attribute_name>: <list_of_constraint_clauses>
```

In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

- **attribute_name:** represents the name of an attribute defined on the assertion context entity (node instance, relationship instance, group instance) and from which value will be evaluated against the defined constraint clauses.
- **list_of_constraint_clauses:** represents the list of constraint clauses that will be used to validate the attribute assertion.

3.6.7.4 Additional Requirement

- Keynames are mutually exclusive, i.e. a filter definition can define only one of the *and*, *or*, or *not* keynames.

3.6.7.5 Notes

- The TOSCA processor SHOULD perform assertion in the order of the list for every defined condition clause or direct assertion definition.

3.6.7.6 Example

Following represents a workflow condition clause with a single direct assertion definition:

```
condition:
  - my_attribute: [{equal: my_value}]
```

Following represents a workflow condition clause with a direct assertion definition with multiple constraints:

```
condition:
  - my_attribute:
    - min_length: 8
    - max_length: 11
```

Following represents a workflow condition clause with single equals constraints on two different attributes.

```
condition:
  - my_attribute: [{equal: my_value}]
  - my_other_attribute: [{equal: my_other_value}]
```

Note that these two direct assertion constraints are logically *and*-ed. This means that the following is logically identical to the previous example:

```
condition:
  - and:
    - my_attribute: [{equal: my_value}]
    - my_other_attribute: [{equal: my_other_value}]
```

Following represents a workflow condition clause with an or constraint on two different assertions:

```
condition:
  - or:
    - my_attribute: [{equal: my_value}]
    - my_other_attribute: [{equal: my_other_value}]
```

The following shows an example of the *not* operator. The condition yields TRUE when the attribute my_attribute1 takes any value other than value1:

```
condition:
  - not:
    - my_attribute1: [{equal: value1}]}
```

The following condition yields TRUE when none of the attributes my_attribute1 and my_attribute2 is equal to value1.

```
condition:
  - not:
    - and:
      - my_attribute1: [{equal: value1}]
      - my_attribute2: [{equal: value1}]
```

The following condition is a functional equivalent of the previous example:

```
condition:
  - or:
    - not:
      - my_attribute1: [{equal: value1}]
    - not:
      - my_attribute2: [{equal: value1}]
```

Following represents multiple levels of condition clauses with direct assertion definitions to build the following logic: use http on port 80 or https on port 431:

```
condition:
  - or:
    -     and:
      - protocol: { equal: http }
      - port: { equal: 80 }
    -     and:
      - protocol: { equal: https }
      - port: { equal: 431 }
```

3.6.8 Assertion definition

A workflow assertion is used to specify a single condition on a workflow filter definition. The assertion allows to assert the value of an attribute based on TOSCA constraints.

3.6.8.1 Keynames

The TOSCA workflow assertion definition has no keynames.

3.6.8.2 Grammar

Workflow assertion definitions have the following grammar:

```
<attribute_name>: <list_of_constraint_clauses>
```

In the above grammars, the pseudo values that appear in angle brackets have the following meaning:

- **attribute_name:** represents the name of an attribute defined on the assertion context entity (node instance, relationship instance, group instance) and from which value will be evaluated against the defined constraint clauses.

- **list_of_constraint_clauses:** represents the list of constraint clauses that will be used to validate the attribute assertion.

3.6.8.3 Example

Following represents a workflow assertion with a single equals constraint:

```
my_attribute: [{equal : my_value}]
```

Following represents a workflow assertion with multiple constraints:

```
my_attribute:
  - min_length: 8
  - max_length : 10
```

3.6.9 Activity definitions

An activity defines an operation to be performed in a TOSCA workflow step or in an action body of a policy trigger. Activity definitions can be of the following types:

- Delegate workflow activity definition:
    - Defines the name of the delegate workflow and optional input assignments. This activity requires the target to be provided by the orchestrator (no-op node or relationship).
- Set state activity definition:
    - Sets the state of a node.
- Call operation activity definition:
    - Calls an operation defined on a TOSCA interface of a node, relationship or group. The operation name uses the <interface_name>.<operation_name> notation. Optionally, assignments for the operation inputs can also be provided. If provided, they will override for this operation call the operation inputs assignment in the node template.
- Inline workflow activity definition:
    - Inlines another workflow defined in the topology (allowing reusability). The definition includes the name of a workflow to be inlined and optional workflow input assignments.

3.6.9.1 Delegate workflow activity definition

3.6.9.1.1 Keynames

The following is a list of recognized keynames for a delegate activity definition.

Keyname	Required	Type	Description
delegate	yes	string or empty (see grammar below)	Defines the name of the delegate workflow and optional input assignments. This activity requires the target to be provided by the orchestrator (no-op node or relationship).
workflow	no	string	The name of the delegate workflow. Required in the extended notation.
inputs	no	map of parameter assignments	The optional map of input parameter assignments for the delegate workflow.

3.6.9.1.2 Grammar

A delegate activity definition has the following grammar. The short notation can be used if no input assignments are provided.

3.6.9.1.2.1 Short notation

```
- delegate: <delegate_workflow_name>
```

3.6.9.1.2.2 Extended notation

```
- delegate:
    workflow: <delegate_workflow_name>
    inputs:
```

```
        <parameter_assignments>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- delegate_workflow_name: represents the name of the workflow of the node provided by the TOSCA orchestrator
- **parameter_assignments**: represents the optional map of parameter assignments for passing parameters as inputs to this workflow delegation.

3.6.9.2 Set state activity definition

Sets the state of the target node.

3.6.9.2.1 Keynames

The following is a list of recognized keynames for a set state activity definition.

Keyname	Required	Type	Description
set_state	yes	string	Value of the node state.

3.6.9.2.2 Grammar

A set state activity definition has the following grammar.

```
- set_state: <new_node_state>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- new_node_state: represents the state that will be affected to the node once the activity is performed.

3.6.9.3 Call operation activity definition

This activity is used to call an operation on the target node. Operation input assignments can be optionally provided.

3.6.9.3.1 Keynames

The following is a list of recognized keynames for a call operation activity definition.

Keyname	Required	Type	Description
call_operation	yes	string or empty (see grammar below)	Defines the opration call. The operation name uses the <interface_name>.<operation_name> notation. Optionally, assignments for the operation inputs can also be provided. If provided, they will override for this operation call the operation inputs assignment in the node template.
operation	no	string	The name of the operation to call, using the <interface_name>.<operation_name> notation. Required in the extended notation.
inputs	no	map of parameter assignments	The optional map of input parameter assignments for the called operation. Any provided input assignments will override the operation input assignment in the target node template for this operation call.

3.6.9.3.2 Grammar

A call operation activity definition has the following grammar. The short notation can be used if no input assignments are provided.

3.6.9.3.2.1 Short notation

```
- call_operation: <operation_name>
```

3.6.9.3.2.2 Extended notation

```
- call_operation:
    operation: <operation_name>
    inputs:
      <parameter_assignments>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- operation_name: represents the name of the operation that will be called during the workflow execution. The notation used is <interface_sub_name>.<operation_sub_name>, where interface_sub_name is the interface name and the operation_sub_name is the name of the operation whitin this interface.

