ECSS-E-TM-40-07 Volume 2A
25 January 2011

Space engineering

Simulation modelling platform - Volume 2: Metamodel

ECSS Secretariat
ESA-ESTEC
Requirements & Standards Division
Noordwijk, The Netherlands

Foreword

This document is one of the series of ECSS Technical Memoranda. Its Technical Memorandum status indicates that it is a non-normative document providing useful information to the space systems developers' community on a specific subject. It is made available to record and present non-normative data, which are not relevant for a Standard or a Handbook. Note that these data are non-normative even if expressed in the language normally used for requirements.

Therefore, a Technical Memorandum is not considered by ECSS as suitable for direct use in Invitation To Tender (ITT) or business agreements for space systems development.

Disclaimer

ECSS does not provide any warranty whatsoever, whether expressed, implied, or statutory, including, but not limited to, any warranty of merchantability or fitness for a particular purpose or any warranty that the contents of the item are error-free. In no respect shall ECSS incur any liability for any damages, including, but not limited to, direct, indirect, special, or consequential damages arising out of, resulting from, or in any way connected to the use of this document, whether or not based upon warranty, business agreement, tort, or otherwise; whether or not injury was sustained by persons or property or otherwise; and whether or not loss was sustained from, or arose out of, the results of, the item, or any services that may be provided by ECSS.

Published by: ESA Requirements and Standards Division
ESTEC, P.O. Box 299,
2200 AG Noordwijk
The Netherlands
Copyright: 2011© by the European Space Agency for the members of ECSS

Change log

ECSS-E-TM-40-07 Volume 2A 25 January 2011	First issue

Table of contents

Change log ... **3**

Introduction ... **12**

1 Scope ... **14**

2 Metamodel ... **16**

2.1 Overview ... 16
2.1.1 Catalogue ... 17
2.1.2 Assembly ... 17
2.1.3 Schedule ... 17
2.1.4 Package ... 17
2.1.5 Configuration ... 17
2.2 Schemas ... 18
2.2.1 Dependencies ... 18
2.3 XML Links ... 19
2.3.1 Simple Link ... 20
2.4 Primitive Types ... 20
2.4.1 Integer Types ... 20
2.4.2 Floating-point Types ... 21
2.4.3 Character and String Types ... 21
2.4.4 Other Primitive Types ... 22

3 Core Elements ... **23**

3.1 Basic Types ... 23
3.1.1 Identifier ... 24
3.1.2 Figure 3-Name ... 24
3.1.3 Description ... 24
3.1.4 UUID ... 25
3.2 Elements ... 25
3.2.1 Named Element ... 25
3.2.2 Document ... 26
3.3 Metadata ... 26

3.3.1 Metadata 27
3.3.2 Comment 27
3.3.3 Documentation 28

4 Core Types 29

4.1 Types 29
4.1.1 Visibility Element 29
4.1.2 Visibility Kind 30
4.1.3 Type 30
4.1.4 Language Type 31
4.1.5 Value Type 31
4.1.6 Value Reference 31
4.1.7 Native Type 32
4.1.8 Platform Mapping 32
4.2 Value Types 33
4.2.1 Simple Type 34
4.2.2 Primitive Type 35
4.2.3 Enumeration 35
4.2.4 Enumeration Literal 36
4.2.5 Integer 36
4.2.6 Float 37
4.2.7 String 37
4.2.8 Array 38
4.2.9 Structure 38
4.2.10 Exception 39
4.2.11 Class 40
4.3 Features 40
4.3.1 Constant 41
4.3.2 Field 41
4.3.3 Property 42
4.3.4 Access Kind 43
4.3.5 Field versus Property 43
4.3.6 Association 44
4.3.7 Operation 45
4.3.8 Parameter 45
4.3.9 Parameter Direction Kind 46
4.4 Values 46
4.4.1 Value 47

4.4.2 Simple Value 47
4.4.3 Simple Array Value 47
4.4.4 Array Value 48
4.4.5 Structure Value 48
4.4.6 Simple Values 49
4.4.7 Simple Array Values 50
4.5 Attributes 53
4.5.1 Attribute Type 53
4.5.2 Attribute 53

5 Smdl Catalogue 55
5.1 Catalogue 55
5.1.1 Catalogue 56
5.1.2 Namespace 57
5.2 Reference Types 57
5.2.1 Reference Type 58
5.2.2 Component 59
5.2.3 Interface 60
5.2.4 Model 61
5.2.5 Service 62
5.3 Modelling Mechanisms 63
5.3.1 Class based Design 63
5.3.2 Component based Design 63
5.3.3 Event based Design 65
5.4 Catalogue Attributes 66
5.4.1 Fallible 67
5.4.2 Forcible 67
5.4.3 Minimum 68
5.4.4 Maximum 68
5.4.5 View 69
5.4.6 View Kind 69

6 Smdl Assembly 71
6.1 Assembly 71
6.1.1 Assembly Node 72
6.1.2 Instance Node 73
6.1.3 Service Proxy 74
6.1.4 Assembly 74
6.1.5 Model Instance 75

6.1.6 Assembly Instance 76
6.2 Links 76
6.2.1 Link 77
6.2.2 Interface Link 77
6.2.3 Event Link 78
6.2.4 Field Link 79

7 Smdl Schedule 81
7.1 Schedule 81
7.1.1 Schedule 81
7.2 Tasks 82
7.2.1 Task 82
7.2.2 Activity 83
7.2.3 Trigger 83
7.2.4 Transfer 84
7.2.5 Sub Task 84
7.3 Events 84
7.3.1 Event 84
7.3.2 Time Kind 85
7.3.3 Simulation Event 86
7.3.4 Epoch Event 86
7.3.5 Mission Event 87
7.3.6 Zulu Event 87
7.4 Schedule Attributes 87
7.4.1 Initialisation 87
7.4.2 Field Update 88
7.4.3 Field Update Kind 89

