

UML to YANG Mapping Guidelines

Draft TR-531 v1.1.03

August 29, 2023

Work in progress!

**Disclaimer**

THIS SPECIFICATION IS PROVIDED “AS IS” WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

Any marks and brands contained herein are the property of their respective owners.

Open Networking Foundation

1000 El Camino Real, Suite 100, Menlo Park, CA 94025

[www.opennetworking.org](http://www.opennetworking.org)

©2018 Open Networking Foundation. All rights reserved.

Open Networking Foundation, the ONF symbol, and OpenFlow are registered trademarks of the Open Networking Foundation, in the United States and/or in other countries. All other brands, products, or service names are or may be trademarks or service marks of, and are used to identify, products or services of their respective owners.

**Important note**

This Technical Recommendations has been approved by the OIMT Project TST but has not been approved by the ONF board. This Technical Recommendation has been approved under the ONF publishing guidelines for 'Informational' publications that allow Project technical steering teams (TSTs) to authorize publication of Informational documents. The designation of '-info' at the end of the document ID also reflects that the project team (not the ONF board) approved this TR.

Table of Contents

1 Introduction 7

2 References 7

3 Abbreviations 8

4 Overview 9

4.1 Documentation Overview 9

5 Mapping Guidelines 9

5.1 Introduction 9

5.2 Generic Mapping Guidelines 10

5.2.1 Naming Conventions Mapping 10

5.2.2 Mapping Scope 11

5.2.3 YANG Workarounds 11

5.3 Mapping of Classes 11

5.4 Mapping of Attributes 19

5.5 Mapping of Data Types 24

5.5.1 Generic Mapping of Primitive Data Types 24

5.5.2 Generic Mapping of Complex Data Types 25

5.5.3 Mapping of Common Primitive and Complex Data Types 26

5.5.4 Mapping of Enumeration Types 31

5.5.5 Mapping of Bit Field Data Types 34

5.6 Mapping of Relationships 35

5.6.1 Mapping of Associations 35

5.6.2 Mapping of Dependencies 41

5.7 Mapping of Interfaces (grouping of operations) 45

5.8 Mapping of Operations 46

5.9 Mapping of Operation Parameters 50

5.10 Mapping of Notifications 52

5.11 Mapping of UML Packages 55

5.12 Mapping of Lifecycle 55

5.13 Mapping Issues 56

5.13.1 YANG 1.0 or YANG 1.1? 56

5.13.2 Combination of different Associations? 56

6 Mapping Patterns 57

6.1 UML Recursion 57

6.1.1 Reference Based Approach 57

6.2 UML Conditional Pacs 59

6.3 {xor} Constraint 60

6.4 «Choice» Stereotype (obsolete) 63

6.5 Mapping of UML Support and Condition Properties 65

6.6 Proxy Class Association Mapping 67

6.7 Building YANG Tree 67

7 Generic UML Model and specific YANG Configuration Information 69

7.1 YANG Module Header 69

7.2 Lifecycle State Treatment 71

8 Mapping Basics 73

8.1 UML 🡪 YANG or XMI 🡪 YANG 73

8.2 Open Model Profile YANG Extensions 74

9 Reverse Mapping From YANG to UML 76

10 Requirements for the YANG Module Structure 76

11 Main Changes between Releases 78

11.1 Summary of main changes between version 1.0 and 1.1 78

12 Proposed Addendum 2 79

List of Figures

Figure 4.1: Specification Architecture 9

Figure 5.1: Pre-defined Packages in a UML Module 55

Figure 6.1: Example: Proxy Class Mapping 67

Figure 8.1: Example UML to YANG Mapping 73

Figure 8.2: Example XMI (Papyrus) to YANG Mapping 74

List of Tables

Table 5.1: Naming Conventions Mapping 10

Table 5.2: Class Mapping (Mappings required by currently used UML artifacts) 12

Table 5.3: Class Mapping (Mappings for remaining YANG substatements) 14

Table 5.4: Class Mapping Examples 15

Table 5.5: Attribute Mapping (Mappings required by currently used UML artifacts) 19

Table 5.6: Attribute Mapping (Mappings for remaining YANG substatements) 21

Table 5.7: Attribute Type Mapping Example 22

Table 5.8: Primitive Data Type Mapping 24

Table 5.9: Complex Data Type Mapping 25

Table 5.10: Complex Data Type Mapping Example 26

Table 5.11: Common Primitive and Complex Data Type Mapping 27

Table 5.12: Enumeration Type Mapping (Mappings required by currently used UML artifacts) 32

Table 5.13: Enumeration Type Mapping Example 33

Table 5.14: Bit Set Encoding Mapping Example 35

Table 5.15: Association Mapping Examples 37

Table 5.16: Association Mapping Summary 41

Table 5.17: Dependency Mapping Examples 42

Table 5.18: UML Interface Mapping 45

Table 5.19: Operation Mapping 46

Table 5.20: Interface/Operation Mapping Example 49

Table 5.21: Operation Exception Mapping Example 50

Table 5.22: Parameter Mapping 50

Table 5.23: Interface/Operation/Parameter Mapping Example 52

Table 5.24: Notification Mapping 52

Table 5.25: Notification Mapping Example 54

Table 5.26: UML Package to YANG Heading Mapping 55

Table 5.27: Lifecycle Mapping 55

Table 5.28: Combination of Associations Mapping Examples 57

Table 6.1: Recursion Mapping Examples 58

Table 6.2: Mapping of Conditional Packages 59

Table 6.3: {xor} Constraint Mapping Examples 60

Table 6.4: «Choice» Stereotype Mapping Examples 64

Table 6.5: Support and Condition Mapping Examples 66

Table 6.6: Composite Aggregation Associations Mapping to YANG Tree Example 68

Document History

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Description of Change** | |
| 1.0 | Sept. 20, 2016 | Initial version. | |
| 1.1 | July 2018 | Version 1.1  A summary of main changes between version 1.0 and 1.1 is contained in section 11.1. | |
| 1.1.01 | Nov. 8, 2018 | Initial version for 1.2 development.  Preliminary solution 1 for bit set modeling deleted. Statement added in section 1 that RFC 8342 NMDA is fulfilled. Mapping of leafrefs refined in section 5.6.1. «StrictComposite» and «LifecycleAggregate» mappings removed. Uses statement in an augmentation wrapped by a container statement. | |
| 1.1.02 | Dec. 12, 2018 | Revision marks from v1.1.01 accepted. Mapping of model name to module name in case the model contains “.” described in section 7.1 | |
| 1.1.03 | Aug. 29, 2023 | Revision marks from v1.1.02 accepted.   * Section on YANG Workarounds deleted * Mapping of read only key added to Table 5.15 * New section 5.6.1.1 Explanation to “Read-write shared association to read-only key” added * Guideline to convert also the artefact names within UML descriptions to YANG style added in Table 5.1. * «ExtendedComposite» example updated in Table 5.15 * Comments/changes regarding «Specify» from Italo added. * «Specify» association mapping example corrected in Table 5.17 and mapping rule added * Enumeration Type Inheritance Mapping Example added in section 5.5.4 (Table 5.14) * Enumeration Type Abstraction Mapping Example added in section 5.5.4 (Table 5.15) * New Appendix: UML to YANG Tool User Guide added |

# Introduction

This Technical Recommendation has been developed within IISOMI (**I**nformal **I**nter-**S**DO **O**pen **M**odel **I**nitiative) and is published by ONF.

IISOMI is an open source project founded by UML model designers from various SDOs like ETSI NFV, ITU-T, MEF, ONF and TM Forum.  
The goal is to develop guidelines and tools for a harmonized modeling infrastructure that is not specific to any SDO, technology or management protocol and can then be used by all SDOs.  
The deliverables are developed in an open source community under the “Creative Commons Attribution 4.0 International Public License”.

This document defines the guidelines for mapping protocol-neutral UML information models to YANG data schemas. The UML information model to be mapped has to be defined based on the UML Modeling Guidelines defined in [7].  
The mapping rules are in line with YANG 1.1 defined in RFC 7950 [11] and the NMDA principles defined in RFC 8342 [13].

In parallel, a tool which automates the mapping from UML 🡪 YANG is being developed in the Open Source SDN community. The current draft version of the tool is available on Github [9]. A video which introduces the UML 🡪 YANG mapping tool is provided in [10].  
The mapping tool is using YANG Version 1.0 (RFC 6020).

Note:  
Mapping in the reverse direction from YANG to UML is possible for the class artifacts but has some issues to be taken into account; see also section 9.

Note:  
This version of the guidelines is still a work in progress! Known open issues are marked in yellow and by comments.

# References

1. [RFC 6020](https://datatracker.ietf.org/doc/rfc6020/) “YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)”
2. Guidelines for Authors and Reviewers of YANG Data Model Documents (<https://wiki.tools.ietf.org/wg/netmod/draft-ietf-netmod-rfc6087bis>)
3. A Guide to NETCONF for SNMP Developers  
   (by Andy Bierman, v0.6 2014-07-10)
4. YANG Central ([http://www.yang-central.org](http://www.yang-central.org/))
5. NetConf Central (<http://www.netconfcentral.org>)
6. YANG patterns (<https://tools.ietf.org/html/draft-schoenw-netmod-yang-pattern>)
7. IISOMI 514 “UML Modeling Guidelines Version 1.3” (<https://3vf60mmveq1g8vzn48q2o71a-wpengine.netdna-ssl.com/wp-content/uploads/2018/08/TR-514_UML_Modeling_Guidelines_v1.3-1-1.pdf>)
8. OpenModelProfile (<https://github.com/OpenNetworkingFoundation/EAGLE-Open-Model-Profile-and-Tools/tree/OpenModelProfile>)
9. EAGLE UML-Yang Mapping Tool (<https://github.com/OpenNetworkingFoundation/EAGLE-Open-Model-Profile-and-Tools/tree/UmlYangTools>)
10. Video to introduce the UML to YANG mapping tool  
    Youtube: <https://www.youtube.com/watch?v=6At3YFrE8Ag&feature=youtu.be>  
    Youku: <http://v.youku.com/v_show/id_XMTQ4NDc2NDg0OA==.html>

1. [RFC 7950](https://datatracker.ietf.org/doc/rfc7950/) “The YANG 1.1 Data Modeling Language”
2. Draft OpenConfig YANG best practices: <http://www.openconfig.net/docs/style-guide/>

1. [RFC 8342](https://tools.ietf.org/html/rfc8342) “Network Management Datastore Architecture (NMDA)”

# Abbreviations

App Application

C Conditional

CM Conditional-Mandatory

CO Conditional-Optional

DN Distinguished Name

DS Data Schema

DSCP Differentiated Services Codepoint

IM Information Model

JSON JavaScript Object Notation

M Mandatory

MAC Media Access Control

NA Not Applicable

O Optional

OF Open Flow

Pac Package

ro read only

RPC Remote Procedure Call

rw read write

SDN Software Defined Network

SMI Structure of Management Information

UML Unified Modeling Language

URI Uniform Resource Identifier

UUID Universally Unique Identifier

XOR Exclusive OR

XMI XML Metadata Interchange

XML Extensible Markup Language

YANG "Yet Another Next Generation".

# Overview

## Documentation Overview

This document is part of a suite of guidelines. The location of this document within the documentation architecture is shown in Figure 4.1 below:



Figure 4.1: Specification Architecture

# Mapping Guidelines

## Introduction

The mapping rules are defined in table format and are structured based on the UML artifacts defined in [7]. Two tables are created for every UML artifact. The first table shows the mapping to YANG for the UML artifacts defined in [7]. The second table shows the potential mapping of the remaining YANG substatements which have not been covered in the first table.  
Example mappings are shown below the mapping tables.