- **parameter_assignments**: represents the optional map of parameter assignments for passing parameters as inputs to this workflow delegation.

3.6.9.4 Inline workflow activity definition

This activity is used to inline a workflow in the activities sequence. The definition includes the name of the inlined workflow and optional input assignments.

3.6.9.4.1 Keynames

The following is a list of recognized keynames for an inline workflow activity definition.

Keyname	Required	Type	Description
inline	yes	string or empty (see grammar below)	The definition includes the name of a workflow to be inlined and optional workflow input assignments.
workflow	no	string	The name of the inlined workflow. Required in the extended notation.
inputs	no	map of parameter assignments	The optional map of input parameter assignments for the inlined workflow.

3.6.9.4.2 Grammar

An inline workflow activity definition has the following grammar. The short notation can be used if no input assignments are provided.

3.6.9.4.2.1 Short notation

```
- inline: <inlined_workflow_name>
```

3.6.9.4.2.2 Extended notation

```
- inline:
    workflow: <inlined_workflow_name>
    inputs:
      <parameter_assignments>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- inlined_workflow_name: represents the name of the workflow to inline.

- **parameter_assignments**: represents the optional map of parameter assignments for passing parameters as inputs to this workflow delegation.

3.6.9.5 Example

The following represents a list of activity definitions (using the short notation):

```
- delegate: deploy
- set_state: started
- call_operation: tosca.interfaces.node.lifecycle.Standard.start
- inline: my_workflow
```

3.7 Workflows

3.7.1 Imperative Workflow definition

A workflow definition defines an imperative workflow that is associated with a TOSCA topology. A workflow definition can either include the steps that make up the workflow, or it can refer to an artifact that expresses the workflow using an external workflow language.

3.7.1.1 Keynames

The following is the list of recognized keynames for a TOSCA workflow definition:

Keyname	Required	Type	Description
description	no	description	The optional description for the workflow definition.
metadata	no	map of string	Defines a section used to declare additional metadata information.
inputs	no	map of parameter definitions	The optional map of input parameter definitions.
preconditions	no	list of precondition definitions	List of preconditions to be validated before the workflow can be processed.
steps	no	map of step definitions	An optional map of valid imperative workflow step definitions.
implementation	no	operation implementation definition	The optional definition of an external workflow definition. This keyname is mutually exclusive with the **steps** keyname above.
outputs	no	map of attribute mappings	The optional map of attribute mappings that specify workflow output values and their mappings onto attributes of a node or relationship defined in the topology.

3.7.1.2 Grammar

Imperative workflow definitions have the following grammar:

```
<workflow_name>:
  description: <workflow_description>
  metadata:
    <map of string>
  inputs:
    <parameter_definitions>
  preconditions:
   - <workflow_precondition_definition>
  steps:
    <workflow_steps>
  implementation:
    <operation_implementation_definitions>
  outputs:
    <attribute_mappings>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- workflow_name:
- workflow_description:
- parameter_definitions:
- workflow_precondition_definition:
- workflow_steps:
- operation_implementation_definition: represents a full inline definition of an implementation artifact
- attribute_mappings: represents the optional map of of attribute_mappings that consists of named output values returned by operation implementations (i.e. artifacts) and associated mappings that specify the attribute into which this output value must be stored.

3.7.2 Workflow precondition definition

A workflow condition can be used as a filter or precondition to check if a workflow can be processed or not based on the state of the instances of a TOSCA topology deployment. When not met, the workflow will not be triggered.

3.7.2.1 Keynames

The following is the list of recognized keynames for a TOSCA workflow condition definition:

Keyname	Required	Type	Description
target	yes	string	The target of the precondition (this can be a node template name, a group name)
target_relationship	no	string	The optional name of a requirement of the target in case the precondition has to be processed on a relationship rather than a node or group. Note that this is applicable only if the target is a node.
condition	no	list of condition clause definitions	A list of workflow condition clause definitions. Assertion between elements of the condition are evaluated as an AND condition.

3.7.2.2 Grammar

Workflow precondition definitions have the following grammars:

```
- target: <target_name>
  target_relationship: <target_requirement_name>
  condition:
    <list_of_condition_clause_definition>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- target_name: represents the name of a node template or group in the topology.
- target_requirement_name: represents the name of a requirement of the node template (in case target_name refers to a node template.
- list_of_condition_clause_definition: represents the list of condition clauses to be evaluated. The value of the resulting condition is evaluated as an AND clause between the different elements.

3.7.3 Workflow step definition

A workflow step allows to define one or multiple sequenced activities in a workflow and how they are connected to other steps in the workflow. They are the building blocks of a declarative workflow.

3.7.3.1 Keynames

The following is the list of recognized keynames for a TOSCA workflow step definition:

Keyname	Required	Type	Description
target	yes	string	The target of the step (this can be a node template name, a group name)
target_relationship	no	string	The optional name of a requirement of the target in case the step refers to a relationship rather than a node or group. Note that this is applicable only if the target is a node.
operation_host	no	string	The node on which operations should be executed (for TOSCA call_operation activities). This element is required only for relationships and groups target. If target is a relationships operation_host is required and valid_values are SOURCE or TARGET – referring to the relationship source or target node. If target is a group operation_host is optional. If not specified the operation will be triggered on every node of the group. If specified the valid_value is a node_type or the name of a node template.
filter	no	list of constraint clauses	Filter is a map of attribute name, list of constraint clause that allows to provide a filtering logic.
activities	yes	list of activity definition	The list of sequential activities to be performed in this step.
on_success	no	list of string	The optional list of step names to be performed after this one has been completed with success (all activities has been correctly processed).
on_failure	no	list of string	The optional list of step names to be called after this one in case one of the step activity failed.

3.7.3.2 Grammar

Workflow step definitions have the following grammars:

```
steps:
  <step_name>
    target: <target_name>
    target_relationship: <target_requirement_name>
    operation_host: <operation_host_name>
    filter:
      - <list_of_condition_clause_definition>
    activities:
      - <list_of_activity_definition>
    on_success:
      - <target_step_name>
    on_failure:
      - <target_step_name>
```

In the above grammar, the pseudo values that appear in angle brackets have the following meaning:

- target_name: represents the name of a node template or group in the topology.
- target_requirement_name: represents the name of a requirement of the node template (in case target_name refers to a node template.
- **operation_host:** the node on which the operation should be executed
- **list_of_condition_clause_definition:** represents a list of condition clause definition.
- **list_of_activity_definition**: represents a list of activity definition
- **target_step_name**: represents the name of another step of the workflow.

3.8 Normative values

3.8.1 Node States

As components (i.e. nodes) of TOSCA applications are deployed, instantiated and orchestrated over their lifecycle using normative lifecycle operations (see section 5.8 for normative lifecycle definitions) it is important define normative values for communicating the states of these components normatively between orchestration and workflow engines and any managers of these applications.

The following table provides the list of recognized node states for TOSCA that will be set by the orchestrator to describe a node instance's state:

Node State		
Value	**Transitional**	**Description**
initial	no	Node is not yet created. Node only exists as a template definition.
creating	yes	Node is transitioning from initial state to created state.
created	no	Node software has been installed.
configuring	yes	Node is transitioning from created state to configured state.
configured	no	Node has been configured prior to being started.
starting	yes	Node is transitioning from configured state to started state.

Node State		
Value	**Transitional**	**Description**
started	no	Node is started.
stopping	yes	Node is transitioning from its current state to a configured state.
deleting	yes	Node is transitioning from its current state to one where it is deleted and its state is no longer tracked by the instance model.
error	no	Node is in an error state.

3.8.2 Relationship States

Similar to the Node States described in the previous section, Relationships have state relative to their (normative) lifecycle operations.

The following table provides the list of recognized relationship states for TOSCA that will be set by the orchestrator to describe a node instance's state:

Node State		
Value	**Transitional**	**Description**
initial	no	Relationship is not yet created. Relationship only exists as a template definition.

3.8.2.1 Notes

- Additional states may be defined in future versions of the TOSCA specification.

3.8.3 Directives

The following directive values are defined for this version of TOSCA :

Directive	**Description**
substitute	Marks a node template as abstract and instructs the TOSCA Orchestrator to substitute this node template with an appropriate substituting template.
substitutable	This **deprecated** directive is synonymous to the substitute directive.
select	Marks a node template as abstract and instructs the TOSCA Orchestrator to select a node of this type from its inventory (based on constraints specified in the optional node_filter in the node template)
selectable	This **deprecated** directive is synonymous to the select directive.

3.8.4 Network Name aliases

The following are recognized values that may be used as aliases to reference types of networks within an application model without knowing their actual name (or identifier) which may be assigned by the underlying Cloud platform at runtime.

Alias value	Description
PRIVATE	An alias used to reference the first private network within a property or attribute of a Node or Capability which will be assigned to them by the underlying platform at runtime. A private network contains IP addresses and ports typically used to listen for incoming traffic to an application or service from the Intranet and not accessible to the public internet.
PUBLIC	An alias used to reference the first public network within a property or attribute of a Node or Capability which will be assigned to them by the underlying platform at runtime. A public network contains IP addresses and ports typically used to listen for incoming traffic to an application or service from the Internet.

3.8.4.1 Usage

These aliases will be used in the tosca.capabilities.Endpoint Capability type (and types derived from it) within the network_name field for template authors to use to indicate the type of network the Endpoint is supposed to be assigned an IP address from.

4 TOSCA functions

Except for the examples, this section is **normative** and includes functions that are supported for use within a TOSCA Service Template.

4.1 Reserved Function Keywords

The following keywords MAY be used in some TOSCA function in place of a TOSCA Node or Relationship Template name. A TOSCA orchestrator will interpret them at the time the function will be evaluated (e.g. at runtime) as described in the table below. Note that some keywords are only valid in the context of a certain TOSCA entity as also denoted in the table.

Keyword	Valid Contexts	Description
SELF	Node Template or Relationship Template	A TOSCA orchestrator will interpret this keyword as the Node or Relationship Template instance that contains the function at the time the function is evaluated.
SOURCE	Relationship Template only.	A TOSCA orchestrator will interpret this keyword as the Node Template instance that is at the source end of the relationship that contains the referencing function.
TARGET	Relationship Template only.	A TOSCA orchestrator will interpret this keyword as the Node Template instance that is at the target end of the relationship that contains the referencing function.
HOST	Node Template only	A TOSCA orchestrator will interpret this keyword to refer to the all nodes that "host" the node using this reference (i.e., as identified by its HostedOn relationship). Specifically, TOSCA orchestrators that encounter this keyword when evaluating the get_attribute or get_property functions SHALL search each node along the "HostedOn" relationship chain starting at the immediate node that hosts the node where the function was evaluated (and then that node's host node, and so forth) until a match is found or the "HostedOn" relationship chain ends.

4.2 Environment Variable Conventions

4.2.1 Reserved Environment Variable Names and Usage

TOSCA orchestrators utilize certain reserved keywords in the execution environments that implementation artifacts for Node or Relationship Templates operations are executed in. They are used to provide information to these implementation artifacts such as the results of TOSCA function evaluation or information about the instance model of the TOSCA application

The following keywords are reserved environment variable names in any TOSCA supported execution environment:

Keyword	Valid Contexts	Description
TARGETS	Relationship Template only.	• For an implementation artifact that is executed in the context of a relationship, this keyword, if present, is used to supply a list of Node Template instances in a TOSCA application's instance model that are currently target of the context relationship. • The value of this environment variable will be a comma-separated list of identifiers of the single target node instances (i.e., the tosca_id attribute of the node).

Keyword	Valid Contexts	Description
TARGET	Relationship Template only.	• For an implementation artifact that is executed in the context of a relationship, this keyword, if present, identifies a Node Template instance in a TOSCA application's instance model that is a target of the context relationship, and which is being acted upon in the current operation. • The value of this environment variable will be the identifier of the single target node instance (i.e., the tosca_id attribute of the node).
SOURCES	Relationship Template only.	• For an implementation artifact that is executed in the context of a relationship, this keyword, if present, is used to supply a list of Node Template instances in a TOSCA application's instance model that are currently source of the context relationship. • The value of this environment variable will be a comma-separated list of identifiers of the single source node instances (i.e., the tosca_id attribute of the node).
SOURCE	Relationship Template only.	• For an implementation artifact that is executed in the context of a relationship, this keyword, if present, identifies a Node Template instance in a TOSCA application's instance model that is a source of the context relationship, and which is being acted upon in the current operation. • The value of this environment variable will be the identifier of the single source node instance (i.e., the tosca_id attribute of the node).

For scripts (or implementation artifacts in general) that run in the context of relationship operations, select properties and attributes of both the relationship itself as well as select properties and attributes of the source and target node(s) of the relationship can be provided to the environment by declaring respective operation inputs.

Declared inputs from mapped properties or attributes of the source or target node (selected via the SOURCE or TARGET keyword) will be provided to the environment as variables having the exact same name as the inputs. In addition, the same values will be provided for the complete set of source or target nodes, however prefixed with the ID if the respective nodes. By means of the SOURCES or TARGETS variables holding the complete set of source or target node IDs, scripts will be able to iterate over corresponding inputs for each provided ID prefix.

The following example snippet shows an imaginary relationship definition from a load-balancer node to worker nodes. A script is defined for the add_target operation of the Configure interface of the relationship, and the ip_address attribute of the target is specified as input to the script:

```
node_templates:
  load_balancer:
    type: some.vendor.LoadBalancer
    requirements:
      - member:
          relationship: some.vendor.LoadBalancerToMember
            interfaces:
              Configure:
                add_target:
                  inputs:
                    member_ip: { get_attribute: [ TARGET, ip_address ] }
                  implementation: scripts/configure_members.py
```

The add_target operation will be invoked, whenever a new target member is being added to the load-balancer. With the above inputs declaration, a member_ip environment variable that will hold the IP address of the target being added will be provided to the configure_members.py script. In addition, the IP addresses of all current load-balancer members will be provided as environment variables with a naming scheme of <target node ID>_member_ip. This will allow, for example, scripts that always just write the

complete list of load-balancer members into a configuration file to do so instead of updating existing list, which might be more complicated.

Assuming that the TOSCA application instance includes five load-balancer members, node1 through node5, where node5 is the current target being added, the following environment variables (plus potentially more variables) will be provided to the script:

```
# the ID of the current target and the IDs of all targets
TARGET=node5
TARGETS=node1,node2,node3,node4,node5

# the input for the current target and the inputs of all targets
member_ip=10.0.0.5
node1_member_ip=10.0.0.1
node2_member_ip=10.0.0.2
node3_member_ip=10.0.0.3
node4_member_ip=10.0.0.4
node5_member_ip=10.0.0.5
```

With code like shown in the snippet below, scripts could then iterate of all provided member_ip inputs:

```
#!/usr/bin/python
import os

targets = os.environ['TARGETS'].split(',')

for t in targets:
  target_ip = os.environ.get('%s_member_ip' % t)
  # do something with target_ip ...
```

4.2.2 Prefixed vs. Unprefixed TARGET names

The list target node types assigned to the TARGETS key in an execution environment will have names prefixed by unique IDs that distinguish different instances of a node in a running model Future drafts of this specification will show examples of how these names/IDs will be expressed.

4.2.2.1 Notes

- Target of interest is always un-prefixed. Prefix is the target opaque ID. The IDs can be used to find the environment var. for the corresponding target. Need an example here.
- If you have one node that contains multiple targets this would also be used (add or remove target operations would also use this you would get set of all current targets).

4.3 Intrinsic functions

These functions are supported within the TOSCA template for manipulation of template data.

4.3.1 concat

The concat function is used to concatenate two or more string values within a TOSCA service template.

4.3.1.1 Grammar

```
concat: [<string_value_expressions_*> ]
```

4.3.1.2 Parameters

Parameter	Required	Type	Description
<string_value_expressions_*>	yes	list of string or string value expressions	A list of one or more strings (or expressions that result in a string value) which can be concatenated together into a single string.

4.3.1.3 Examples

```
outputs:
  description: Concatenate the URL for a server from other template values
  server_url:
  value: { concat: [ 'http://',
                     get_attribute: [ server, public_address ],
                     ':',
                     get_attribute: [ server, port ] ] }
```

4.3.2 join

The join function is used to join an array of strings into a single string with optional delimiter.

4.3.2.1 Grammar

join: [<list of string_value_expressions_*> [<delimiter>]]

4.3.2.2 Parameters

Parameter	Required	Type	Description
<list of string_value_expressions_*>	yes	list of string or string value expressions	A list of one or more strings (or expressions that result in a list of string values) which can be joined together into a single string.
<delimiter>	no	string	An optional delimiter used to join the string in the provided list.

4.3.2.3 Examples

```
outputs:
  example1:
      # Result: prefix_1111_suffix
      value: { join: [ ["prefix", 1111, "suffix" ], "_" ] }
  example2:
      # Result: 9.12.1.10,9.12.1.20
      value: { join: [ { get_input: my_IPs }, "," ] }
```

4.3.3 token

The token function is used within a TOSCA service template on a string to parse out (tokenize) substrings separated by one or more token characters within a larger string.

4.3.3.1 Grammar

```
token: [ <string_with_tokens>, <string_of_token_chars>, <substring_index> ]
```

4.3.3.2 Parameters

Parameter	Required	Type	Description
string_with_tokens	yes	string	The composite string that contains one or more substrings separated by token characters.
string_of_token_chars	yes	string	The string that contains one or more token characters that separate substrings within the composite string.
substring_index	yes	integer	The integer indicates the index of the substring to return from the composite string. Note that the first substring is denoted by using the '0' (zero) integer value.

4.3.3.3 Examples

```
outputs:
   webserver_port:
     description: the port provided at the end of my server's endpoint's IP
address
     value: { token: [ get_attribute: [ my_server, data_endpoint,
ip_address ],
                       ':',
                       1 ] }
```

4.4 Property functions

The get_input function is used within a service template to obtain template input parameter values. The get_property function is used to get property values from property definitions declared in the same service template (e.g. node or relationship templates).

Note that the get_input and get_property functions may only retrieve the static values of parameter or property definitions of a TOSCA application as defined in the TOSCA Service Template. The get_attribute function should be used to retrieve values for attribute definitions (or property definitions reflected as attribute definitions) from the runtime instance model of the TOSCA application (as realized by the TOSCA orchestrator).

4.4.1 get_input

The get_input function is used to retrieve the values of parameters declared within the inputs section of a TOSCA Service Template.

4.4.1.1 Grammar

```
get_input: <input_parameter_name>
```

or

```
get_input: [ <input_parameter_name>,
<nested_input_parameter_name_or_index_1>, ...,
<nested_input_parameter_name_or_index_n> ]
```

4.4.1.2 Parameters

Parameter	Required	Type	Description
<input_parameter_name>	yes	string	The name of the parameter as defined in the inputs section of the service template.
<nested_input_paratmer_name_or_index_*>	no	string|integer	Some TOSCA input parameters are complex (i.e., composed as nested structures). These parameters are used to dereference into the names of these nested structures when needed. Some parameters represent list types. In these cases, an index may be provided to reference a specific entry in the list (as identified by the previous parameter) to return.

4.4.1.3 Examples

The following snippet shows an example of the simple get_input grammar:

```
inputs:
  cpus:
    type: integer

node_templates:
  my_server:
    type: tosca.nodes.Compute
    capabilities:
      host:
        properties:
          num_cpus: { get_input: cpus }
```

The following template shows an example of the nested get_input grammar. The template expects two input values, each of which has a complex data type. The get_input function is used to retrieve individual fields from the complex input data.