8 Smdl Package 90
8.1 Package 90
8.1.1 Implementation 90
8.1.2 Package 91

9 Smdl Configuration 92
9.1 Component Configuration 92
9.2 Configuration 93
9.3 Configuration Usage 94

10 Validation Rules 95
10.1 EuroSim Validation Rules 95

10.2 SIMSAT Validation Rules 97
10.2.1 General Validation Rules 97
10.2.2 Catalogue Validation Rules 97
10.2.3 Assembly Validation Rules 99
10.2.4 Schedule Validation Rules 99
10.2.5 Package Validation Rules 100

Annex A XML Schema Documents 101
A.1 Core Elements 101
A.2 Core Types 106
A.3 Smdl Catalogue 135
A.4 Smdl Assembly 146
A.5 Smdl Schedule 155
A.6 Smdl Package 163
A.7 Smdl Configuration 166

Figures

Figure 2-1: Overview 16
Figure 2-2: Schemas 18
Figure 2-3: Dependencies 19
Figure 2-4: Simple Link 20
Figure 2-5: Integer Types 21
Figure 2-6: Floating-point Types 21
Figure 2-7: Character and String Types 22
Figure 2-8: Other Primitive Types 22
Figure 3-1: Elements 23
Figure 3-2: Basic Types 24
Figure 3-3: Identifier 24
Figure 3-4: Name 24
Figure 3-5: Description 25
Figure 3-6: UUID 25
Figure 3-7: Elements 25
Figure 3-8: Named Element 26
Figure 3-9: Document 26
Figure 3-10: Metadata 27
Figure 3-11: Comment 27
Figure 3-12: Documentation 28
Figure 4-1: Types 29

Figure 4-2: Visibility Element 30
Figure 4-3: Type 30
Figure 4-4: Language Type 31
Figure 4-5: Value Type 31
Figure 4-6: Value Reference 32
Figure 4-7: Native Type 32
Figure 4-8: Platform Mapping 33
Figure 4-9: Value Types 34
Figure 4-10: Simple Type 35
Figure 4-11: Primitive Type 35
Figure 4-12: Enumeration 36
Figure 4-13: Enumeration Literal 36
Figure 4-14: Integer 37
Figure 4-15: Float 37
Figure 4-16: String 38
Figure 4-17: Array 38
Figure 4-18: Structure 39
Figure 4-19: Exception 39
Figure 4-20: Class 40
Figure 4-21: Features 41
Figure 4-22: Constant 41
Figure 4-23: Field 42
Figure 4-24: Property 43
Figure 4-25: Field versus Property 44
Figure 4-26: Association 45
Figure 4-27: Operation 45
Figure 4-28: Parameter 46
Figure 4-29: Values 47
Figure 4-30: Value 47
Figure 4-31: Simple Value 47
Figure 4-32: Simple Array Value 48
Figure 4-33: Array Value 48
Figure 4-34: Structure Value 48
Figure 4-35: Simple Values 49
Figure 4-36: Simple Array Values 51
Figure 4-37: Attribute Type 53
Figure 4-38: Attribute 54

Figure 5-1: Catalogue ... 56
Figure 5-2: Catalogue ... 57
Figure 5-3: Namespace ... 57
Figure 5-4: Reference Types ... 58
Figure 5-5: Reference Type ... 59
Figure 5-6: Component ... 60
Figure 5-7: Interface ... 61
Figure 5-8: Model ... 62
Figure 5-9: Service ... 63
Figure 5-10: Entry Point ... 63
Figure 5-11: Container ... 64
Figure 5-12: Reference ... 64
Figure 5-13: Event based Design ... 65
Figure 5-14: Event Type ... 65
Figure 5-15: Event Source ... 66
Figure 5-16: Event Sink ... 66
Figure 5-17: Fallible ... 67
Figure 5-18: Forcible ... 67
Figure 5-19: Minimum ... 68
Figure 5-20: Maximum ... 68
Figure 5-21: View ... 69
Figure 5-22: View Kind ... 69
Figure 6-1: Assembly ... 72
Figure 6-2: Assembly Node ... 73
Figure 6-3: Instance Node ... 74
Figure 6-4: Service Proxy ... 74
Figure 6-5: Assembly ... 75
Figure 6-6: Model Instance ... 75
Figure 6-7: Assembly Instance ... 76
Figure 6-8: Links ... 77
Figure 6-9: Link ... 77
Figure 6-10: Interface Link ... 78
Figure 6-11: Event Link ... 79
Figure 6-12: Field Link ... 80
Figure 7-1: Schedule ... 81
Figure 7-2: Schedule ... 82
Figure 7-3: Task ... 82

Figure 7-4: Activity 83
Figure 7-5: Trigger 83
Figure 7-6: Transfer 84
Figure 7-7: Sub Task 84
Figure 7-8: Event 85
Figure 7-9: Time Kind 85
Figure 7-10: Simulation Event 86
Figure 7-11: Epoch Event 86
Figure 7-12: Mission Event 87
Figure 7-13: Zulu Event 87
Figure 7-14: Initialisation 88
Figure 7-15: Field Update 88
Figure 7-16: Field Update Kind 89
Figure 8-1: Package 90
Figure 8-2: Implementation 91
Figure 8-3: Package 91
Figure 9-1: Configuration 92
Figure 9-2: Component Configuration 93
Figure 9-3: Configuration 93
Figure 9-4: Configuration Usage 94

Tables

Table 4-1: Enumeration Literals of VisibilityKind 30
Table 4-2: Enumeration Literals of AccessKind 43
Table 4-3: Enumeration Literals of ParameterDirectionKind 46
Table 5-1: Enumeration Literals of ViewKind 70
Table 7-1: Enumeration Literals of TimeKind 86
Table 7-2: Enumeration Literals of FieldUpdateKind 89
Table 10-1 General Validation rules 97
Table 10-2: Catalogue Validation Rules 97
Table 10-3: Assembly Validation rules 99
Table 10-4: Schedule Validation Rules 99
Table 10-5: Package Validation Rules 100

Introduction

Space programmes have developed simulation software for a number of years, and which are used for a variety of applications including analysis, engineering operations preparation and training. Typically different departments perform developments of these simulators, running on several different platforms and using different computer languages. A variety of subcontractors are involved in these projects and as a result a wide range of simulation models are often developed. This Technical Memorandum addresses the issues related to portability and reuse of simulation models. It builds on the work performed by ESA in the development of the Simulator Portability Standards SMP1 and SMP2.