Open issues are either marked in yellow and/or by comments.  
General mapping issues are defined in section 5.13.

## Generic Mapping Guidelines

### Naming Conventions Mapping

UML and YANG use different naming conventions. UML mainly use camel case and YANG use only lower-case letters, numbers, and dashes in identifier names.

The grammar of an identifier in YANG is defined as (from [11]):

(ALPHA / "\_")  
\*(ALPHA / DIGIT / "\_" / "-" / ".")

Where:

* ALPHA = %x41-5A / %x61-7A  
   ; A-Z / a-z
* DIGIT = %x30-39  
   ; 0-9

Note: The restriction that identifiers should not start with the characters "xml" was removed from YANG in version 1.1.

Table 5.1: Naming Conventions Mapping

|  |  |  |
| --- | --- | --- |
| UML Naming | YANG Naming | Comments |
| lowerCamelCase | lower-camel-case |  |
| \_navigableAssociationEnd | navigable-association-end |  |
| UpperCamelCase | upper-camel-case | The Mapping Tool should convert also the artefact names within the UML descriptions to YANG style (i.e., kebab-case). The list of names to be converted in the descriptions is limited to the UML artefact names. |
| LITERAL\_NAME\_1 | enum LITERAL\_NAME\_1 identity LITERAL\_NAME\_1 | literals stay the same |
| Abc\_Pac | abc-pac |  |
| ABC ABCon OMS\_TrailTerminationPoint | a-b-c a-b-con o-m-s-trail-termination-point | abbreviations |
| qom1024Times 15Min | qom-1024-times 15-min | letter strings and numbers |
| AbcClass | grouping: abc-class(-g) identity: abc-class | object classes suffix “-g” only if required in the config file |
| AbcDataType | typedef: abc-data-type grouping: abc-data-type | primitive data types complex data types |
| AbcEnumeration | typedef: abc-enumeration identity: abc-enumeration | enumerations |

### Mapping Scope

UML artifacts which are annotated by the «Example» stereotype are ruled out from mapping.

## Mapping of Classes

The classes are mapped in three steps.

**Step 1:** All classes are mapped to “grouping” statements; even if they only have a single attribute identified as partOfObjectKey.

**Step 2:** One or more additional groupings are defined for every class that has at least one attribute identified as partOfObjectKey. These “*class-reference-grouping*” statements group together the key-attributes of the specific class as well as the key attributes of its ancestors (starting from its immediate parent, all the way up to the root-element). Thus, it follows that one reference grouping would be defined per schema-tree-path for the class. (see section 5.6.1. for more details)

**Step 3:** The groupings are then integrated into the YANG schema tree:

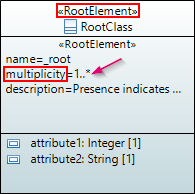
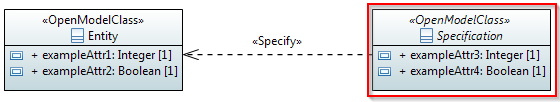
* Object classes identified as «RootElement» are incorporated as top-level Yang root elements using a “list” or “container” statement which in-turn includes the associated *class-grouping* via the “uses” sub-statement. The RootElement::multiplicity stereotype-property is used to determine whether to use a list or container.  
  RootElement::multiplicity = 1 🡪 container statement.  
  RootElement::multiplicity > 1 🡪 list statement.  
    
  See example mapping in Table 5.4.
* Object class-attributes (UML association-end-role) which are referenced via a composite aggregation association are incorporated into the *class-grouping* statement of the owner class using a “list” or “container” statement which in-turn includes the attribute *class-grouping* via the “uses” sub-statement. The association-end-role multiplicity property is used to determine whether to use a list or container. See example mapping in Table 5.15.
* Object class-attributes (UML association-end-role) which are referenced via an «ExtendedComposite» association are incorporated into the *class-grouping* statement of the owner class directly via the “uses” sub-statement. See example mapping in Table 5.15.
* Object class-attributes (UML association-end-role) which are referenced via a pointer association (incl. shared aggregation association) are incorporated into the *class-grouping* statement of the owner class using a “list” or “container” statement which in-turn includes the attribute *class-reference-grouping* via the “uses” sub-statement. The association-end-role multiplicity property is used to determine whether to use a list or container. See example mapping in Table 5.15.
* Specification classes (i.e., classes pointing via the «Specify» abstraction relationship to the entity classes) enhance their entity classes using the *augment* statement which in-turn includes the specification *class-grouping* indirectly via the “uses” sub-statement.  
  In the – not recommended – case of a concrete specification class the *augment* statement the specification *class-grouping indirectly* wrapped by a container statement.  
    
  See example in Table 5.17.

Table 5.2: Class Mapping  
(Mappings required by currently used UML artifacts)

|  |  |  |
| --- | --- | --- |
| Class 🡪 “grouping” statement | | |
| UML Artifact | YANG Artifact | Comments |
| documentation “Applied comments” (carried in XMI as “ownedComment”) | “description” substatement | Multiple “applied comments” defined in UML, need to be collapsed into a single “description” substatement. |
| superclass(es) | "grouping" statement | Concrete superclasses are then mapped to container/list which uses these groupings. |
| abstract | "grouping" statement | It is possible that the superclass or abstract class contains the key attribute for the instantiated, subclass. When the subclass is instantiated, the key value shall be identified from within the used grouping of the superclass. |
| isLeaf | ?? |  |
| object identifier  See OpenModelAttribute::partOfObjectKey in Table 5.5. |  | Attributes of the class can be marked as object identifier. |
| object identifier list  Does not appear in the UML which is used for mapped to YANG. |  | The splitting of a list attribute (marked as key) into a single key attribute and an additional list attribute will be done in UML during Pruning&Refactoring. I.e., the mapping tool will never get a list attribute which is part of the object identifier. |
| OpenInterfaceModel\_Profile:: objectCreationNotification [YES/NO/NA] (obsolete) | “notification” statement | See section 5.10 Goes beyond the simple “a notification has to be sent”; a tool can construct the signature of the notification by reading the created object. |
| OpenInterfaceModel\_Profile:: objectDeletionNotification [YES/NO/NA] (obsolete) | “notification” statement | See section 5.10 Goes beyond the simple “a notification has to be sent”; a tool can construct the signature of the notification by providing the object identifier of the deleted object (i.e., not necessary to provide the attributes of the deleted object). |
| OpenInterfaceModel\_Profile:: «RootElement» | “list“ or “container” statement  depending on OpenInterfaceModel\_Profile::RootElement::multiplicity | See guidelines above the table. |
| multiplicity >1 on association to the class | list::“min-elements” and “max-elements” substatements | min-elements default = 0 max-elements default = unbounded mandatory default = false |
| OpenModel\_Profile::«Reference» | “reference” substatement |  |
| OpenModel\_Profile::«Example» | Ignore Example elements and all composed parts |  |
| OpenModel\_Profile::lifecycleState | “status” substatement or “description” substatement | See section 5.12. |
| Proxy Class See section 6.6.  XOR See section 6.3  OpenModel\_Profile::«Choice» (obsolete) See section 6.4 | “choice” substatement |  |
| OpenModelClass::support | “if-feature” substatement | Support and condition belong together. If the “support” is conditional, then the “condition” explains the conditions under which the class has to be supported. |
| OpenModelClass::condition |
| operation | “action” substatement | YANG 1.0 supports only rpc 🡪 add prefix to the rpc name; i.e., objectClass::rpc; action requires YANG 1.1 |
| Conditional Pac | “container“ statement with “presence” substatement | See section 6.2. |
| Root container | Presence statement = “This container shall not be deleted.” |

Table 5.3: Class Mapping  
(Mappings for remaining YANG substatements)

|  |  |  |
| --- | --- | --- |
| Class 🡪 “grouping” statement 🡪 “list“ or “container” statement | | |
| UML Artifact | YANG Artifact | Comments |
|  | ~~“config” substatement~~ | not relevant to class |
| not needed now | ~~“must” substatement~~ | not relevant to class |
| not needed now | ~~list::“ordered-by” substatement~~ | not relevant to class  ordered-by default = system |
| {<constraint>} | “when” substatement |  |

Table 5.4: Class Mapping Examples

|  | | grouping super-class-1 {  leaf attribute-1 {  …  mandatory true;  }  leaf-list attribute-2 {  …  min-elements 2;  max-elements 4;  } }  grouping super-class-2 {  leaf attribute-3 {  …  mandatory true;  }  leaf-list attribute-4 {  …  min-elements 1;  } }  grouping sub-class {  leaf-list attribute-5 {  …  }  leaf attribute-6 {  …  } }  container sub-class {  …  uses sub-class  uses super-class-1;  uses super-class-2; } | |
| --- | --- | --- | --- |
| Delete example ?? | | from IETF draft-dharini-netmod-g-698-2-yang-04: module ietf-opt-if-g698-2 {  namespace "urn:ietf:params:xml:ns:yang:ietf-opt-if-  g698-2";  prefix ietf-opt-if-g698-2;  import ietf-interfaces {  prefix if;  }  …  augment "/if:interfaces/if:interface" {  description "Parameters for an optical interface";  container opt-if-o-ch-rs-ss {  description "RsSs path configuration for an   Interface";  container if-current-application-code {  description "Current Application code of the  interface";  uses opt-if-o-ch-application-code;  }  container if-supported-application-codes {  config false;  description "Supported Application codes of  the interface";  uses opt-if-o-ch-application-code-list;  }  uses opt-if-o-ch-power;  uses opt-if-o-ch-central-frequency;  } }  … | |
|  | | grouping root-class {  …  leaf attribute-1 {  type uint64;  …  }  leaf attribute-2 {  type string;  …  } }  container root {  presence "Presence indicates data-store is enabled";  uses root-class;  … } | |
|  | | grouping root-class {  …  leaf attribute-1 {  type uint64;  …  }  leaf attribute-2 {  type string;  …  } }  list root {  key attribute-1;  uses *root-class*;  …; }  Note: If RootElement::multiplicity is > 1, a description property is ignored. | |
| grouping top-level-ref {  leaf top-level-attribute-1 {  type leafref {  path 'main-model:top-level/main-model:attribute-1';  }  }  }  grouping sub-level-2-ref {  uses top-level-ref;  leaf sub-level-2-key {  type leafref {  path 'main-model:sub-level-2/main-model:key';  }  }  }  grouping sub-level-4-ref {  uses top-level-ref;  leaf sub-level-4-key-1 {  type leafref {  path 'main-model:sub-level-4/main-model:key-1';  }  }  }  grouping top-level-specification-ref {  leaf top-level-specification-attribute-2 {  type leafref {  path 'specification-model:top-level-specification/specification-model:attribute-2';  }  }  } | | module main-model {  … grouping top-level {  leaf attribute-1 {  type int64;  }  … }  grouping sub-level-1 {  … }  grouping sub-level-2 {  …  leaf key {  type string;  }  … }  grouping sub-level-3 {  … }  grouping sub-level-4 {  …  leaf key-1 {  type string;  }  … }  list root {  key attribute-1;  uses top-level;  container sub-level-1 {  uses sub-level-1;  }  list sub-level-2 {  uses sub-level-2-ref;  key ‘sub-level-2-key’;  }  container sub-level-3 {  uses sub-level-3;  }  list sub-level-4 {  uses sub-level-4-ref;  key ‘sub-level-4-key-1’;  }  … }  module specifications-model {  …  grouping top-level-specification {  leaf attribute-2 {  type string;  …  }  …  }  augment "/main-model:root" {  uses top-level-specification;  }  } | |