```
data_types:
  NetworkInfo:
    derived_from: tosca.Data.Root
    properties:
      name:
        type: string
      gateway:
        type: string

  RouterInfo:
    derived_from: tosca.Data.Root
    properties:
      ip:
        type: string
      external:
        type: string

topology_template:
  inputs:
    management_network:
      type: NetworkInfo
    router:
      type: RouterInfo

  node_templates:
    Bono_Main:
      type: vRouter.Cisco
      directives: [ substitutable ]
```

```
        properties:
          mgmt_net_name: { get_input: [management_network, name]}
          mgmt_cp_v4_fixed_ip: { get_input: [router, ip]}
          mgmt_cp_gateway_ip: { get_input: [management_network, gateway]}
          mgmt_cp_external_ip: { get_input: [router, external]}
        requirements:
        - lan_port:
            node: host_with_net
            capability: virtualBind
        - mgmt_net: mgmt_net
```

4.4.2 get_property

The get_property function is used to retrieve property values between modelable entities defined in the same service template.

4.4.2.1 Grammar

```
get_property: [ <modelable_entity_name>, <optional_req_or_cap_name>,
<property_name>, <nested_property_name_or_index_1>, ...,
<nested_property_name_or_index_n> ]
```

4.4.2.2 Parameters

Parameter	Required	Type	Description
<modelable entity name> | SELF | SOURCE | TARGET | HOST	yes	string	The required name of a modelable entity (e.g., Node Template or Relationship Template name) as declared in the service template that contains the property definition the function will return the value from. See section B.1 for valid keywords.
<optional_req_or_cap_name>	no	string	The optional name of the requirement or capability name within the modelable entity (i.e., the <modelable_entity_name> which contains the property definition the function will return the value from. **Note**: If the property definition is located in the modelable entity directly, then this parameter MAY be omitted.
<property_name>	yes	string	The name of the property definition the function will return the value from.
<nested_property_name_or_index_*>	no	string|integer	Some TOSCA properties are complex (i.e., composed as nested structures). These parameters are used to dereference into the names of these nested structures when needed. Some properties represent list types. In these cases, an index may be provided to reference a specific entry in the list (as identified by the previous parameter) to return.

4.4.2.3 Examples

The following example shows how to use the get_property function with an actual Node Template name:

```
node_templates:

  mysql_database:
    type: tosca.nodes.Database
    properties:
      name: sql_database1
```

```
        wordpress:
          type: tosca.nodes.WebApplication.WordPress
          ...
          interfaces:
            Standard:
              configure:
                inputs:
                  wp_db_name: { get_property: [ mysql_database, name ] }
```

The following example shows how to use the get_property function using the SELF keyword:

```
      node_templates:

        mysql_database:
          type: tosca.nodes.Database
          ...
          capabilities:
            database_endpoint:
              properties:
                port: 3306

        wordpress:
          type: tosca.nodes.WebApplication.WordPress
          requirements:
            ...
            - database_endpoint: mysql_database
          interfaces:
            Standard:
              create: wordpress_install.sh
              configure:
                implementation: wordpress_configure.sh
                inputs:
                  ...
                  wp_db_port: { get_property: [ SELF, database_endpoint, port ] }
```

The following example shows how to use the get_property function using the TARGET keyword:

```
      relationship_templates:
          my_connection:
            type: ConnectsTo
            interfaces:
              Configure:
                inputs:
                  targets_value: { get_property: [ TARGET, value ] }
```

4.5 Attribute functions

These functions (attribute functions) are used within an instance model to obtain attribute values from instances of nodes and relationships that have been created from an application model described in a service template. The instances of nodes or relationships can be referenced by their name as assigned in the service template or relative to the context where they are being invoked.

4.5.1 get_attribute

The **get_attribute** function is used to retrieve the values of named attributes declared by the referenced node or relationship template name.

4.5.1.1 Grammar

```
get_attribute: [ <modelable_entity_name>, <optional_req_or_cap_name>,
<attribute_name>, <nested_attribute_name_or_index_1>, ...,
<nested_attribute_name_or_index_n> ]
```

4.5.1.2 Parameters

Parameter	Required	Type	Description
<modelable entity name> | SELF | SOURCE | TARGET | HOST	yes	string	The required name of a modelable entity (e.g., Node Template or Relationship Template name) as declared in the service template that contains the attribute definition the function will return the value from. See section B.1 for valid keywords.
<optional_req_or_cap_name>	no	string	The optional name of the requirement or capability name within the modelable entity (i.e., the <modelable_entity_name> which contains the attribute definition the function will return the value from. **Note**: If the attribute definition is located in the modelable entity directly, then this parameter MAY be omitted.
<attribute_name>	yes	string	The name of the attribute definition the function will return the value from.
<nested_attribute_name_or_index_*>	no	string|integer	Some TOSCA attributes are complex (i.e., composed as nested structures). These parameters are used to dereference into the names of these nested structures when needed. Some attributes represent list types. In these cases, an index may be provided to reference a specific entry in the list (as identified by the previous parameter) to return.

4.5.1.3 Examples:

The attribute functions are used in the same way as the equivalent Property functions described above. Please see their examples and replace "get_property" with "get_attribute" function name.

4.5.1.4 Notes

These functions are used to obtain attributes from instances of node or relationship templates by the names they were given within the service template that described the application model (pattern).

- These functions only work when the orchestrator can resolve to a single node or relationship instance for the named node or relationship. This essentially means this is acknowledged to work only when the node or relationship template being referenced from the service template has a cardinality of 1 (i.e., there can only be one instance of it running).

4.6 Operation functions

These functions are used within an instance model to obtain values from interface operations. These can be used in order to set an attribute of a node instance at runtime or to pass values from one operation to another.

4.6.1 get_operation_output

The **get_operation_output** function is used to retrieve the values of variables exposed / exported from an interface operation.

4.6.1.1 Grammar

```
get_operation_output: <modelable_entity_name>, <interface_name>,
<operation_name>, <output_variable_name>
```

4.6.1.2 Parameters

Parameter	Required	Type	Description
<modelable entity name> | SELF | SOURCE | TARGET	yes	string	The required name of a modelable entity (e.g., Node Template or Relationship Template name) as declared in the service template that implements the interface and operation.
<interface_name>	Yes	string	The required name of the interface which defines the operation.
<operation_name>	yes	string	The required name of the operation whose value we would like to retrieve.
<output_variable_name>	Yes	string	The required name of the variable that is exposed / exported by the operation.

4.6.1.3 Notes

- If operation failed, then ignore its outputs. Orchestrators should allow orchestrators to continue running when possible past deployment in the lifecycle. For example, if an update fails, the application should be allowed to continue running and some other method will be used to alert administrators of the failure.

4.7 Navigation functions

- This version of TOSCA does not define any model navigation functions.

4.7.1 get_nodes_of_type

The **get_nodes_of_type** function can be used to retrieve a list of all known instances of nodes of the declared Node Type.

4.7.1.1 Grammar

```
get_nodes_of_type: <node_type_name>
```

4.7.1.2 Parameters

Parameter	Required	Type	Description
<node_type_name>	yes	string	The required name of a Node Type that a TOSCA orchestrator will use to search a running application instance in order to return all unique, named node instances of that type.

4.7.1.3 Returns

Return Key	Type	Description
TARGETS	<see above>	The list of node instances from the current application instance that match the node_type_name supplied as an input parameter of this function.

4.8 Artifact functions

4.8.1 get_artifact

The get_artifact function is used to retrieve artifact location between modelable entities defined in the same service template.

4.8.1.1 Grammar