This Technical Memorandum is complementary to ECSS-E-ST-40 because it provides the additional requirements which are specific to the development of simulation software. The formulation of this Technical Memorandum takes into account the Simulation Model Portability specification version 1.2. This Technical Memorandum has been prepared by the ECSS-E-40-07 Working Group.

This Technical Memorandum comprises of a number of volumes.

The intended readership of Volume 1 of this Technical Memorandum are the simulator software customer and all suppliers.

The intended readership of Volume 2, 3 and 4 of this Technical Memorandum is the Infrastructure Supplier.

The intended readership of Volume 5 of this Technical Memorandum is the simulator developer.

Note: Volume 1 contains the list of terms and abbreviations used in this document

- **Volume 1 – Principles and requirements**

  This document describes the Simulation Modelling Platform (**SMP**) and the special principles applicable to simulation software. It provides an interpretation of the ECSS-E-ST-40 requirements for simulation software, with additional specific provisions.

- **Volume 2 - Metamodel**

  This document describes the Simulation Model Definition Language (**SMDL**), which provides platform independent mechanisms to design models (Catalogue), integrate model instances (Assembly), and schedule them (Schedule). SMDL supports design and integration techniques for class-based, interface-based, component-based, event-based modelling and dataflow-based modelling.

- **Volume 3 - Component Model**

  This document provides a platform independent definition of the components used within an SMP simulation, where components include models and services, but also the simulator itself. A set of mandatory interfaces that every model has to implement is defined by the document, and a number of optional interfaces for advanced component mechanisms are specified.

  Additionally, this document includes a chapter on Simulation Services. Services are components that the models can use to interact with a Simulation Environment. SMP defines interfaces for mandatory services that every SMP compliant simulation environment must provide.

- **Volume 4 - C++ Mapping**

  This document provides a mapping of the platform independent models (Metamodel, Component Model and Simulation Services) to the ANSI/ISO C++ target platform. Further platform mappings are foreseen for the future.

  The intended readership of this document is the simulator software customer and supplier. The software simulator customer is in charge of producing the project Invitation to Tender (ITT) with the Statement of Work (SOW) of the simulator software. The customer identifies the simulation needs, in terms of policy, lifecycle and programmatic and technical requirements. It may also provide initial models as inputs for the modelling activities. The supplier can take one or more of the following roles:

  — Infrastructure Supplier - is responsible for the development of generic infrastructure or for the adaptation of an infrastructure to the specific needs of a project. In the context of a space programme, the involvement of Infrastructure Supplier team(s) may not be required if all required simulators are based on full re-use of exiting infrastructure(s), or where the infrastructure has open interfaces allowing adaptations to be made by the Simulator Integrator.

  — Model Supplier - is responsible for the development of project specific models or for the adaptation of generic models to the specific needs of a project or project phase.

  — Simulator Integrator – has the function of integrating the models into a simulation infrastructure in order to provide a full system simulation with the appropriate services for the user (e.g. system engineer) and interfaces to other systems.

- **Volume 5 – SMP usage**

  This document provides a user-oriented description of the general concepts behind the SMP documents Volume 1 to 4, and provides instructions for the accomplishment of the main tasks involved in model and simulator development using SMP.

1 Scope

The Platform Independent Model (**PIM**) of the Simulation Modelling Platform (**SMP**) consists of two parts:

1. The SMP Metamodel, also called Simulation Model Definition Language (**SMDL**).
2. The SMP Component Model, which includes SMP Simulation Services.

This document details the Simulation Model Definition Language of the SMP Platform Independent Model. As SMDL is not a model itself, but a language to describe models, it is called a **Metamodel**.

Using the platform independent Unified Modelling Language (**UML**), the mechanisms available to define models (Smdl Catalogue), to integrate models (Smdl Assembly), to schedule models (Smdl Schedule), to package model implementations (Smdl Package) and to configure models (Smdl Configuration) are defined in this Metamodel.

This document aims at providing an unambiguous definition of the Simulation Model Definition Language. This language consists of five top-level elements, namely an Smdl Catalogue that defines SMP models, an Smdl Assembly that defines how a collection of model instances is assembled, an Smdl Schedule that defines how an assembly shall be scheduled, an Smdl Package that defines how the implementations of models are packaged, and an Smdl Configuration that allows to configure field values of models from file. Each of these top-level elements is defined as an XML file format.

The purpose of this document is to enable tool developers to implement tools working with files of any of the above mentioned file formats. Such tools may include:

- **Editors** to edit files of these types, e.g. Catalogue Editor, Assembly Editor, or Schedule Editor.
- **Validators** to validate files, e.g. Catalogue Validator, Assembly Validator, or Schedule Validator.
- **Code Generators** that generate platform specific code from the platform independent description in a Catalogue or Package.
- **Document Generators** that generate documentation for models in a Catalogue.
- **Converters** e.g. to convert a model Catalogue into UML (XMI), or vice versa.

As all file formats defined in this document are XML file formats, an XML Schema is provided for each file type. However, XML Schema is not a very visual way of presenting a model, while class diagrams using the Unified Modelling Language are much more appropriate. Therefore, the main content of this document uses UML class diagrams to introduce the language elements, while the XML Schema documents are presented in Annex A

2 Metamodel

2.1 Overview

The Simulation Model Definition Language is the language used to describe SMP compliant types (including models), assemblies, schedules, packages and configurations. While the SMP Component Model defines mechanisms for the interaction of SMP Models and other SMP Components, this language forms the basis for the collaboration of SMP compliant tools.