## Mapping of Attributes

Table 5.5: Attribute Mapping  
(Mappings required by currently used UML artifacts)

|  |  |  |  |
| --- | --- | --- | --- |
| Attribute 🡪 “leaf” (for single-valued attribute) or “leaf-list” (for multi-valued attribute) statement | | | |
| UML Artifact | YANG Artifact | | Comments |
| documentation “Applied comments” (carried in XMI as “ownedComment”) | “description” substatement | | Multiple “applied comments” defined in UML, need to be collapsed into a single “description” substatement. |
| type | “type” substatement (built-in or derived type) | |  |
| isOrdered | leaf-list::“ordered-by” substatement ("system" or "user”)  The leaf-list::"description" substatement may suggest an order to the server implementor. | | ordered-by default = system  e.g., positionSequence in OTN, layerProtocolList  If the attribute is ordered and   * read only (writeAllowed = WRITE\_NOT\_ALLOWED|CREATE\_ONLY) = system * writeable = user |
| isUnique | No unique sub-statement in leaf-list. | | Only relevant for multi-valued attributes.  YANG 1.0: The values in a leaf-list MUST be unique. YANG 1.1: In configuration data, the values in a leaf-list MUST be unique. I.e., YANG 1.1 allows non-unique values in non-configuration leaf-lists.  See example in Table 5.7 below. |
| Multiplicity (carried in XMI as lowerValue and upperValue) | leaf only: “mandatory” substatements [0..1] => no mapping needed; is leaf default [1] => mandatory substatement = true  leaf-list only: “min-elements” and “max-elements” substatements [0..x] => no mapping needed; is leaf-list default [1..x] => min-elements substatement = 1 [0..3] => max-elements substatement = 3 | | min-elements default = 0 max-elements default= unbounded mandatory default = false |
| defaultValue | "default" substatement | | If a default value exists and it is the desired value, the parameter does not have to be explicitly configured by the user. When the value of “defaultValue” is “NA”, the tool ignores it and doesn’t print out “default” substatement. |
| OpenModelAttribute:: partOfObjectKey >0 | list::“key” substatement | | It is possible that the (abstract) superclass contains the key attribute for the instantiated subclass.  Always read/write. |
| OpenModelAttribute::uniqueSet | list::“unique” substatement | See also Example in Table 5.7. | |
| OpenModelAttribute::isInvariant | “extension” substatement 🡪 ompExt:is-invariant | | See extensions YANG module in section 8.2. |
| OpenModelAttribute::valueRange | For string typed attributes: “pattern”, and/or “length” substatement of “type” substatement.  For integer and decimal typed attributes: “range” substatement of “type” substatement.  For all other typed attributes and for string or integer or decimal typed attributes where the UML definition is not compliant to YANG: “description” substatement. | | The tool should provide a warning at the output of the mapping process notifying when one or more UML valueRange definitions are contained in the description substatement of the corresponding leaf or leaf-list. When the value of “valueRange” is “null”, “NA”, “See data type”, the tool ignores it and doesn’t print out “range” substatement. |
| OpenModelAttribute::unsigned | See data type mapping in Table 5.11. Built-In Type::uintX | | Only relevant for Integer typed attributes. |
| OpenModelAttribute::counter | ietf-yang-types::counterX ietf-yang-types::gaugeX ietf-yang-types::zero-based-counterX | | Only relevant for Integer typed attributes. |
| OpenModelAttribute::unit | “units” substatement | |  |
| OpenInterfaceModelAttribute::writeAllowed |  | |  |
| CREATE\_ONLY + isInvariant = false |  | | e.g., ODUflex with HAO |
| CREATE\_ONLY + isInvariant = true |  | | e.g., fixed size ODU, identifier provided by the client |
| UPDATE\_ONLY + isInvariant = false |  | | initial value provided by the system |
| CREATE\_AND\_UPDATE + isInvariant = false |  | | unrestricted read/write |
| WRITE\_NOT\_ALLOWED + isInvariant = false | “config” substatement (false) | | config default = true  e.g., operationalState |
| WRITE\_NOT\_ALLOWED + isInvariant = true |  | | e.g., identifier provided by the system |
| OpenInterfaceModelAttribute:: attributeValueChangeNotification [YES/NO/NA] (obsolete) |  | |  |
| OpenInterfaceModelAttribute:: bitLength | See data type mapping in Table 5.11. | |  |
| OpenInterfaceModelAttribute:: encoding   * NA * BASE\_64 * HEX * OCTET | * ?? * ?? * ?? | |  |
| OpenInterfaceModel\_Profile::«Bit» | “bit” sub-statement in a “Bits” Built-In Type | |  |
| OpenModel\_Profile:: «PassedByReference» | if passedByReference = true 🡪 type leafref { path “/<object>/<object identifier>"  if passedByReference = false 🡪 either “list” statement (key property, multiple instances) or “container” statement (single instance) | | Relevant only to attributes that have a class defined as their type. |
| OpenModel\_Profile::«Reference» | “reference” substatement | |  |
| OpenModel\_Profile::«Example» | Ignore Example elements and all composed parts | |  |
| OpenModel\_Profile::lifecycleState | “status” substatement or “description” substatement | | See section 5.12. |
| OpenModelAttribute::support | For conditional support only:  “if-feature” substatement  “when” substatement if condition can be formalized as XPath expression (i.e., it is conditioned by the value of another attribute) | | Support and condition belong together. If the “support” is conditional, then the “condition” explains the conditions under which the class has to be supported. |
| OpenModelAttribute::condition |

Table 5.6: Attribute Mapping  
(Mappings for remaining YANG substatements)

|  |  |  |
| --- | --- | --- |
| Attribute 🡪 “leaf” (for single-valued attribute) or “leaf list” (multi-valued) statement | | |
| UML Artifact | YANG Artifact | Comments |
| Operation exception error notification? | “must” substatement |  |
| {<constraint>} | “when” substatement |  |

Table 5.7: Attribute Type Mapping Example

|  | grouping class-1 {  description "This class models the...";   leaf class-1-id {  type string;  mandatory true;  config false;  }   leaf attribute-1 {  type string;  mandatory true;  }   leaf-list attribute-2 {  type int64 {  range "1-100";  }  min-elements 2;  max-elements 6;  }   leaf attribute-3 {  type boolean;  default true;  config false;  ompExt:is-invariant  }   leaf attribute-4 {  type enumeration {  enum LITERAL\_1;  enum LITERAL\_2;  enum LITERAL\_3;  }  default LITERAL\_2;  config false;  }  } |
| --- | --- |
|  | Can this be defined via a single leaf argument in the unique sub-statement of the containing list statement?  I.e., grouping *unique-example* {  leaf-list unique-attribute {  type String;  min-elements 1;  } }  list unique-example {  key "unique-attribute";  unique "unique-attribute";  uses *unique-example*; }  -------  You could transform:  leaf-list numbers {  type int16;  unique true; // NOT ALLOWED IN YANG !!!! }  to:  list numbers {  key number;  leaf number {  type int16;  } }  If this is config=true data uniqueness is guaranteed for the leaf-list itself.  If this is config=false the key statement guarantees uniqueness. If you don't want uniqueness just remove the key statement.  Note: this will change the protocol encoding by adding in a new "numbers" wrapper element.  So… a recommendation could be… If you really need a non-configuration data leaf-list that is guaranteed to be ‘unique’… you need to create a list with a leaf in it and a “key”, instead of using leaf-list. |
|  | grouping unique-set-example {  leaf attribute-1 {  type string;  }  leaf attribute-2 {  type int64;  }  leaf attribute-3 {  type string;  } }  list unique-set-example {  key "attribute-1";  unique "attribute-1 attribute-2";  unique "attribute-2 attribute-3";  uses unique-set-example; } |

## Mapping of Data Types

Various kinds of data types are defined in UML:

1. Primitive Data Types (not further structured; e.g., Integer, MAC address)
2. Complex Data Types (containing attributes; e.g., Host which combines ipAddress and domainName)
3. Enumerations
4. Bit field Data Types (also known as flags)

Data Types are used as type definition of attributes and parameters.

### Generic Mapping of Primitive Data Types

Table 5.8: Primitive Data Type Mapping

|  |  |  |
| --- | --- | --- |
| Primitive Data Type 🡪 “typeDef” statement | | |
| UML Artifact | YANG Artifact | Comments |
| documentation “Applied comments” (carried in XMI as “ownedComment”) | “description” substatement | Multiple “applied comments” defined in UML, need to be collapsed into a single “description” substatement. |
| type | “type” substatement (built-in type or derived type) |  |
| defaultValue | "default" substatement | If a default value exists and it is the desired value, the parameter does not have to be explicitly configured by the user. When the value of “defaultValue” is “NA”, the tool ignores it and doesn’t print out “default” substatement. |
| unit | “units” substatement |  |
| OpenModel\_Profile::«Reference» | “reference” substatement |  |
| OpenModel\_Profile::«Example» | Ignore Example elements and all composed parts |  |
| OpenModel\_Profile::lifecycleState | “status” substatement or “description” substatement | See section 5.12. |

### Generic Mapping of Complex Data Types

Table 5.9: Complex Data Type Mapping

|  |  |  |
| --- | --- | --- |
| Complex Data Type containing only one attribute 🡪 “typedef” statement; see Table 5.8  Complex Data Type containing more than one attribute 🡪 “grouping” statement | | |
| UML Artifact | YANG Artifact | Comments |
| documentation “Applied comments” (carried in XMI as “ownedComment”) | “description” substatement | Multiple “applied comments” defined in UML, need to be collapsed into a single “description” substatement. |
| not used | “action” substatement |  |
| XOR | “choice” substatement | See section 6.3. |
| OpenModel\_Profile::«Reference» | “reference” substatement |  |
| OpenModel\_Profile::«Example» | Ignore Example elements and all composed parts |  |
| OpenModel\_Profile::«Exception» | ?? |  |
| OpenModel\_Profile::lifecycleState | “status” substatement or “description” substatement | See section 5.11. |
| OpenInterfaceModel\_Profile::«Bits» | Typedef of type “bits” | See section 5.5.5. |
| complex attribute | “uses”, “container” or “list” substatement |  |

Note: Leaf and leaf-list can only use built-in types, typedef types or enumerations in their type substatement; i.e., not groupings. Complex data types with more than one item (e.g., name value pair) can only be defined using groupings. Groupings can only be used by grouping, container and list statements.

Table 5.10: Complex Data Type Mapping Example

|  | grouping class-r  …  container attribute-current {  …  uses data-type-a;  }  list attribute-potential {  key "attribute-1 attribute-3";  …  min-elements 1;  uses data-type-a;  } }  -------------------------------------  grouping data-type-a {  …  leaf attribute-1 {  …  type string;  mandatory true;  }  leaf attribute-2 {  …  type int64;  }  leaf-list attribute-3 {  …  type string;  mandatory true;  } } |
| --- | --- |

### Mapping of Common Primitive and Complex Data Types

A list of generic UML data types is defined in a “CommonDataTypes” Model Library. This library is imported to every UML model to make these data types available for the model designer.