```
get_artifact: [ <modelable_entity_name>, <artifact_name>, <location>,
<remove> ]
```

4.8.1.2 Parameters

Parameter	Required	Type	Description
<modelable entity name> | SELF | SOURCE | TARGET | HOST	yes	string	The required name of a modelable entity (e.g., Node Template or Relationship Template name) as declared in the service template that contains the named property definition the function will return the value from. See section B.1 for valid keywords.
<artifact_name>	yes	string	The name of the artifact definition the function will return the value from.
<location> | LOCAL_FILE	no	string	Location value must be either a valid path e.g. '/etc/var/my_file' or 'LOCAL_FILE'. If the value is LOCAL_FILE the orchestrator is responsible for providing a path as the result of the get_artifact call where the artifact file can be accessed. The orchestrator will also remove the artifact from this location at the end of the operation. If the location is a path specified by the user the orchestrator is responsible to copy the artifact to the specified location. The orchestrator will return the path as the value of the get_artifact function and leave the file here after the execution of the operation.
remove	no	boolean	Boolean flag to override the orchestrator default behavior so it will remove or not the artifact at the end of the operation execution. If not specified the removal will depends of the location e.g. removes it in case of 'LOCAL_FILE' and keeps it in case of a path. If true the artifact will be removed by the orchestrator at the end of the operation execution, if false it will not be removed.

4.8.1.3 Examples

The following example uses a snippet of a WordPress [WordPress] web application to show how to use the get_artifact function with an actual Node Template name:

4.8.1.3.1 Example: Retrieving artifact without specified location

```
node_templates:

  wordpress:
    type: tosca.nodes.WebApplication.WordPress
```

```
        ...
        interfaces:
          Standard:
            configure:
              create:
                implementation: wordpress_install.sh
                inputs
                  wp_zip: { get_artifact: [ SELF, zip ] }
        artifacts:
          zip: /data/wordpress.zip
```

In such implementation the TOSCA orchestrator may provide the wordpress.zip archive as

- a local URL (example: file://home/user/wordpress.zip) or
- a remote one (example: http://cloudrepo:80/files/wordpress.zip) where some orchestrator may indeed provide some global artifact repository management features.

4.8.1.3.2 Example: Retrieving artifact as a local path

The following example explains how to force the orchestrator to copy the file locally before calling the operation's implementation script:

```
node_templates:

  wordpress:
    type: tosca.nodes.WebApplication.WordPress
    ...
    interfaces:
      Standard:
        configure:
          create:
            implementation: wordpress_install.sh
            inputs
              wp_zip: { get_artifact: [ SELF, zip, LOCAL_FILE] }
        artifacts:
          zip: /data/wordpress.zip
```

In such implementation the TOSCA orchestrator must provide the wordpress.zip archive as a local path (example: /tmp/wordpress.zip) and **will remove it** after the operation is completed.

4.8.1.3.3 Example: Retrieving artifact in a specified location

The following example explains how to force the orchestrator to copy the file locally to a specific location before calling the operation's implementation script :

```
node_templates:

  wordpress:
    type: tosca.nodes.WebApplication.WordPress
    ...
    interfaces:
      Standard:
        configure:
          create:
            implementation: wordpress_install.sh
            inputs
              wp_zip: { get_artifact: [ SELF, zip, C:/wpdata/wp.zip ] }
        artifacts:
          zip: /data/wordpress.zip
```

In such implementation the TOSCA orchestrator must provide the wordpress.zip archive as a local path (example: C:/wpdata/wp.zip) and **will let it** after the operation is completed.

4.9 Context-based Entity names (global)

Future versions of this specification will address methods to access entity names based upon the context in which they are declared or defined.

4.9.1.1 Goals

- Using the full paths of modelable entity names to qualify context with the future goal of a more robust get_attribute function: e.g., get_attribute(<context-based-entity-name>, <attribute name>)

5 TOSCA Cloud Service Archive (CSAR) format

This section defines the metadata of a cloud service archive as well as its overall structure. Except for the examples, this section is **normative.**

5.1 Overall Structure of a CSAR

A CSAR is a zip file where TOSCA definitions along with all accompanying artifacts (e.g. scripts, binaries, configuration files) can be packaged together. A CSAR zip file MUST contain one of the following:

- A **TOSCA.meta** metadata file that provides entry information for a TOSCA orchestrator processing the CSAR file. The **TOSCA.meta** file may be located either at the root of the archive or inside a **TOSCA-Metadata** directory (the directory being at the root of the archive). The CSAR may contain only one **TOSCA.meta** file.
- a yaml (.yml or .yaml) file at the root of the archive, the yaml file being a valid tosca definition template.

The CSAR file MAY contain other directories and files with arbitrary names and contents.

5.2 TOSCA Meta File

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.

```
<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 is called a block. Blocks are separated by an empty line. The first block, called block_0, contains metadata about the CSAR itself and is further defined below. Other blocks may be used to represent custom generic metadata or metadata pertaining to files in the CSAR. A **TOSCA.meta** file is only required to include block_0. The structure of block_0 in the TOSCA meta file is as follows:

```
CSAR-Version: digit.digit
Created-By: string
Entry-Definitions: string
Other-Definitions: string
```

The name/value pairs are as follows:

- **CSAR-Version:** This is the version number of the CSAR specification. It defines the structure of the CSAR and the format of the **TOSCA.meta** file. The value MUST be "2.0" for this version of the CSAR specification.
- **Created-By**: The person or organization that created the CSAR.
- **Entry-Definitions**: This references the TOSCA definitions file that SHOULD be used as entry point for processing the contents of the CSAR (i.e. the main TOSCA service template).
- **Other-Definitions**: This references an unambiguous set of files containing substitution templates that can be used to implement nodes defined in the main template (i.e. the file declared in **Entry-Definitions**). Thus, all the topology templates defined in files listed under the **Other-Definitions** key are to be used only as substitution templates, and not as standalone services. If such a topology template cannot act as a substitution template, it will be ignored by the orchestrator. The value of the **Other-Definitions** key is a string containing a list of filenames (relative to the root of the CSAR

archive) delimited by a blank space. If the filenames contain blank spaces, the filename should be enclosed by double quotation marks (")

Note that any further TOSCA definitions files required by the definitions specified by **Entry-Definitions** or **Other-Definitions** can be found by a TOSCA orchestrator by processing respective **imports** statements. Note also that artifact files (e.g. scripts, binaries, configuration files) used by the TOSCA definitions and included in the CSAR are fully described and referred via relative path names in artifact definitions in the respective TOSCA definitions files contained in the CSAR.

5.2.1 Custom keynames in the TOSCA.meta file

Besides using the normative keynames in block_0 (i.e. CSAR-Version, Created-By, Entry-Definitions, Other-Definitions) users can populate further blocks in the **TOSCA.meta** file with custom key-value pairs that follow the entry syntax of the **TOSCA.meta** file, but which are outside the scope of the TOSCA specification.