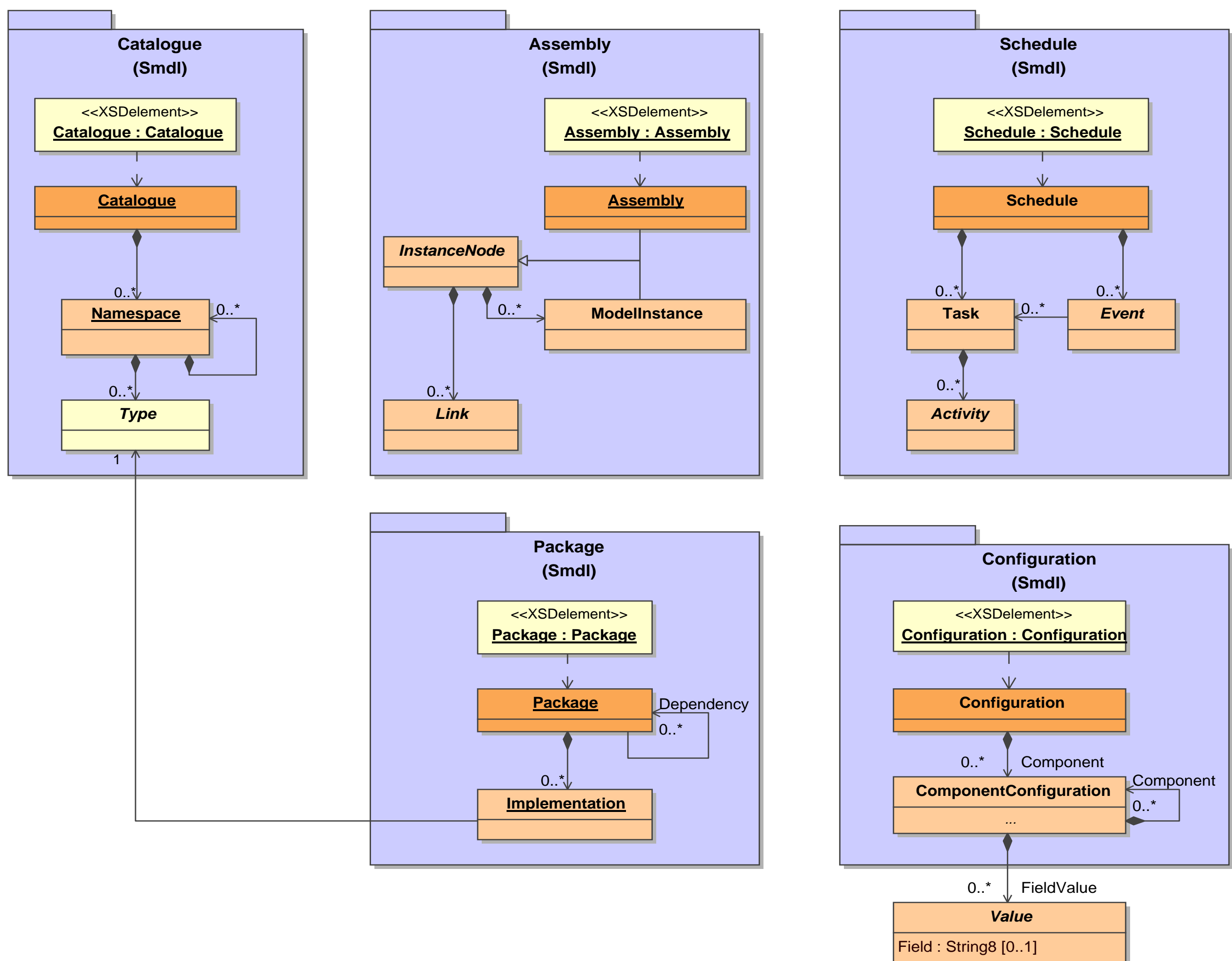

Figure 2-1: Overview

The language is specified in UML and implemented as a set of XML schemas, which are auto-generated from the UML model. Therefore, SMDL model descriptions held in XML documents can be syntactically validated against the auto-generated schemas. This mechanism ensures that SMDL modelling information can be safely exchanged between different stakeholders.

The Metamodel consists of three major and two minor parts. The major parts are the catalogue, the assembly and the schedule. The catalogue holds type information, both for value types and for models, while the assembly describes how model instances are interconnected in a specific set-up, based on the type information held in the catalogue. Finally, the schedule holds additional scheduling information for an assembly. The minor parts are a package defines grouping of type implementations, and a configuration allows specifying field values independently of an assembly.

2.1.1 Catalogue

A catalogue contains namespaces as a primary ordering mechanism. Namespaces may be nested to form a namespace hierarchy, and they typically contain types. Types include value types like integer and string, as well as reference types like interface and model. Further, event types and attribute types can be defined. For a detailed description of the catalogue, see sections 4 and 5.

2.1.2 Assembly

An assembly contains model instances, which are interconnected using links according to the rules and constraints defined in the corresponding types held in one or more catalogues. For a detailed description of the assembly, see section 6.

2.1.3 Schedule

A schedule defines how the model instances and field links of an assembly are to be scheduled. It includes timed events and cyclic events, and tasks triggered by these events. For a detailed description of the schedule, see section 7.

2.1.4 Package

A package defines which types shall be implemented into a binary package, e.g. a library or archive. While for most types only a single implementation exists, it is possible to add more than one implementation for the same model, which can be used for versioning purposes. For a detailed description of the package, see section 8.

2.1.5 Configuration

A configuration document allows specifying arbitrary field values of component instances in the simulation hierarchy. It can be used to initialise or

reinitialise dedicated field values of components in a simulation hierarchy. For a detailed description of the configuration, see section 9.

2.2 Schemas

The Metamodel is subdivided into several packages. UML components decorated with the <<XSDschema>> stereotype are mapped to separate XML schema files. The UML tagged values targetNamespace and targetNamespacePrefix thereby provide additional information about the XML namespace.