Table 5.11: Common Primitive and Complex Data Type Mapping

|  |  |  |
| --- | --- | --- |
| UML CommonDataTypes 🡪 YANG Built-In Types, ietf-yang-types, ietf-inet-types | | |
| UML Artifact | YANG Artifact | Comments |
| UML Primitive Types |  | The following YANG Built-In types are currently not used:   * + binary | Any binary data   + decimal64 | 64-bit signed decimal number |
| Boolean | Built-In Type::boolean |  |
| «LENGTH\_8\_BIT» Integer | Built-In Type::int8 |  |
| «LENGTH\_16\_BIT» Integer | Built-In Type::int16 |  |
| «LENGTH\_32\_BIT» Integer | Built-In Type::int32 |  |
| «LENGTH\_64\_BIT» Integer | Built-In Type::int64 |  |
| Integer | If bitLength = NA and unsigned = default (i.e., false). |
| «UNSIGNED, LENGTH\_8\_BIT» Integer | Built-In Type::uint8 |  |
| «UNSIGNED, LENGTH\_16\_BIT» Integer | Built-In Type::uint16 |  |
| «UNSIGNED, LENGTH\_32\_BIT» Integer | Built-In Type::uint32 |  |
| «UNSIGNED, LENGTH\_64\_BIT» Integer | Built-In Type::uint64 |  |
| Real  (Not used so far. See also float and double below.) | Built-In Type::decimal64 | YANG foresees a corresponding built-in type "decimal64" (RFC6020 sect. 9.3) but, for this built-in type, YANG requires mandatory to express also the accuracy with the "fraction-digit" sub-statement (RFC6020 sect. 9.3.4), which indicates the expected number of significant decimal digits. "fraction-digit" could range from 1 to 18.  Based on the value assigned to the "fraction-digit", the range of real numbers that can be expressed changes significantly. RFC6020 in sect. 9.3.4 shows the supported ranges based on the value chosen for "fraction-digit". Here things work in such a way that, the larger the range you want to express, the lower the accuracy in terms of decimal part.  It's not even so immediate to identify a conventional, "nominal" level of accuracy, since this actually depends on the specific context of application. To achieve this, we should identify a level of accuracy that we are sure suits always to all possible cases.  So, even if we have a 1:1 correspondence of built-in type between UML and YANG, an automatic conversion to provide the correct mapping couldn't be so straightforward as it appears at a first glance. |
| «LENGTH\_32\_BIT» Real (float) | typedef float {  type decimal64 {  fraction-digits 7;  } } |  |
| «LENGTH\_64\_BIT» Real (double) | typedef double {  type decimal64 {  fraction-digits 16;  } } |  |
| String | Built-In Type::string |  |
| ~~Unlimited Natural~~ |  | currently not used |
| Counter and Gauge Types |  |  |
| «COUNTER, LENGTH\_32\_BIT» Integer | ietf-yang-types::counter32 |  |
| «COUNTER, LENGTH\_64\_BIT» Integer | ietf-yang-types::counter64 |  |
| «GAUGE, LENGTH\_32\_BIT» Integer | ietf-yang-types::gauge32 |  |
| «GAUGE, LENGTH\_64\_BIT» Integer | ietf-yang-types::gauge64 |  |
| «ZERO\_COUNTER, LENGTH\_32\_BIT» Integer | ietf-yang-types::zero-based-counter32 |  |
| «ZERO\_COUNTER, LENGTH\_64\_BIT» Integer | ietf-yang-types::zero-based-counter64 |  |
| Date and Time related Types |  |  |
| DateTime | ietf-yang-types::date-and-time |  |
| ~~Timestamp~~ | ~~ietf-yang-types::timestamp~~ | Not needed |
| ~~Timeticks~~ | ~~ietf-yang-types::timeticks~~ | hundredths of seconds since an epoch, best mapped to dateTime  Not needed |
| Domain Name and URI related Types |  |  |
| DomainName | ietf-inet-types::domain-name |  |
|  | ietf-inet-types::host |  |
| Uri | ietf-inet-types::uri |  |
| Address related Types |  |  |
|  | ietf-inet-types::ip-address |  |
| Ipv4Address | ietf-inet-types::ipv4-address |  |
| Ipv6Address | ietf-inet-types::ipv6-address |  |
|  | ietf-inet-types::ip-address-no-zone |  |
| Ipv4AddressNoZone | ietf-inet-types::ipv4-address-no-zone |  |
| Ipv6AddressNoZone | ietf-inet-types::ipv6-address-no-zone |  |
| Ipv4Prefix | ietf-inet-types::ipv4-prefix |  |
| Ipv6Prefix | ietf-inet-types::ipv6-prefix |  |
|  | ietf-inet-types::ip-prefix |  |
| MacAddress | ietf-yang-types::mac-address |  |
| ~~PhysAddress~~ | ~~ietf-yang-types::phys-address~~ | Not needed |
| Protocol Field related Types |  |  |
| Dscp | ietf-inet-types::dscp |  |
|  | ietf-inet-types::ip-version |  |
| IpV6FlowLabel | ietf-inet-types::ipv6-flow-label |  |
| PortNumber | ietf-inet-types::port-number |  |
| String related Types |  |  |
| DottedQuad | ietf-yang-types::dotted-quad |  |
| «OctetEncoded» String | ?? |  |
| HexString  «HexEncoded» String | ietf-yang-types::hex-string |  |
| «Base64Encoded» String | ?? |  |
| Uuid | ietf-yang-types::uuid | To map to a language specific implementation |
| ?? | typedef duration {  type string {  pattern "P[0-9]+Y[0-9]+M[0-9]+DT[0-9]+H[0-9]+M [0-9]+(\.[0-9]+)?S";  }  } } | e.g. P0Y1347M0D |

### Mapping of Enumeration Types

In UML, the definition of enumerated data-types allows to constrain the set of accepted values for an attribute. There are two ways to map this in YANG: either using the “enumeration” built-in type or via the “identity” statement.

YANG allows to use the “enumeration” built-in type either directly in the “leaf” or “leaf-list” definition or indirectly via a separate “typedef”. Since UML supports only the indirect way via the definition of a separate Enumeration data type, the direct definition of an enumeration within a “leaf” or “leaf-list” is not recommended.

The YANG “enumeration” is a centralized definition totally included inside a single YANG module and eventually imported by the other modules. All the importing modules have access to the full set of defined values. Every variation of an enumeration shall be done in the defining module and it is globally available to all modules using it. It is not possible to have local extensions of the value set, where “local” means limited to a single YANG module, as it would be useful in case e.g. of experimental or proprietary extensions which should not affect or should be kept hidden to the rest of the modules.

The YANG “identity” is a distributed definition that can spread across several YANG modules. A YANG module could contain the definition of the base identity, representing the reference container for the allowed values, together with a first common set of values intended for global usage. Each importing module can then also locally add further values related to that identity. Every importing module can access the global sub-set of values and the additional values defined locally, but it has no access to the other local values defined inside other modules that it not imports. This means that extra identity values defined within one YANG module X are not visible to other YANG modules unless they import the module X. This allows for flexible and decoupled extensions and for accommodating additional experimental or proprietary values without impacts on the other modules, which are not even aware of the additional values.

Note: Since the literal names are mapped to identities and all identity names have to be unique within a module, all extendable enumerations have to have different literal names. Therefore, the following rules apply (see also example mapping in Table 5.13):

* Use the “UML literal name styled” enumeration name as the base identity name
* Prefix the literal identity names with the “UML literal name styled” enumeration name
* Convert the UML enumeration name according to the YANG naming convention and use this name as the name of the typedef.

YANG enumeration is in general more straightforward and shall be preferred when the UML enumeration is or can be considered highly consolidated and unlikely to be extended.

YANG identity shall be used for all the cases where the UML enumeration is not fully consolidated or cannot be consolidated, e.g. because the associated set of value is known to be open (or has to be left open) for future yet not known or not consolidated extensions.

The Enumeration property “isLeaf” is used to direct the mapping tool to perform the appropriate mapping. isLeaf = true 🡪 maps to typedef with enum statement; isLeaf = false 🡪 maps to identity/base.

Table 5.12: Enumeration Type Mapping  
(Mappings required by currently used UML artifacts)

|  |  |  |
| --- | --- | --- |
| Fixed Enumeration Type (isLeaf = true) 🡪 “typedef” with “enum” statement  Extendable Enumeration Type 🡪 “identity”/“base” pattern | | |
| UML Artifact | YANG Artifact | Comments |
| documentation “Applied comments” (carried in XMI as “ownedComment”) | “description” substatement | Multiple “applied comments” defined in UML, need to be collapsed into a single “description” substatement. |
| literal name | enum name  identity name |  |
| literal integer | “value” substatement | Example: |
| «Reference» | “reference” substatement |  |
| «Example» | Ignore Example elements and all composed parts |  |
| lifecycleState | “status” substatement or “description” substatement | See section 5.12. |

The table below shows the two approaches applied to the YANG mapping for a UML enumerated type.

Table 5.13: Enumeration Type Mapping Example

|  | **Using “enumeration” (isLeaf = true)**  *direct usage (not recommended):*  *grouping class-h {  …  leaf attribute-1 {  type enumeration {  enum LITERAL\_1;  enum LITERAL\_2;  enum LITERAL\_3;  default LITERAL\_2;  mandatory true;  }  } }*  indirect usage:  typedef enumeration-1 {  type enumeration {  enum LITERAL\_1;  enum LITERAL\_2;  enum LITERAL\_3;  } }  container class-h {  …  leaf attribute-1 {  type enumeration-1;  default LITERAL\_2;  } } |
| --- | --- |
| **Using “identity”/”base” (isLeaf = false)**  // an empty identity value is a “base identity” // i.e. it provides the reference name for a set of values  identity ENUMERATION\_1; identity ENUMERATION\_1\_LITERAL\_1 {  // the “base” statement qualifies this identity value  // as belonging to the ENUMERATION\_1 set  base ENUMERATION\_1; } identity ENUMERATION\_1\_LITERAL\_2 {  base ENUMERATION\_1; } identity ENUMERATION\_1\_LITERAL\_3 {  base ENUMERATION\_1; } …… typedef enumeration-1 {  type identityref {  // “identityref” defines the associated set  base ENUMERATION\_1;  } } ...... leaf attribute-1 {  type enumeration-1; } |

The table below shows the YANG mapping for enumerations that are modelled using inheritance.  
To be noted:

* Abstract enumerations are not defined in a typedef
* All enumerations are marked by “Is Leaf” = true
* The literal integer values must be different in all enumerations that are used in inheritance

Table 5.14: Enumeration Type Inheritance Mapping Example

|  |  |  |
| --- | --- | --- |
|  | {  typedef color-set-1 {  type enumeration {  enum black {  value 1;  }  enum white {  value 2;  }  enum red {  value 10;  }  enum orange {  value 11;  }  enum yellow {  value 12;  }  }  }   typedef color-set-2 {  type enumeration {  enum black {  value 1;  }  enum white {  value 2;  }  enum green {  value 20;  }  enum blue {  value 21;  }  enum indigo {  value 22;  }  enum violet {  value 23;  }  }  } | typedef rainbow-colors {  type enumeration {  enum black {  value 1;  }  enum white {  value 2;  }  enum red {  value 10;  }  enum orange {  value 11;  }  enum yellow {  value 12;  }  enum green {  value 20;  }  enum blue {  value 21;  }  enum indigo {  value 22;  }  enum violet {  value 23;  }  }  }   leaf color-set-1-attribute {  type color-set-1;  }   leaf color-set-2-attribute {  type color-set-2;  }   leaf-list rainbow-color-set-attribute {  type rainbow-colors;  } } |

The table below shows the YANG mapping for enumerations that are extended using the «abstraction» relationship.  
To be noted:

* Extending enumerations are defined as *abstract*
* Extended enumeration must be marked by “Is Leaf” = false

Table 5.15: Enumeration Type «abstraction» Mapping Example

|  |
| --- |
| **{**  **identity** enum-base **{**  **}**  **identity** base-literal **{**  **base** enum-base**;**  **}**  **identity** extension-literal-1 **{**  **base** enum-base**;**  **}**  **identity** extension-literal-2 **{**  **base** enum-base**;**  **}**  **leaf** my-enumeration-abstraction-example **{**  **type** **identityref** **{**  **base** enum-base**;**  **}**  **}**  **}** |

### Mapping of Bit Field Data Types

Data types which are annotated by the «Bits» Stereotype are mapped to a YANG typedef using the “bits” built-in type. Each attribute of the data type is mapped to a “bit” sub-statement with the following mapping of the “bit” sub-statements:

* Applied comment 🡪 description
* OpenModel\_Profile::OpenModelAttribute::support/condition 🡪 if-feature
* OpenInterfaceModel\_Profile::Bit::position 🡪 position
* OpenModel\_Profile::Reference::reference 🡪 reference
* OpenModel\_Profile::<Lifecycle> 🡪 status

The default setting of each bit is defined by an OCL expression in the default value of the bits typed attribute and is added to the default sub-statement of the corresponding leaf statement as an ordered space-separated list.