Nevertheless, future versions of the TOSCA specification may add definitions of new keynames to be used in the **TOSCA.meta** file. In case of a keyname collision (with a custom keyname) the TOSCA specification definitions take precedence.

To minimize such keyname collisions the specification reserves the use of keynames starting with TOSCA and tosca. It is recommended as a good practice to use a specific prefix (e.g. identifying the organization, scope, etc.) when using custom keynames.

5.2.2 Example

The following listing represents a valid **TOSCA.meta** file according to this TOSCA specification.

```
CSAR-Version: 2.0
Created-By: OASIS TOSCA TC
Entry-Definitions: tosca_elk.yaml
Other-Definitions: definitions/tosca_moose.yaml definitions/tosca_deer.yaml
```

This **TOSCA.meta** file indicates its structure (as well as the overall CSAR structure) by means of the **CSAR-Version** keyname with value **2.0**. The **Entry-Definitions** keyname points to a TOSCA definitions YAML file with the name **tosca_elk.yaml** which is contained in the root of the CSAR file. Additionally, it specifies that substitution templates can be found in the files **tosca_moose.yaml** and **tosca_deer.yaml** found in the directory called **definitions** in the root of the CSAR file.

5.3 Archive without TOSCA-Metadata

In case the archive doesn't contains a **TOSCA.meta** file the archive is required to contains a single YAML file at the root of the archive (other templates may exist in sub-directories).

TOSCA processors should recognize this file as being the CSAR Entry-Definitions file. The CSAR-Version is inferred from the tosca_definitions_version keyname in the Entry-Definitions file. For tosca_definitions_version: tosca_2_0 and onwards, the corresponding CSAR-version is 2.0 unless further defined.

Note that in a CSAR without TOSCA-metadata it is not possible to unambiguously include definitions for substitution templates as we can have only one topology template defined in a yaml file.

5.3.1 Example

The following represents a valid TOSCA template file acting as the CSAR Entry-Definitions file in an archive without TOSCA-Metadata directory.

```
tosca_definitions_version: tosca_2_0
```

```
metadata:
  template_name: my_template
  template_author: OASIS TOSCA TC
  template_version: 1.0
```

6 Security Considerations

(**Note:** OASIS strongly recommends that Technical Committees consider issues that could affect security when implementing their specification and document them for implementers and adopters. For some purposes, you may find it required, e.g. if you apply for IANA registration.

While it may not be immediately obvious how your specification might make systems vulnerable to attack, most specifications, because they involve communications between systems, message formats, or system settings, open potential channels for exploit. For example, IETF [RFC3552] lists "eavesdropping, replay, message insertion, deletion, modification, and man-in-the-middle" as well as potential denial of service attacks as threats that must be considered and, if appropriate, addressed in IETF RFCs.

In addition to considering and describing foreseeable risks, this section should include guidance on how implementers and adopters can protect against these risks.

We encourage editors and TC members concerned with this subject to read *Guidelines for Writing RFC Text on Security Considerations,* IETF [RFC3552], for more information.)

7 Conformance

7.1 Conformance Targets

The implementations subject to conformance are those introduced in Section 11.3 "Implementations". They are listed here for convenience:

- TOSCA YAML service template
- TOSCA processor
- TOSCA orchestrator (also called orchestration engine)
- TOSCA generator
- TOSCA archive

7.2 Conformance Clause 1: TOSCA YAML service template

A document conforms to this specification as TOSCA YAML service template if it satisfies all the statements below:

1. It is valid according to the grammar, rules and requirements defined in section 3 "TOSCA definitions in YAML".
2. When using functions defined in section 4 "TOSCA functions", it is valid according to the grammar specified for these functions.
3. When using or referring to data types, artifact types, capability types, interface types, node types, relationship types, group types, policy types defined in section 5 "TOSCA normative type definitions", it is valid according to the definitions given in section 5.

7.3 Conformance Clause 2: TOSCA processor

A processor or program conforms to this specification as TOSCA processor if it satisfies all the statements below:

1. It can parse and recognize the elements of any conforming TOSCA YAML service template, and generates errors for those documents that fail to conform as TOSCA YAML service template while clearly intending to.
2. It implements the requirements and semantics associated with the definitions and grammar in section 3 "TOSCA definitions in YAML", including those listed in the "additional requirements" subsections.
3. It resolves the imports, either explicit or implicit, as described in section 3 "TOSCA definitions in YAML".
4. It generates errors as required in error cases described in sections 3.1 (TOSCA Namespace URI and alias), 3.2 (Parameter and property type) and 3.6 (Type-specific definitions).
5. It normalizes string values as described in section 5.4.9.3 (Additional Requirements)

7.4 Conformance Clause 3: TOSCA orchestrator

A processor or program conforms to this specification as TOSCA orchestrator if it satisfies all the statements below:

1. It is conforming as a TOSCA Processor as defined in conformance clause 2: TOSCA Processor.
2. It can process all types of artifact described in section 5.3 "Artifact types" according to the rules and grammars in this section.
3. It can process TOSCA archives as intended in section 6 "TOSCA Cloud Service Archive (CSAR) format" and other related normative sections.
4. It can understand and process the functions defined in section 4 "TOSCA functions" according to their rules and semantics.
5. It can understand and process the normative type definitions according to their semantics and requirements as described in section 5 "TOSCA normative type definitions".
6. It can understand and process the networking types and semantics defined in section 7 "TOSCA Networking".
7. It generates errors as required in error cases described in sections 2.10 (Using node template substitution for chaining subsystems), 5.4 (Capabilities Types) and 5.7 (Interface Types).).

7.5 Conformance Clause 4: TOSCA generator

A processor or program conforms to this specification as TOSCA generator if it satisfies at least one of the statements below:

1. When requested to generate a TOSCA service template, it always produces a conforming TOSCA service template, as defined in Clause 1: TOSCA YAML service template,
2. When requested to generate a TOSCA archive, it always produces a conforming TOSCA archive, as defined in Clause 5: TOSCA archive.