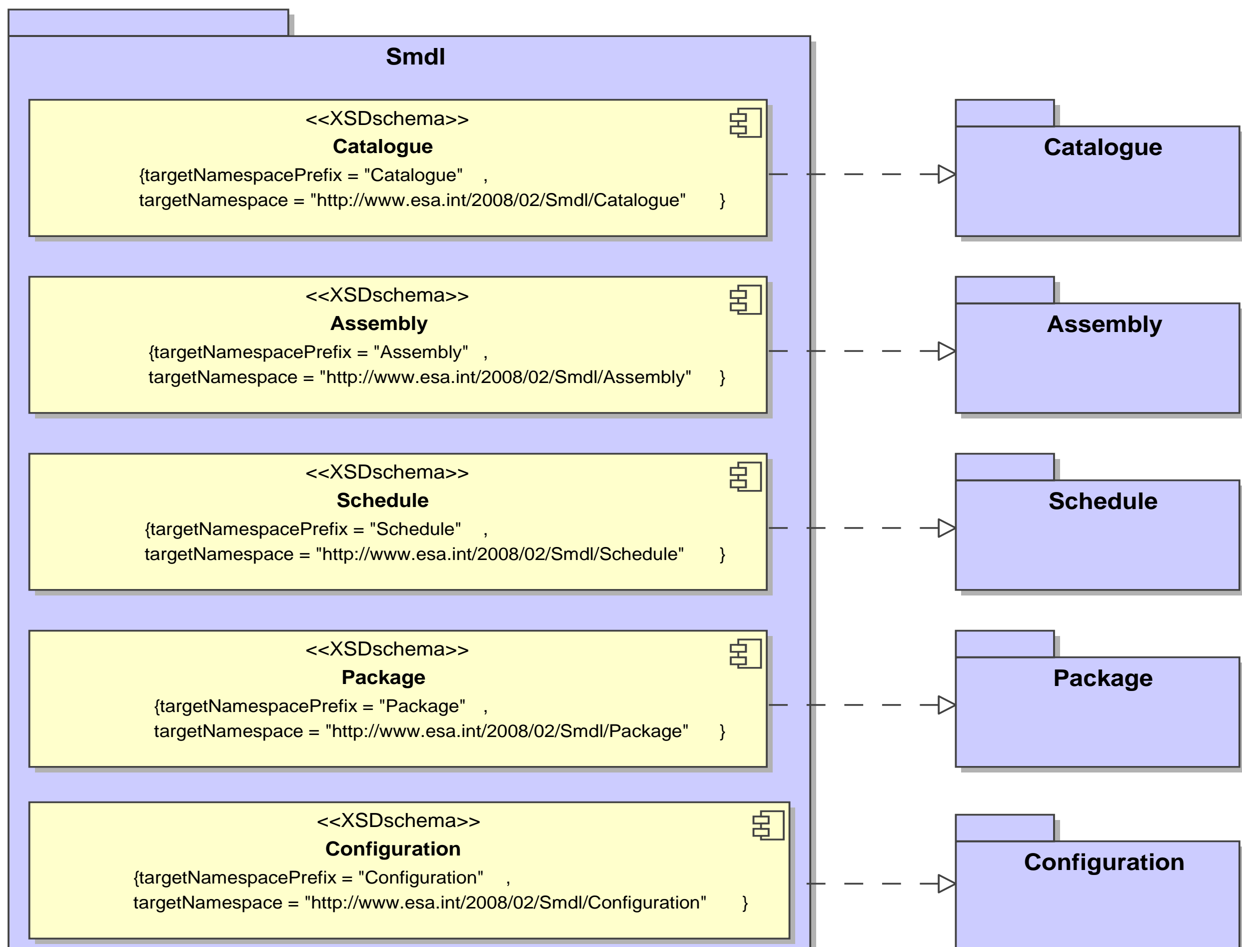

Figure 2-2: Schemas

2.2.1 Dependencies

A number of dependencies between the defined XML Schema documents exist, which translate into import statements within the schema documents. It is important to note that the Configuration schema has no dependency to Catalogue nor to Assembly.