Table 5.14: Bit Set Encoding Mapping Example

| Related Profile properties:    Example Model: | typedef *mybits-type* {  type bits {  bit disable-nagle {  position 0;  }  bit auto-sense-speed {  position 1;  }  bit ten-mb-only {  position 2;  }  }  }  grouping *bits-using-class* {  leaf bits-typed-attribute {  type *mybits-type*;  default “auto-sense-speed”;  }  } |
| --- | --- |

## Mapping of Relationships

### Mapping of Associations

Pointers and shared aggregation associations are per default passed by reference (i.e., contain only the reference (name, identifier, address) to the referred instance(s) when being transferred across the interface). Composite aggregation and «ExtendedComposite» associations are always passed by value (i.e., contain the complete information of the instance(s) when being transferred across the interface).

This lead to the following 4 kinds of association scenarios:

1. Pointers and shared aggregations which are passed by reference
2. Composite aggregation associations which are passed by value
3. «ExtendedComposite» associations which can also be somehow treated as passed by value.
4. Shared aggregation associations which adds a lifecycle dependency to a class (in addition to an already existing composite aggregation lifecycle dependency).

Corresponding mapping examples are contained in Table 5.15.

References in YANG can be expressed by leafref. Each leafref for a reference to an object instance is defined in a reference grouping. The mapping tool needs to define a reference grouping per naming path/tree of the class for all classes that have at least one attribute identified as partOfObjectKey. Multiple leafrefs are defined by the mapping tool in a single reference grouping. The different levels of the reference hierarchy are composed by referencing (via uses) the reference groupings of the higher levels. The leafrefs in the reference groupings shall contain an absolute path expression using constraints if necessary. The reference groupings are contained in the “grouping statements for object references” section of the YANG module.   
The name of the reference grouping is composed by <class name> + “-ref” in case of a single reference and <class name> + “-ref-1”, “-ref-2”, etc. in case of multiple references.  
The name of the leafref typed leaf is composed by <class name> + “-” + <key attribute name>.

**Issue:** It is very hard to automatically map a reference that is pointing to an instance outside the containment hierarchy of the pointing instance; this kind of references will have to be mapped manually by hand.

Table 5.15: Association Mapping Examples

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | | | |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* grouping statements for object references  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  grouping *class-d-ref* {  leaf class-d-name {  type leafref {  path 'model:class-d/model:name';  require-instance false;  }  }  }  Information added by the tool  Please see also detailed description when the pointing attribute is read-write and the key is read-only. In the example above, when the leaf class-d-name of type leafref is read-write and the key attribute leaf name of class-d is read-only. | | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* grouping statements for object classes  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  grouping *container* {  …  list class-d {  uses *class-d*;  key 'class-d-name'  }  }  grouping *class-c* {  …  leaf attribute-1 {  …  }  leaf attribute-2 {  …  }  list class-d {  uses *class-d-ref*;  key ‘class-d-name’;  }  }  grouping *class-d* {  leaf name {  …  }  leaf attribute-3 {  …  }  }  Note:   * The “instantiation” of the classes is not shown in this example. | | |
|  | | | | |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* grouping statements for object references  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  grouping *class-d-ref* {  leaf class-d-name {  type leafref {  path 'model:class-d/model:name';  }  }  }  Information added by the tool | | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* grouping statements for object classes  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  grouping *class-c* { // multiplicity = [1]  …  leaf attribute-1 {  …  }  leaf attribute-2 {  …  }  container class-d {  uses *class-d*;  }  }  grouping *class-c* { // multiplicity = [\*]  …  leaf attribute-1 {  …  }  leaf attribute-2 {  …  }  list class-d {  uses *class-d*;  key ‘class-d-name’;  }  }  grouping *class-d* {  leaf name {  …  }  leaf attribute-3 {  …  }  }  Note:   * The “instantiation” of the classes is not shown in this example. | | |
| Note: Only key attributes are shown | | | | |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* grouping statements for object references  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  grouping *class-1-ref* {  leaf class-1-uuid {  type leafref {  path 'model:class-1/model:uuid';  }  }  }  grouping *class-2-ref* {  uses *class-1-ref*;  leaf class-2-identifier {  type leafref {  path 'model:class-1/model:class-2/model:uuid';  }  }  }  grouping *class-3-ref* {  uses *class-1-ref*;  leaf class-3-name {  type leafref {  path 'model:class-1/model:class-3/model:name';  }  }  }  grouping *class-4-ref-1* {  uses *class-2-ref*;  leaf class-4-identifier {  type leafref {  path 'model:class-1/model:class-2/model:class-4/model:uuid';  }  }  leaf class-4-otherId {  type leafref {  path 'model:class-4/model:otherId';  }  }  }  grouping *class-4-ref-2* {  uses *class-3-ref*;  leaf class-4-identifier {  type leafref {  path 'model:class-1/model:class-3/model:class-4/model:uuid';  }  }  leaf class-4-otherId {  type leafref {  path 'model:class-4/model:otherId';  }  }  }  Information added by the tool | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* grouping statements for object classes  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  grouping *class-5* {  container class-2 {  uses *class-2-ref*;  }  }  grouping *class-6* {  container class-4 {  uses *class-4-ref-1*;  }  }  grouping *class-7* {  list class-4 {  uses *class-4-ref-2*;  key 'class-1-uuid class-3-name class-4-uuid class-4-other-id';  }  } | | | |
|  | | | grouping class-a {  …  leaf attribute-1 {  …  }  leaf attribute-2 {  …  }  uses extending-class-1;  } }  grouping extending-class-1 {  leaf attribute 3 {  …  }  leaf attribute-4 {  …  } }  Lifecycle requirement from UML is enforced inYANG. |
| Read-write shared association to read-only key  Please see detailed explanation below. | | | ```    leaf selected {       type leafref {         path "/list-item/key"; require-instance false;       }       must 'boolean(/list-item[key=current()])';     }  ``` | | |

The following table summarizes the association mappings.

Table 5.16: Association Mapping Summary

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | UML | | |
|  |  | containment | association | inheritance |
| YANG | nesting | √ |  |  |
| grouping |  |  | √ abstract superclasses |
| augment |  |  | √ concrete superclasses |
| leafref |  | √ |  |

#### Explanation to “Read-write shared association to read-only key”

In case of an UML association of aggregation “none” or “shared” leading to a read-write attribute pointing to a read-only key attribute a specific mapping is required to generate valid yang.

A typical example for such case is, when a device offers a selection of several modes. National regulation agencies for example allow different channel plans. To avoid products for different regulation agencies, devices support several channel plans. The allowed channel plan for a certain region can be configured during first provisioning of the device. By selecting the right channel plan a field operation person configures the device.

Modeling such case in UML is quite simple. One class has an association of aggregation “shared” to another class. The attribute representing the pointer holds the selected key value of the other class (channel-plan). However, if the class with the key is read-only (the list of supported channel plans is fixed and part of the device capabilities) then there must be a special mapping to yang implemented.

The figure below shows the circumstance for such special mapping to yang.

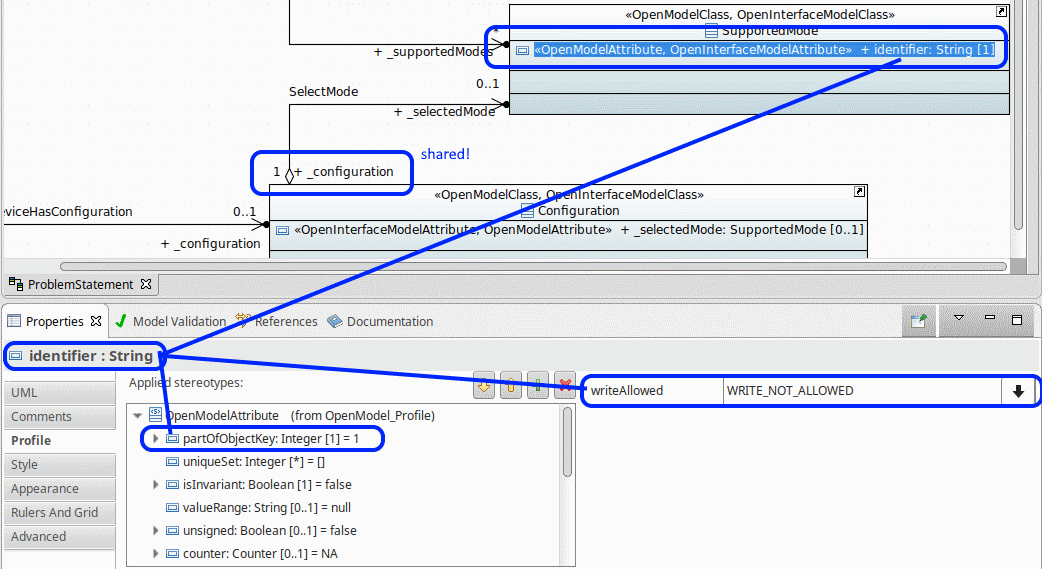


Figure 5.1: Read-write shared association to read-only key

Without a special mapping rule for the described case, the expected yang would look like as follows:

**YANG:**

[…]  
grouping configuration {  
 leaf selected-mode {  
 type leafref {  
    path "/ps:device/ps:capabilities/ps:supported-modes/ps:identifier";  
  }  
[…]

**YANG tree:**

module: problem-statement  
 +--rw device  
 +--rw identifier? string  
 +--ro capabilities  
 | +--ro supported-modes\* [identifier]  
 | +--ro identifier string  
 +--rw configuration  
 +--rw selected-mode? -> /device/capabilities/supported-modes/identifier

Unfortunately, such yang file is not valid.

$ pyang --lint problem-statement.yang

problem-statement.yang:82: **error**: the path for selected-mode is config   
but refers to a non-config leaf "identifier" defined at   
target/yang/problem-statement.yang:42   
(at target/yang/problem-statement.yang:92)

In order to avoid this error the yang 1.1 statement “require-instance” should be used with value false. The statement instructs the NetConf server not validating the existents of the corresponding key value.

The result is a compilable and valid yang module, but the missing validation by the NetConf server may lead to operational issues using the API. To improve the situation next to the “require-instance” statement a “must” statement is required, validating the existent of the key attribute value.

A valid yang module representing the UML example is:

**YANG:**

[…]  
  grouping configuration {  
    leaf selected-mode {  
      type leafref {  
        path "/ps:device/ps:capabilities/ps:supported-modes/ns:identifier";  
        require-instance false;  
      }  
      must "boolean(/ps:device/ps:capabilities/ps:supported-modes[ns:identifier=current()])";  
[…]

The content of the must statement can be calculated from the leafref path value. Please see the blue highlighted text of the example above. The path to the identifier in the example is the same in both statements – just in the must statement the identifier value must match the current value of the data object (identifier = current()).