7.6 Conformance Clause 5: TOSCA archive

A package artifact conforms to this specification as TOSCA archive if it satisfies all the statements below:

1. It is valid according to the structure and rules defined in section 6 "TOSCA Cloud Service Archive (CSAR) format".

Appendix A. Acknowledgments

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

Participants:
Alex Vul (alex.vul@intel.com), Intel
Anatoly Katzman (anatoly.katzman@att.com), AT&T
Arturo Martin De Nicolas (arturo.martin-de-nicolas@ericsson.com), Ericsson
Avi Vachnis (avi.vachnis@alcatel-lucent.com), Alcatel-Lucent
Calin Curescu (calin.curescu@ericsson.com), Ericsson
Chris Lauwers (lauwers@ubicity.com)
Claude Noshpitz (claude.noshpitz@att.com), AT&T
Derek Palma (dpalma@vnomic.com), Vnomic
Dmytro Gassanov (dmytro.gassanov@netcracker.com), NetCracker
Frank Leymann (Frank.Leymann@informatik.uni-stuttgart.de), Univ. of Stuttgart
Gábor Marton (gabor.marton@nokia.com), Nokia
Gerd Breiter (gbreiter@de.ibm.com), IBM
Hemal Surti (hsurti@cisco.com), Cisco
Ifat Afek (ifat.afek@alcatel-lucent.com), Alcatel-Lucent
Idan Moyal, (idan@gigaspaces.com), Gigaspaces
Jacques Durand (jdurand@us.fujitsu.com), Fujitsu
Jin Qin, (chin.qinjin@huawei.com), Huawei
Jeremy Hess, (jeremy@gigaspaces.com), Gigaspaces
John Crandall, (mailto:jcrandal@brocade.com), Brocade
Juergen Meynert (juergen.meynert@ts.fujitsu.com), Fujitsu
Kapil Thangavelu (kapil.thangavelu@canonical.com), Canonical
Karsten Beins (karsten.beins@ts.fujitsu.com), Fujitsu
Kevin Wilson (kevin.l.wilson@hp.com), HP
Krishna Raman (kraman@redhat.com), Red Hat
Luc Boutier (luc.boutier@fastconnect.fr), FastConnect
Luca Gioppo, (luca.gioppo@csi.it), CSI-Piemonte
Matej Artač, (matej.artac@xlab.si), XLAB
Matt Rutkowski (mrutkows@us.ibm.com), IBM
Moshe Elisha (moshe.elisha@alcatel-lucent.com), Alcatel-Lucent
Nate Finch (nate.finch@canonical.com), Canonical
Nikunj Nemani (nnemani@vmware.com), Wmware
Priya TG (priya.g@netcracker.com) NetCracker
Richard Probst (richard.probst@sap.com), SAP AG
Sahdev Zala (spzala@us.ibm.com), IBM
Shitao li (lishitao@huawei.com), Huawei
Simeon Monov (sdmonov@us.ibm.com), IBM
Sivan Barzily, (sivan@gigaspaces.com), Gigaspaces
Sridhar Ramaswamy (sramasw@brocade.com), Brocade
Stephane Maes (stephane.maes@hp.com), HP
Steve Baillargeon (steve.baillargeon@ericsson.com), Ericsson
Thinh Nguyenphu (thinh.nguyenphu@nokia.com), Nokia
Thomas Spatzier (thomas.spatzier@de.ibm.com), IBM
Ton Ngo (ton@us.ibm.com), IBM
Travis Tripp (travis.tripp@hp.com), HP

Vahid Hashemian (vahidhashemian@us.ibm.com), IBM
Wayne Witzel (wayne.witzel@canonical.com), Canonical
Yaron Parasol (yaronpa@gigaspaces.com), Gigaspaces

Appendix B. Example Title

text

B.1 Subsidiary section

text

B.1.1 Sub-subsidiary section

Text

B.1.1.1 Sub-sub-subsidiary section

text

B.1.1.1.1 Sub-sub-sub-subsidiary section

text

Appendix C. Revision History

Revision	Date	Editor	Changes Made
WD01, Rev01	2019-04-01	Chris Lauwers	Initial WD01, Revision 01 baseline for TOSCA v2.0
WD01, Rev02	2019-04-22	Chris Lauwers	Split of introductory chapters into the *Introduction to TOSCA Version 2.0* document.
WD01, Rev03	2019-05-08	Calin Curescu	Incorporate fixes from latest v1.3 specification
WD01, Rev04	2019-05-10	Chris Lauwers	Fix syntax of schema constraint examples (Sections 5.3.2 and 5.3.4)
WD01, Rev05	2019-08-30	Chris Lauwers	Cleanup formatting. No content changes.
WD01, Rev06	2019-08-30	Chris Lauwers	• Remove 3.6.20.3 since it is no longer relevant. • Separate out new Operation Assignment section 3.8.3 from the original Operation Definition section 3.6.17 • Separate out new Notification Assignment section 3.8.4 from the original Notification Definition section 3.6.19 • Separate out new Interface Assignment section 3.8.5 from the original Interface Definition section 3.6.20 • Update the Interface Type definitions in section 5.8 to show the (now mandatory) 'operations' keyname. • Remove erroneous interface definition in tosca.groups.Root type (section 5.10.1) • Added 'description' keyname to Requirement definition (section 3.7.3)
WD01, Rev07	2019-09-08	Calin Curescu	• Added the "value" keyname to property definition (Section 3.6.10 Property Definition), • Updated the refinement rules for property definitions (Section 3.6.10.6 Refining Property Definitions). • Added the refinement rules for attribute definitions (Section 3.6.12.7 Refining Attribute Definitions) • Made the difference between outgoing and incoming parameters in the parameter definition (Section 3.6.14 Parameter definition) • Added the "mapping" keyname to the parameter definition, for mapping the incoming parameter to an attribute (Section 3.6.14 Parameter definition) • Added refinement rules for parameter definitions (Section 3.6.14.5 Refining Parameter Definitions) • Changed the wrong usage of "property definitions" and "property assignments" instead of "parameter definitions" and "parameter assignments" throughout the document. For example, a larger impact can be seen in the definition of the get_input function (Section 4.4.1 get_input). • Changed Section "3.6.16 Operation implementation definition" to include notification implementation definition (Section 3.6.16 Operation implementation definition and notification implementation definition). • Deleted Section "3.6.18 Notification implementation definition" since it was redundant and all relevant information has been transferred to Section "3.6.16 Operation implementation definition and notification

			implementation definition". The "Notification definition" section becomes the new Section 3.6.18. • Added operation refinement rules to the operation definition section (Section 3.6.17 Operation definition). • Added operation assignment rules to the operation assignment section (Section 3.8.3 Operation Assignment). • Added notification refinement rules to the notification definition section (Section 3.6.18 Notification definition) • Added notification assignment rules to the notification assignment section (Section 3.8.4 Notification assignment). • Added interface refinement rules to the interface definition section (Section 3.6.19 Interface definition) • Added interface assignment rules to the interface assignment section (Section 3.8.5 Interface assignment). • Changed "interface definitions" with "interface assignments" in the node template specification, given that we have split interface assignments from interface definitions (Section 3.8.6 Node Template) • Changed "interface definitions" with "interface assignments" in the relationship template specification, given that we have split interface assignments from interface definitions (Section 3.8.7 Relationship Template)
WD01, Rev08	2019-09-30	Chris Lauwers	Fix error in TimeInterval example (Section 5.3.7.3.1)
WD01, Rev09	2020-02-20	Chris Lauwers	• Move normative type definitions to the "Intro to TOSCA" document • Move non-normative type definitions to the "Intro to TOSCA" document • Move "CSAR" specification from the "intro to TOSCA" document into this document
WD01, Rev10	2020-04-15	Calin Curescu	• Reorganized sections into a new layout (starting with the main concepts).