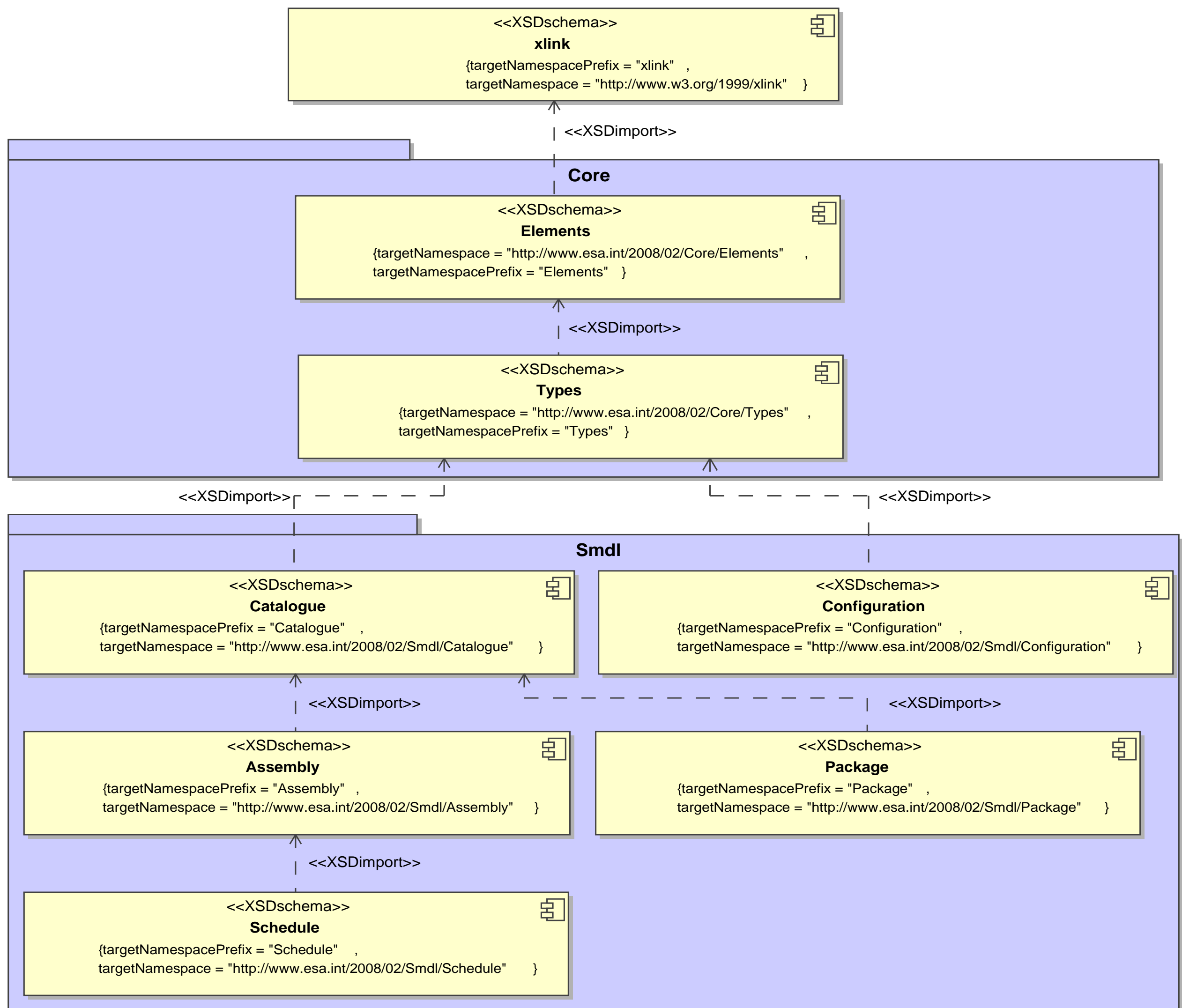

Figure 2-3: Dependencies

2.3 XML Links

The XML Linking Language (see http://www.w3.org/TR/xlink/) standardises mechanisms for creating links between elements in XML documents. This includes internal links (i.e. within a document) as well as external links, relations, and link bases. For the SMDL, limited use is made of the XML Linking Language.

Currently, only the Documentation metaclass makes use of simple links to reference external resources. Furthermore, simple links are used within the SMDL schemas as referencing mechanism.

As there is no standardised XML schema for the XML Linking Language, SMDL provides an xlink schema, which holds the definitions of the standard xlink attributes and the SimpleLink metaclass.

2.3.1 Simple Link

A SimpleLink is an XML Link of a fixed type ("simple") which references any Uniform Resource Identifier (U

RI) using its href attribute. Further, it may additionally identify the link target by name using its title attribute.

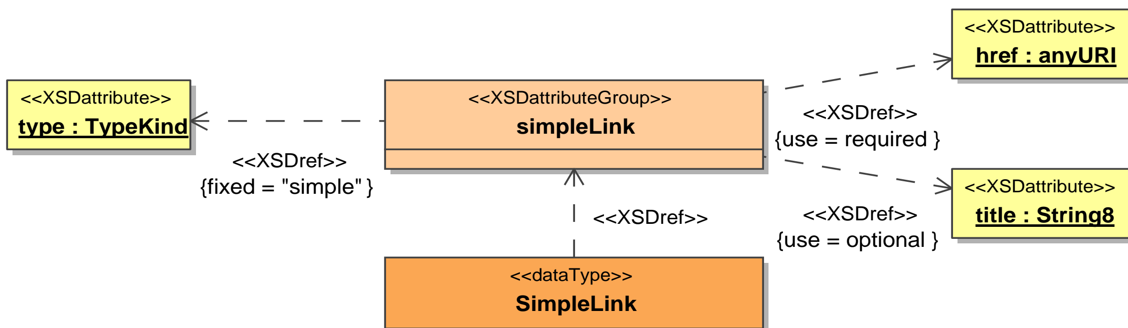

Figure 2-4: Simple Link

Remark: An XML simple link is the equivalent of the href element in the Hypertext Mark-up Language (HTML).

2.4 Primitive Types

The Metamodel is based on a set of primitive types, which have well-defined mappings into XML Schema (and other target platforms, including CORBA IDL and ANSI/ISO C++).

2.4.1 Integer Types

SMDL supports integer types in four different sizes, both signed and unsigned. In order to be unambiguous about the actual size and range of an integer, each type has the size (in bits) in the name. Unsigned integer types start with a "U". The range of the integer types is defined as follows:

- The range of signed integer types is [-2**(size-1), 2**(size-1) - 1], where "size" denotes the number of bits.
- The range of unsigned integer types is [0, 2**size - 1], where "size" denotes the number of bits.

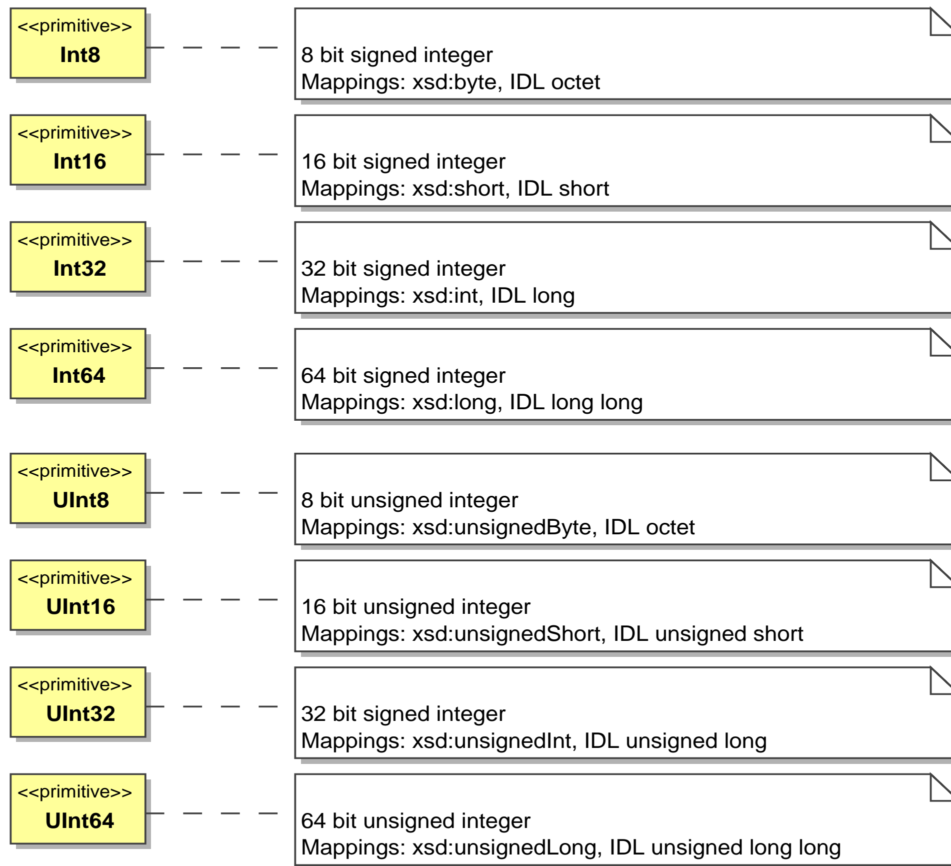

Figure 2-5: Integer Types

2.4.2 Floating-point Types

SMDL supports two floating-point types.