### Mapping of Dependencies

Three different kinds of dependency scenarios need to be mapped:

1. Dependency relationship annotated by the «NamedBy» stereotype
2. Usage dependency relationship between an Interface and the object class the Interface is working on (along with the relationship name)
3. Abstraction dependency relationship annotated by the «Specify» stereotype

Table 5.17: Dependency Mapping Examples

|  | | ?? |
| --- | --- | --- |
|  | | ?? |
|  | | module main-model {  …  grouping *entity* {  leaf example-attr-1 {  type int64;  …  }  leaf example-attr-2 {  type boolean;  …  }  …  }  container xxx {  list entity {  key 'example-attr-1';  uses *entity*;  …  }  } }  module specification-model {  …  grouping specification {  leaf example-attr-3 {  type int64;  …  }  leaf example-attr-4 {  type boolean;  …  }  …  }  augment "/…/main-model:Xxx/main-model:entity"  {  uses specification;  …  } } |
| Mapping rule for «Specify» abstraction: The "target" value has the following format: [/<ModelName>:<ClassName>:<navigable association end role name>/…] This has to be mapped to: augment “/<model-name>: <navigable association end role name (without the “\_” prefix)>/ …  I.e., the red «Specify» abstraction example above is mapped to: augment "/base-model:root“  The blue «Specify» abstraction example above is mapped to: augment "/base-model:root/specify1-model:object-class-a/specify1-model:object-class-b“ | | | |
|  | | |
| module spec-1-model {  namespace "urn:onf:params:xml:ns:yang:spec-1-model";  prefix spec-1-model;  import common-model {  prefix common-model;  }   augment "/common-model:object-class-1" {  when "common-model:attribute-1='ABC'";  uses *object-class-1-spec-1*;  description "none";  }  augment "/common-model:object-class-1" {  when "common-model:attribute-1='XYZ'";  uses *object-class-1-spec-2*;  description "none";  }  grouping *object-class-1-spec-1* {  leaf attribute-a-1 {  type string;  description "none";  }  description "none";  }  grouping *object-class-1-spec-2* {  leaf attribute-a-2 {  type string;  description "none";  }  description "none";  }  …… | module common-model {  namespace "urn:onf:params:xml:ns:yang:common-model";  prefix common-model;   container object-class-1 {  uses *object-class-1*;  presence "Presence indicates ...";  description "none";  }  grouping *object-class-1* {  leaf attribute-1 {  type string;  description "none";  }  description "Stereotype <RootElement>  name: String [1] = \_objectClass1  multiplicity: String [1] = 1..1  description: String [0..1] = Presence indicates ...";  }  …… | |

## Mapping of Interfaces (grouping of operations)

Table 5.18: UML Interface Mapping

|  |  |  |
| --- | --- | --- |
| UML Interface 🡪 Submodule | | |
| UML Artifact | YANG Artifact | Comments |
| documentation “Applied comments” (carried in XMI as “ownedComment”) | “description” statement | Multiple “applied comments” defined in UML, need to be collapsed into a single “description” substatement. |
| abstract | "grouping" statement |  |
| OpenModel\_Profile::«Reference» | “reference” statement |  |
| OpenModel\_Profile::«Example» | Ignore Example elements and all composed parts |  |
| OpenModel\_Profile::lifecycleState | “status” substatement or “description” substatement | See section 5.12. |
| OpenModelInterface::support | “if-feature” substatement | Support and condition belong together. If the “support” is conditional, then the “condition” explains the conditions under which the class has to be supported. |
| OpenModelInterface::condition |

## Mapping of Operations

Table 5.19: Operation Mapping

|  |  |  |
| --- | --- | --- |
| Operation 🡪 “action” and “rpc” statements  (RFC 6020: The difference between an action and an rpc is that an action is tied to a node in the data tree, whereas an rpc is associated at the module level.) | | |
| UML Artifact | YANG Artifact | Comments |
| documentation “Applied comments” (carried in XMI as “ownedComment”) | “description” substatement | Multiple “applied comments” defined in UML, need to be collapsed into a single “description” substatement. |
| pre-condition | “extension” substatement🡪 ompExt: pre-condition | RFC 6020: During the NETCONF <edit-config> processing errors are already send for: - Delete requests for non-existent data. - Create requests for existent data. - Insert requests with "before" or "after"  parameters that do not exist. - Modification requests for nodes tagged with  "when", and the "when" condition evaluates  to "false".  See extensions YANG module in section 8.2. |
| post-condition | “extension” substatement🡪 ompExt: post-condition | See extensions YANG module in section 8.2. |
| input parameter | “input” substatement |  |
| output parameter | “output” substatement |  |
| operation exceptions  Internal Error Unable to Comply Comm Loss Invalid Input Not Implemented Duplicate Entity Not Found Object In Use Capacity Exceeded Not In Valid State Access Denied | “extension” substatement🡪 ompExt:operation-exceptions   |  |  | | --- | --- | | error-tag | error-app-tag | | operation-failed | too-many-elements too-few-elements must-violation | | data-missing | instance-required missing-choice | | bad-attribute | missing-instance | | See extensions YANG module in section 8.2. |
| OpenModelOperation:: isOperationIdempotent (obsolete) | “extension” substatement🡪 ompExt:is-operation-idempotent | See extensions YANG module in section 8.2. |
| OpenModelOperation:: isAtomic (obsolete) | “extension” substatement🡪 ompExt:is-atomic | See extensions YANG module in section 8.2 |
| OpenModel\_Profile::«Reference» | “reference” substatement |  |
| OpenModel\_Profile::«Example» | Ignore Example elements and all composed parts |  |
| OpenModel\_Profile::lifecycleState | “status” substatement or “description” substatement | See section 5.12. |
| OpenModelOperation::support | “if-feature” substatement | Support and condition belong together. If the “support” is conditional, then the “condition” explains the conditions under which the class has to be supported. |
| OpenModelOperation::condition |

Table 5.20: Interface/Operation Mapping Example

|  | module open-model-uml-yang-mapping-guidelines {  <yang-version statement>  <namespace statement>  prefix "mapg";  <import statements>  include “interface-a” {  …;  }  <organization statement>  <contact statement>  <description statement>  <reference statement>  … }  submodule interface-a {  <yang-version statement>  belongs-to " open-model-uml-yang-mapping-guidelines" {  prefix "mapg";  }  <import statements>  <organization statement>  <contact statement>  <description statement>  <reference statement>  <revision statements>  …  rpc operation-1 {  …  }  rpc operation-2 {  …  } } |
| --- | --- |

Table 5.21: Operation Exception Mapping Example

|  | ?? |
| --- | --- |

## Mapping of Operation Parameters

Table 5.22: Parameter Mapping

|  |  |  |
| --- | --- | --- |
| Operation Parameters 🡪 “input” substatement or “output” substatement | | |
| UML Artifact | YANG Artifact | Comments |
| documentation “Applied comments” (carried in XMI as “ownedComment”) | “description” substatement | Multiple “applied comments” defined in UML, need to be collapsed into a single “description” substatement. |
| direction | “input” / “output” substatement |  |
| type | see mapping of attribute types (grouping, leaf, leaf-list, container, list, typedef, uses) |  |
| isOrdered |  |
| multiplicity |  |
| defaultValue |  |
| OpenModelParameter::valueRange |  |
| OpenInterfaceModel\_Profile:: passedByReference | if passedByReference = true 🡪 type leafref { path “/<object>/<object identifier>"  if passedByReference = false 🡪 either “list” statement (key property, multiple instances) or “container” statement (single instance) | Relevant only to parameters that have a class defined as their type. |
| OpenModel\_Profile::«Reference» | “reference” substatement of the individual parameters (container, leaf, leaf-list, list, uses) |  |
| OpenModel\_Profile::«Example» | Ignore Example elements and all composed parts |  |
| OpenModel\_Profile::lifecycleState | “status” substatement or “description” substatement  of the individual parameters (container, leaf, leaf-list, list, uses) | See section 5.12. |
| OpenModelParameter::support | “if-feature” substatement of the individual parameters (container, leaf, leaf-list, list, uses) | Support and condition belong together. If the “support” is conditional, then the “condition” explains the conditions under which the class has to be supported. |
| OpenModelParameter::condition |
| XOR: See section 6.3 | “choice” substatement |  |
| error notification? | “must” substatement |  |
| complex parameter | “uses” substatement |  |

Table 5.23: Interface/Operation/Parameter Mapping Example

|  | submodule interface-a {  …  rpc operation-1 {  …  input {  leaf parameter-1 {  type string;  mandatory true;  }  leaf parameter-2 {  type boolean;  mandatory true;  }  }  output {  leaf parameter-2 {  type boolean;  mandatory true;  }  leaf-list parameter-3 {  type int64;  min-elements 3; }  }  }  rpc operation2 {  …  output {  leaf-list parameter-4 {  type string;  }  }  } } |
| --- | --- |

## Mapping of Notifications

Like the class mapping, the signals are also mapped in two steps. In the first step, all signals are mapped to “grouping” statements. In the second step the groupings of all non-abstract signals are “instantiated” in “notification” statements.

Table 5.24: Notification Mapping

|  |  |  |
| --- | --- | --- |
| Signal 🡪 “grouping” statement 🡪 “notification“ statement | | |
| UML Artifact | YANG Artifact | Comments |
| documentation “Applied comments” (carried in XMI as “ownedComment”) | “description” substatement | Multiple “applied comments” defined in UML, need to be collapsed into a single “description” substatement. |
| OpenModel\_Profile::«Reference» | “reference” substatement |  |
| OpenModel\_Profile::«Example» | Ignore Example elements and all composed parts |  |
| OpenModel\_Profile::lifecycleState | “status” substatement or “description” substatement | See section 5.12. |
| OpenModelNotification:: triggerConditionList | Not mapped |  |
| OpenModelNotification::support | “if-feature” substatement | Support and condition belong together. If the “support” is conditional, then the “condition” explains the conditions under which the class has to be supported. |
| OpenModelNotification::condition |
| Proxy Class: See section 6.6.  XOR: See section 6.3. | “choice” substatement |  |
| error notification? | “must” substatement |  |
| attributes | see mapping of attribute types (grouping, leaf, leaf-list, container, list, typedef, uses) |  |
| complex attribute | “uses” substatement |  |

Table 5.25: Notification Mapping Example

|  | grouping *generic-notification* {  …  leaf generic-attribute-1 {  …  mandatory true;  }  leaf-list generic-attribute-2 {  …  mandatory true;  } }  grouping notification-a {  …  leaf attribute-1 {  type string;  …  }  leaf attribute-2 {  type integer;  …  } }  notification notification-a {  …  uses generic-notification;  uses notification-a; }  notification notification-a {  …  uses generic-notification;  uses notification-a; }  Table from [onf2015.276](https://login.opennetworking.org/bin/c5i?mid=4&rid=7&gid=0&k1=1498&k2=1&k3=9&tid=1449652081):   | **Parameter name** | **ITU-T M.3702** | **3GPP TS32.302** | | --- | --- | --- | | objectClass | M | M | | objectInstance | M | M | | notificationId aka notificationIdentifier | M | M | | eventTime | M | M | | systemDN | M | M | | notificationType | M | M | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |

## Mapping of UML Packages

The mapping tool shall generate a YANG module per UML model.

According to the UML Modeling Guidelines [7], each UML model is basically structured into the following packages:

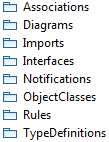


Figure 5.1: Pre-defined Packages in a UML Module

The grouping that is provided through these packages shall persist in the YANG module using headings defined as comments.

Table 5.26: UML Package to YANG Heading Mapping

| **UML Package** | **YANG Heading** |
| --- | --- |
|  | grouping statements for object references  augment statements |
|  | identity statements  typedef statements  grouping statements for complex data types |
|  | grouping statements for object-classes |
|  | rpc statements |
|  | notification statements |

## Mapping of Lifecycle

Table 5.27: Lifecycle Mapping

|  |  |  |
| --- | --- | --- |
| UML Lifecycle | | |
| UML Artifact | YANG Artifact | Comments |
| <Lifecycle Stereotypes> | “status“ substatement or “description” substatement | «UML» 🡪 “YANG”  «Deprecated» 🡪 "deprecated" «Experimental» (default) 🡪 description «Faulty» 🡪 description «LikelyToChange» 🡪 description «Mature» 🡪 "current" (default) «Obsolete» 🡪 "obsolete" «Preliminary» 🡪 description  Allow having a switch per state in the mapping tool to map it or not; default is Mature only. See also section 7.2. |

## Mapping Issues

### YANG 1.0 or YANG 1.1?

YANG 1.0 is approved and defined in RFC 6020 [1].  
YANG 1.1 is approved and defined in RFC 7950 [11]. The enhancements are listed in section “Summary of Changes from RFC 6020”.

### Combination of different Associations?

Table 5.28: Combination of Associations Mapping Examples

|  | container class-n {  …  container class-p {  …  leaf-list class-o {  type leafref {  path “/class-o/name";  }  }  leaf class-q {  type leafref {  path “/class-q/name";  }  }  }  list class-o {  …  container class-q {  …  }  } } |
| --- | --- |

# Mapping Patterns

## UML Recursion

As YANG defines hierarchical data store, any instances that need to store recursive containment will require translation. A mapping between object-oriented store and a hierarchical store is possible; however, there is more than one option: e.g.,

* Reference based approach - have a flat list of objects, where the objects are linked into a hierarchy using references. An example of a two-way navigable approach is in RFC 7223.
* Assume some specific number of “recursions”; i.e., specify some default number of recursion levels, and define a configurable parameter to allow changing the number of levels.

Text to be inserted discussing the pros and cons of these options, and rational for selecting the referenced based approach.

### Reference Based Approach

Table 6.1: Recursion Mapping Examples

|  | list object {  key name;  leaf name {  type string;  }  leaf-list object-within-object {  type leafref {  path “/object/name";  }  } } |
| --- | --- |
|  | Example from IETF RFC 7223 (https://datatracker.ietf.org/doc/rfc7223/)  +--rw interfaces | +--rw interface\* [name] | +--rw name string | +--rw description? string | +--rw type identityref | +--rw enabled? Boolean | +--rw link-up-down-trap-enable? enumeration +--ro interfaces-state  +--ro interface\* [name]  +--ro name string  +-- ...  +--ro higher-layer-if\* interface-state-ref  +--ro lower-layer-if\* interface-state-ref  +-- ...  where  typedef interface-state-ref {  type leafref {  path "/if:interfaces-state/if:interface/if:name";  }  description  "This type is used by data models that need to  reference the operationally present interfaces.";  }  leaf-list higher-layer-if {  type interface-state-ref;  description  "A list of references to interfaces layered on top  of this interface.";  reference  "RFC 2863: The Interfaces Group MIB -   fStackTable";  }  leaf-list lower-layer-if {  type interface-state-ref;  description  "A list of references to interfaces layered   underneath this interface.";  reference  "RFC 2863: The Interfaces Group MIB -   ifStackTable";  } |

## UML Conditional Pacs

UML conditional Pacs are abstract classes used to group attributes which are associated to the containing class under certain conditions. The abstract “attribute containers” are mapped to container statements. The condition is mapped to the “presence” property of the container statement.

Note: An example of this usage is given in the “Data nodes for the operational state of IP on interfaces." within ietf-ip.yang (RFC 7277).

Table 6.2: Mapping of Conditional Packages

|  |  |
| --- | --- |
|  | grouping class-e {  …  leaf object-identifier {  type string;  }  leaf attribute-2 {  …  }  grouping class-f-pac {  …  leaf attribute-3 {  …  }  leaf attribute-4 {  …  } }  grouping class-g-pac {  …  leaf attribute-5 {  …  }  leaf attribute-6 {  …  } }  list class-e {  key "object-identifier";  uses class-e;  …  container class-f-pac {  presence " <condition for ClassF\_Pac attributes>";  uses class-f-pac;  …  }  container class-g-pac {  presence " <condition for ClassG\_Pac attributes>";  uses class-g-pac;  …  } } |

## {xor} Constraint

Associations related by the {xor} constraint are mapped to the “choice” statement.

Table 6.3: {xor} Constraint Mapping Examples

|  | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* grouping statements for object classes  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  grouping *substitute* {  leaf attribute-1 {  type string;  }  choice alternative-choice {  case alt-1 {  list alternative-1 {  key 'name';  uses *alternative-1*;  }  }  case alt-2 {  container alternative-2 {  uses *alternative-2*;  }  }  case alt-3 {  container alternative-3 {  uses *alternative-3*;  }  }  }  }  grouping *alternative-1* {  leaf name {  type string;  }  }  grouping *alternative-2* {  leaf name {  type string;  }  }  grouping *alternative-3* {  leaf name {  type string;  }  } |
| --- | --- |
|  | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* grouping statements for complex data types  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  grouping *probable-cause* {  choice probable-cause-type-choice {  case integer-type {  container integer-probable-cause {  uses *integer-probable-cause*;  }  }  case string-type {  container string-probable-cause {  uses *string-probable-cause*;  }  }  }  }  grouping *integer-probable-cause* {  }  grouping *string-probable-cause* {  } |
|  | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* grouping statements for complex data types  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  grouping *parent-1* {  }  grouping *parent-2* {  }  grouping *parent-3* {  }  grouping *child* {  choice parent-choice {  case parent-1 {  container parent-1 {  uses *parent-1*;  }  }  case parent-2 {  container parent-2 {  uses *parent-2*;  }  }  case parent-3 {  container parent-3 {  uses *parent-3*;  }  }  }  } |
|  | |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* grouping statements for complex data types  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  grouping *choice-1-data-type* {  choice choice-1-choice {  default case-2;  case case-1 {  uses *case-1-data-type*;  }  case case-2 {  uses *case-2-data-type*;  }  }  }  grouping *choice-2-data-type* {  choice choice-2-choice {  default case-3;  case case-3 {  uses *case-3-data-type*;  }  case case-4 {  uses *case-4-data-type*;  }  }  }  grouping *case-1-data-type* {  leaf attribute-1 {  type string;  }  }  grouping *case-2-data-type* {  leaf attribute-2 {  type string;  }  choice choice-3-choice {  case case-5 {  uses *case-5-data-type*;  }  case case-6 {  uses *case-6-data-type*;  }  }  } | grouping *case-3-data-type* {  leaf attribute-3 {  type string;  }  }  grouping *case-4-data-type* {  leaf attribute-4 {  type string;  }  }  grouping *case-5-data-type* {  leaf attribute-5 {  type string;  }  }  grouping case-6-data-type {  leaf attribute-6 {  type string;  }  }  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* grouping statements for object classes  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  grouping choice-using-class {  container choice-1 {  uses choice-1-data-type;  }  container choice-2 {  uses choice-2-data-type;  }  } |

## «Choice» Stereotype (obsolete)

The «Choice» stereotype can be associated in UML to a class or a data type. The class or a data type which is annotated with the «Choice» stereotype represents one of a set of classes/data types. This pattern is mapped to the “choice” property of the container/list/grouping statement.

Table 6.4: «Choice» Stereotype Mapping Examples

|  | grouping substitute-object-class {  …  }  grouping alternative-1-object-class {  … }  grouping alternative-2-object-class {  … }  grouping alternative-3-object-class {  … }  container alternative-1-object-class {  uses alternative-1-object-class;  … }  container alternative-2-object-class {  uses alternative-2-object-class;  … }  container alternative-3-object-class {  uses alternative-3-object-class;  … }  list substitute-object-class {  key …;  …  choice \_alternative {  case alternative-1-object-class {  leaf alternative-1-object-class {  type leafref {  path '/alternative-1-object-class’;  }  }  }  case alternative-2-object-class {  leaf alternative-2-object-class {  type leafref {  path '/alternative-2-object-class’;  }  }  }  case alternative-3-object-class {  leaf alternative-3-object-class {  type leafref {  path '/alternative-3-object-class;  }  }  }  } } |
| --- | --- |
|  | grouping integer-probable-cause {  …  leaf probable-cause {  type int64;  …  } }  grouping string-probable-cause {  …  leaf probable-cause {  type string;  …  } }  grouping probable-cause {  …  choice probable-cause {  case integer-probable-cause {  container integer-probable-cause {  uses integer-probable-cause;  …  }  }  case string-probable-cause {  container string-probable-cause {  uses string-probable-cause;  …  }  }  } } |

## Mapping of UML Support and Condition Properties

The UML Modeling Guidelines [7] define support and condition properties for the UML artifacts class, attribute, signal, interface, operation and parameter. The support property can be defined as one of M – Mandatory, O – Optional, C – Conditional, CM – Conditional-Mandatory, CO – Conditional-Optional. It qualifies the support of the artifact at the management interface. The condition property contains the condition for the condition-related support qualifiers (C, CM, CO).

M – Mandatory maps to the “mandatory” substatement. O – Optional need not be mapped since the default value of the “mandatory” substament is “false”; i.e., optional.

For the conditional UML support qualifiers, the first line of the condition text is mapped to a “feature” statement. The mapping tool needs to scan – in a first step – all conditions and create “feature” statements for each **different** first line of all conditions. The second and further lines of the condition text are mapped to the “description” substatement of the “feature” statement. I.e., all condition strings which have the same first line must also have the same condition text. It is possible to add the same condition text to more than one artifact. In a second step, the tool adds an “if-feature” substatement with a reference to the corresponding “feature” to all mapped UML conditional artefacts.

Table 6.5: Support and Condition Mapping Examples

|  | feature a-b-c {  description  "If ABC is supported by the system.";  }  container conditional-class {  …  if-feature “a-b-c”;  leaf attribute-1 {  type string;  …  }  leaf attribute-2 {  type int64;  …  } } |
| --- | --- |
|  | feature x-y-z {  description  "If XYZ is supported by …";  }  container conditional-attribute-class {  …  leaf attribute-3 {  type string;  …  }  leaf attribute-4 {  type int64;  if-feature “x-y-z”;  …  }  leaf attribute-5 {  type string;  if-feature “not x-y-z”;  …  } } |

## Proxy Class Association Mapping

UML allows an association to an abstract proxy class. This abstract proxy class is acting as a placeholder for all related (via inheritance or composition) classes. The mapping tool has to map this single association into relationships to all classes which are related to the proxy class.



Figure 6.1: Example: Proxy Class Mapping

## Building YANG Tree

The YANG data schema is tree structured. The tool analyses the UML composite aggregation associations and creates the YANG tree(s).

UML classes which are not component of any other class (via a composition relationship) are mapped to YANG tree roots. The YANG trees are created below the roots following the “lines” of composite aggregation associations in UML.