- Float32 is an IEEE 754 single-precision floating-point type with a length of 32 bits.
- Float64 is an IEEE 754 double-precision floating-point type with a length of 64 bits.

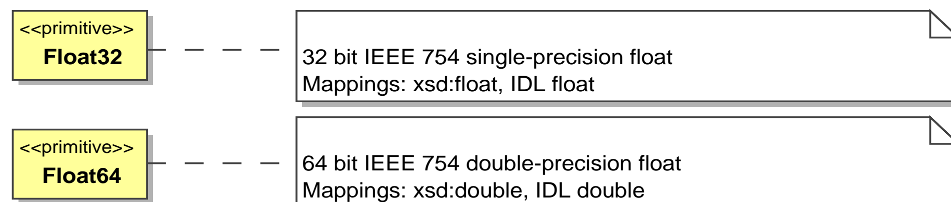

Figure 2-6: Floating-point Types

2.4.3 Character and String Types

SMDL supports an 8-bit character type in order to represent textual characters.

Furthermore, SMDL also supports 8-bit character strings based on UTF-8 encoding, which is commonly used in XML.

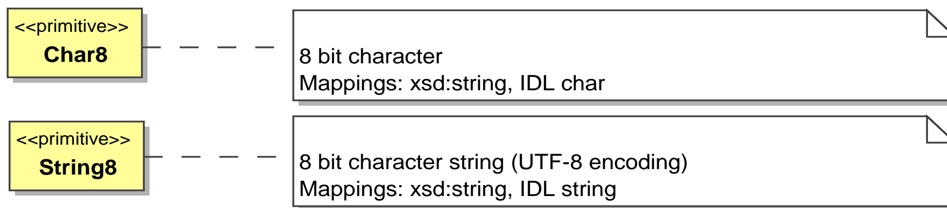

Figure 2-7: Character and String Types

2.4.4 Other Primitive Types

Apart from the above integer, floating-point, and character and string types, SMDL additionally supports the following primitive types:

- DateTime is a point in time specified as nanoseconds relative to Modified Julian Date (MJD) 2000+0.5 (1st January 2000, 12.00) that is represented by a signed 64-bit integer.
- Duration is a duration of time in nanoseconds that is represented by a signed 64-bit integer.
- Bool is a binary logical type with values true or false.

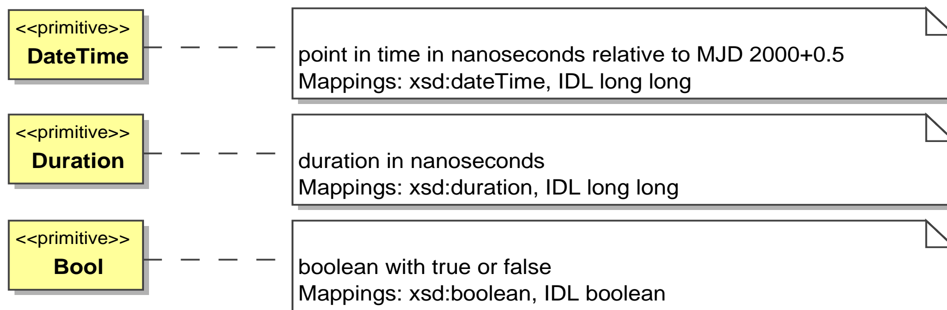

Figure 2-8: Other Primitive Types

3
Core Elements

This package defines base metaclasses and annotation mechanisms used throughout the SMP Metamodel.

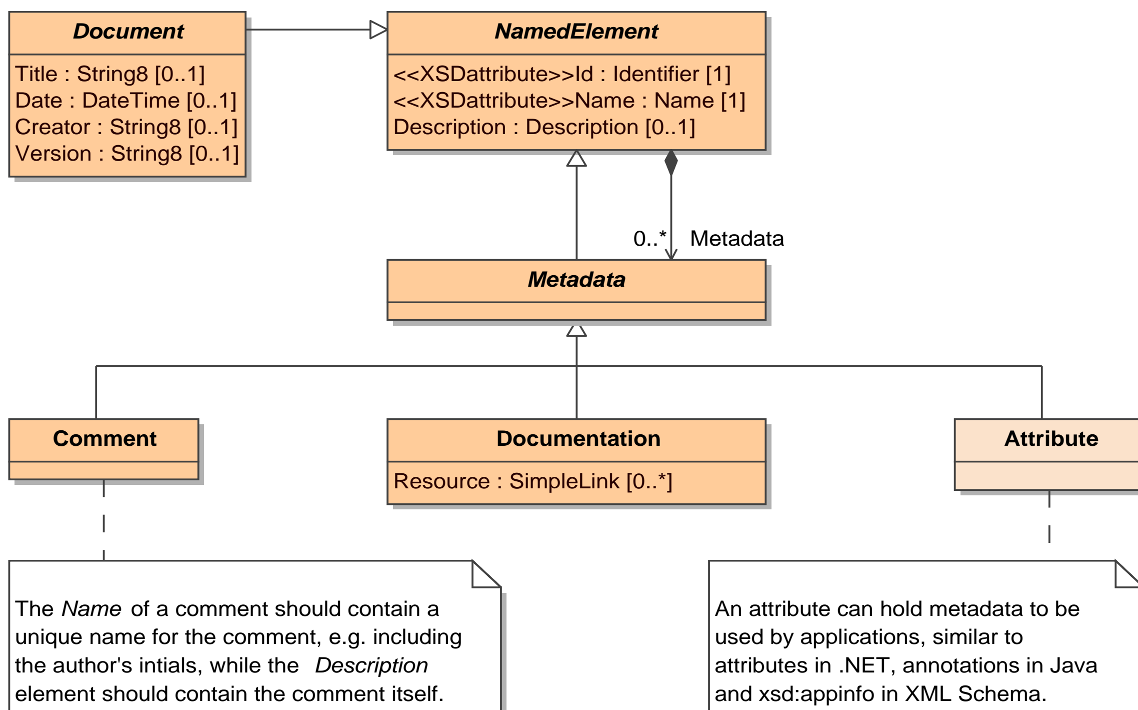

Figure 3-1: Elements

3.1 Basic Types

The Core Elements schema defines some basic types which are used for attributes of other types. Some string types use a pattern tag to specify a regular expression that limits their value space.