Table 6.6: Composite Aggregation Associations Mapping to YANG Tree Example

|  |  |  |
| --- | --- | --- |
|  | | |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* Groupings  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  grouping class-s {  leaf attribute1 {  … }  }  grouping class-t {  leaf attribute2 {  … }  }  grouping class-u {  leaf attribute3 {  … }  }  grouping class-v {  leaf attribute4 {  … }  }  grouping class-w {  leaf attribute5 {  … }  }  grouping class-x {  leaf attribute6 {  … }  }  grouping class-y {  leaf attribute7 {  … }  } | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* Main Containers  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  list class-s {  key attribute1;  uses class-s;  …  list class-u {  key attribute3;  uses class-u;  …  list class-v {  key attribute4;  uses class-v;  uses class-x  …  }  }  }  list class-s {  key attribute1;  uses class-s;  …  list class-u {  key attribute3;  uses class-u;  …  list class-w {  key attribute5;  uses class-w;  …  list class-y {  key attribute7;  uses class-y;  …  }  }  }  } | list class-t {  key attribute2;  uses class-t;  …  list class-u {  key attribute3;  uses class-u;  …  list class-v {  key attribute4;  uses class-v;  uses class-x  …  }  }  }  list class-t {  key attribute2;  uses class-t;  …  list class-u {  key attribute3;  uses class-u;  …  list class-w {  key attribute5;  uses class-w;  …  list class-y {  key attribute7;  uses class-y;  …  }  }  }  } |

# 

# Generic UML Model and specific YANG Configuration Information

The UML model provides already a lot of generic information which is also mapped to YANG. If necessary, the user can overwrite some of the generic UML model information before it is mapped to YANG and can provide a few mapping instructions to the tool in a config file.

## YANG Module Header

RFC 6087bis [2] Appendix C defines a YANG Module Template which require information that is not contained in the UML model; see also section 10. The tool needs to ask the user for the following information and insert it into the YANG header:

|  |  |  |
| --- | --- | --- |
| Input | | Output |
| UML | Config file | YANG Module Header Information |
| UML model name |  | module name  // same as the UML model name; transformed into the YANG naming scheme; e.g., “UmlYangSimpleTestModel.uml” is mapped to “uml-yang-simple-test-model.yang” or “G.1234.5\_v0.03-model.uml” is mapped to “g-dot1234dot5-v-dot0dot03-model.yang”. |
| OpenModelStatement::revision:Revision::date <yyyy>-<mm>-<dd>  See revision::date below | "revision"::"date":"<yyyy>-<mm>-<dd>"  See revision::date below | file name  // consists of the module name and the date: "<module-name>@<yyyy>-<mm>-<dd>.yang"; |
| OpenModelStatement::namespace urn:<sdo>:<project> | "namespace":"urn:<sdo>:<project>:yang:",  e.g., urn:onf:otcc:{tapi|wt|wr|…}:yang:<module name> e.g., urn:itu:t:rec:g.8052.1:yang:{ITU-T-YangModuleIdentifier} | namespace "urn:<sdo>:<project>:yang:<module name>",  // string with a unique namespace URN value |
|  | "prefix":{  "<UML model name1>":"<prefix1>”,  "<UML model name2>":"<prefix2>”,  "<UML model name3>":"<prefix3>”,  "<UML model name4>":"<prefix4>”,  …  }, | prefix  // try to pick a unique prefix; should not have more than 8-10 characters |
| OpenModelStatement::organization | "organization":"<SDO and project/wg name>",(human friendly written) e.g., ONF OTCC (Open Transport Configuration & Control) Project", | organization  // identify also the working group if applicable |
| OpenModelStatement::contact:Contact [1]   * projectWeb:String [1] = <<https://.../project-name/>> * projectEmail:String [1] = <<mailto:project-name@...>> * editorName:String [0..1] = <project editor> * editorEmail:String [0..1] = <mailto:project-editor@example.com> | "contact":"\n Project Web: <<https://.../project-name/>>\n Project Email: <<mailto:project-name@...>>\n Editor: <project editor>\n <<mailto:project-editor@example.com>>”, | contact “  Project Web: <https://.../project-name/>  Project Email: <mailto:project-name@...>  Editor: <project editor> <mailto:project-editor@example.com>" |
| OpenModelStatement::description [0..1] “This model defines < brief description of the model; 1 line>”  OpenModelStatement::copyright [1]  OpenModelStatement::licence [1] |  | description  “This module defines <brief description of the module; 1 line>  <copyright notice>  <license statement>”  //Blue text hard coded in the mapping tool |
|  | "reference":“<clearly identify the source (e.g., UML model)>” | reference  “<clearly identify the source (e.g., UML model)>” |
| OpenModelStatement::revision:Revision [1..\*]   * date [1] <yyyy>-<mm>-<dd> * version [1] <project/project version> * description [0..1]  {<additional specific description>} * changelog [0..1] <link to a github UML change log> * additionalChanges [0..1] {<additional manual changes>} * reference [0..1] <list of referenced documents> | "revision": [{  "date":"<yyyy>-<mm>-<dd>",  "description":”<project/project version>\n {<additional specific description>}\n <link to the github yang change log> {<additional manual changes>}",  "reference":"< list of referenced documents>"  }] | revision “<revisiondate; <yyyy>-<mm>-<dd>>” {  revision::description  description “<project/project version> This YANG module is automatically generated by the xmi2yang Mapping Tool, version <version of the tool>. {<additional specific description>.} Changes in this revision: <link to the github yang change log> {<additional manual changes>}";  revision::reference “<list of referenced documents>”;  }  // what changed in this revision  //Blue text hard coded in the mapping tool |
| Applied profiles:  OpenModelProfile, v<version>  OpenInterfaceModelProfile, v<version>  ProfileLifecycleProfile, v<version> |  |  |
|  | "withSuffix":true|false, | Add suffix “-g” to the groupings: No|Yes (default: No) |

## Lifecycle State Treatment

UML elements are annotated by at least one of the following lifecycle states (see also section 5.12):

* «Deprecated»
* «Experimental»
* «Faulty»
* «LikelyToChange»
* «Mature»
* «Obsolete»
* «Preliminary».

The tool shall allow the user to select – based on the lifecycle states – which UML elements are mapped; default is Mature only.

# Mapping Basics

## UML 🡪 YANG or XMI 🡪 YANG

Figure 8.1: Example UML to YANG Mapping



Figure 8.2: Example XMI (Papyrus) to YANG Mapping

## Open Model Profile YANG Extensions

The additional UML artifact properties defined in the Open Model Profile are mapped as YANG extension statements.

<CODE BEGINS> file "iisomi-OpenModelProfileExtensions@2017-04-12.yang"

// Contents of "OpenModelProfileExtensions"

module OpenModelProfileExtensions {

namespace "urn:IISOMI:OpenModelProfileExtensions";

prefix "ompExt";

organization

"IISOMI (Informal Inter-SDO Open Model Initiative)";

description

"This module defines the Open Model Profile extensions for

usage in other YANG modules.";

revision 2017-04-12 {

description "ONF replaced by IISOMI";

reference "IISOMI 514 UML Modeling Guidelines";

}

revision 2015-07-28 {

description "Initial revision";

}

// extension statements

extension is-invariant {

description

"Used with attribute definitions to indicate that the value

of the attribute cannot be changed after it has been created.";

}

extension pre-condition {

description

"Used with operation definitions to indicate the conditions

that have to be true before the operation can be started

(i.e., if not true, the operation will not be started at all

and a general “precondition not met” error will be returned,

i.e., exception is raised).";

argument "condition-list";

}

extension post-condition {

description

"Used with operation definitions to indicate the state of

the system after the operation has been executed (if

successful, or if not successful, or if partially successful).

Note that partially successful post-condition(s) can only

be defined in case of non-atomic operations.

Note that when an exception is raised, it should not be

assumed that the post-condition(s) are satisfied.";

argument "condition-list";

}

extension operation-exceptions {

description

"Used with operation definitions to indicate the allowed

exceptions for the operation.

The model uses predefined exceptions which are split in

2 types:

- generic exceptions which are associated to all operations

by default

- common exceptions which needs to be explicitly associated

to the operation.

Note: These exceptions are only relevant for a protocol

neutral information model. Further exceptions may be

necessary for a protocol specific information model.

Generic exceptions:

• Internal Error: The server has an internal error.

• Unable to Comply: The server cannot perform the operation.

Use Cases may identify specific conditions that will result

in this exception.

• Comm Loss: The server is unable to communicate with an

underlying system or resource, and such communication is

required to complete the operation.

• Invalid Input: The operation contains an input parameter

that is syntactically incorrect or identifies an object

of the wrong type or is out of range (as defined in the

model or because of server limitation).

• Not Implemented: The entire operation is not supported

by the server or the operation with the specified input

parameters is not supported.

• Access Denied: The client does not have access rights

to request the given operation.

Common exceptions:

• Entity Not Found: Is thrown to indicate that at least

one of the specified entities does not exist.

• Object In Use: The object identified in the operation

is currently in use.

• Capacity Exceeded: The operation will result in resources

being created or activated beyond the capacity supported

by the server.

• Not In Valid State: The state of the specified object is

such that the server cannot perform the operation. In

other words, the environment or the application is not in

an appropriate state for the requested operation.

• Duplicate: Is thrown if an entity cannot be created because

an object with the same identifier/name already exists.";

argument "exception-list";

}

extension is-operation-idempotent {

description

"Used with operation definitions to indicate that the operation

is idempotent.";

}

}

<CODE ENDS>

# Reverse Mapping From YANG to UML

Given the many YANG drafts that have been created, in some cases it might be helpful to revert the mapping (i.e., from YANG to UML; re-engineer) so that comparison/analysis/augmentation can be made.

Note: Since UML to YANG is not a 1:1 mapping, a tool supported reverse mapping of YANG to UML maybe different from origin UML.

# Requirements for the YANG Module Structure

This definition is following the YANG Module Template in Appendix C of RFC 6087bis [2].

<CODE BEGINS> file "<module name>@<yyyy>-<mm>-<dd>.yang"

module <module name> {

// The module header is constructed according to section 7.1

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* augment statements

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

…

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* extension statements

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

…

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* feature statements

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

…

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* identity statements

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

…

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* typedef statements

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

…

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* grouping statements for complex data types

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

…

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* grouping statements for object references

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

…

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* grouping statements for object-classes

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

…

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* data definition statements

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

…

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* rpc statements

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

…

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* notification statements

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

…

}

<CODE ENDS>

# Appendix: UML to YANG Tool User Guide

Note: This tool user guide uses the mapping of the ONF TAPI model as an example.



# Main Changes between Releases

## Summary of main changes between version 1.0 and 1.1

* Adapted to ETSI drafting rules.
* Usage of isLeaf attribute property clarified in section 5.5.4.
* Usage of isLeaf class property added in section 5.3.
* New section 5.6.2 on Mapping of Dependencies added.
* «Specify» abstraction mapping example added in Table 5.4 and Table 5.17.
* Naming Conventions Mapping in Table 5.1 enhanced and naming updated throughout the document.
* Section 5.2 on Generic Mapping Guidelines added.
* Mapping example for RootElement added to Table 5.4.
* New property attribute writeAllowed added to Table 5.5.
* Pointer association mapping examples added in Table 5.15.
* New section 5.2.3 on YANG Workarounds added.
* Literal name style mapping updated in Table 5.1.
* Conditional Specify/Augment mapping example added in Table 5.17.
* Mapping of bit encoding for Enumerations added in Table 5.13.
* «LifecycleAggregate» Relationship Mapping Examples added in Table 5.15.
* Mapping of references added in section 5.6.1 and Table 5.15.
* Mapping of Choice stereotype obsolete.
* Mapping OpenInterfaceModelClass stereotype obsolete.
* Mapping OpenModelOperation::isOperationIdempotent property obsolete.
* Mapping OpenModelOperation::isAtomic property obsolete.
* Mapping OpenInterfaceModelAttribute::attributeValueChangeNotification property obsolete.
* Composite association mapped to list using grouping instead of reference in Table 5.15.  
  “require-instance” substatement = false added to all navigable pointer and shared aggregation associations.
* {xor}-choice mapping updated in section 6.3.
* Bits mapping updated in section 5.5.5.

# Proposed Addendum 2

Stereotype:Do not generate DS?

Using spanning tree algorithm?

Depth first search (DFS) & Breadth First Search (BFS)?