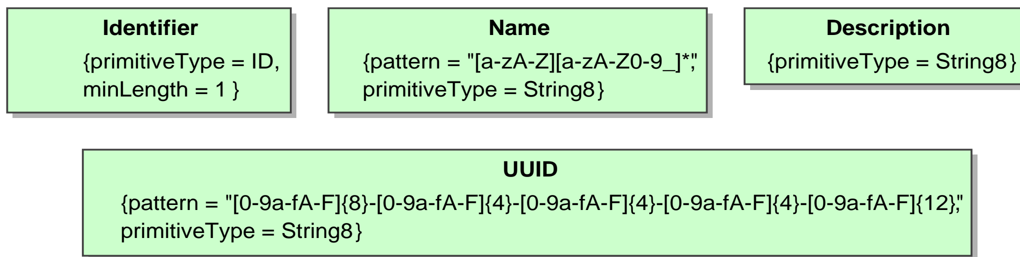

Figure 3-2: Basic Types

3.1.1 Identifier

An Identifier is a machine-readable identification string for model elements stored in XML documents, being a possible target for XML links. This type is used in the Id attribute of the NamedElement metaclass (see below).

An identifier must not be empty, which is indicated by the minLength tag.

Identifier is defined as an XML ID type. It must be unique in the space of identified things. Identifier must be used as an attribute because it is of the xsd:ID type, and this must exist only in attributes, never elements, to retain compatibility with XML 1.0 DTD's.

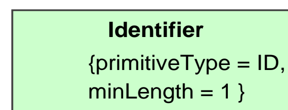

Figure 3-3: Identifier

3.1.2 Figure 3-Name

A Name is a user-readable name for model elements. This type is used in the Name attribute of the NamedElement metaclass (see below). A name must start with a character, and is limited to characters, digits, and the underscore ('_').

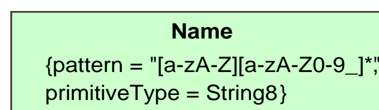

Figure 3-4: Name

Remark: It is recommended to limit names to a maximum of 32 characters.

3.1.3 Description

A Description holds a description for a model element. This type is used in the Description element of the NamedElement metaclass (see below).

Description
{primitiveType = String8}

Figure 3-5: Description

3.1.4 UUID

A UUID is Universally Unique Identifier (UUID) for model elements, which takes the form of hexadecimal integers separated by hyphens, following the pattern 8-4-4-4-12 as defined by the Open Group. This type is used in the Uuid attribute of the Type metaclass.

UUID
{pattern = "[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}," primitiveType = String8}

Figure 3-6: UUID

3.2 Elements

The Core Elements schema defines the common base class NamedElement and the Document class which is the base for all SMP documents.

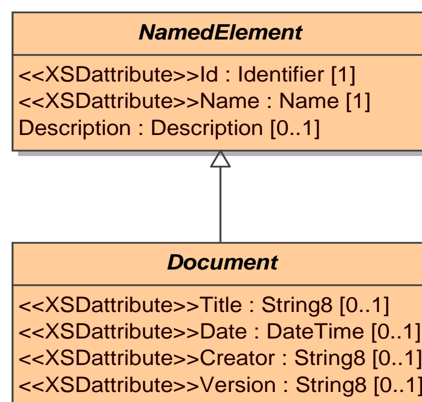

Figure 3-7: Elements

3.2.1 Named Element

The metaclass NamedElement is the common base for most other language elements. A named element has an Id attribute for unique identification in an XML file, a Name attribute holding a human-readable name to be used in applications, and a Description element holding a human-readable description. Furthermore, a named element can hold an arbitrary number of metadata children.

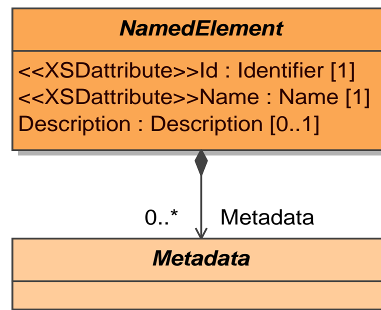

Figure 3-8: Named Element

3.2.2 Document

A Document is a named element that can be the root element of an XML document. It therefore adds the Title, Date, Creator and Version attributes to allow identification of documents.

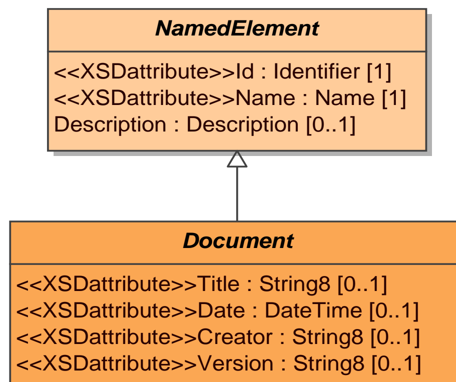

Figure 3-9: Document

3.3 Metadata

Metadata is additional, named information stored with a named element. It is used to further annotate named elements, as the Description element is typically not sufficient.

Metadata can either be a simple Comment, a link to external Documentation, or an Attribute. Please note that the Attribute metaclass is shown on this diagram already, but not defined in the Core Elements schema. It is added by the Core Types schema later, because attributes are typed by an attribute type.

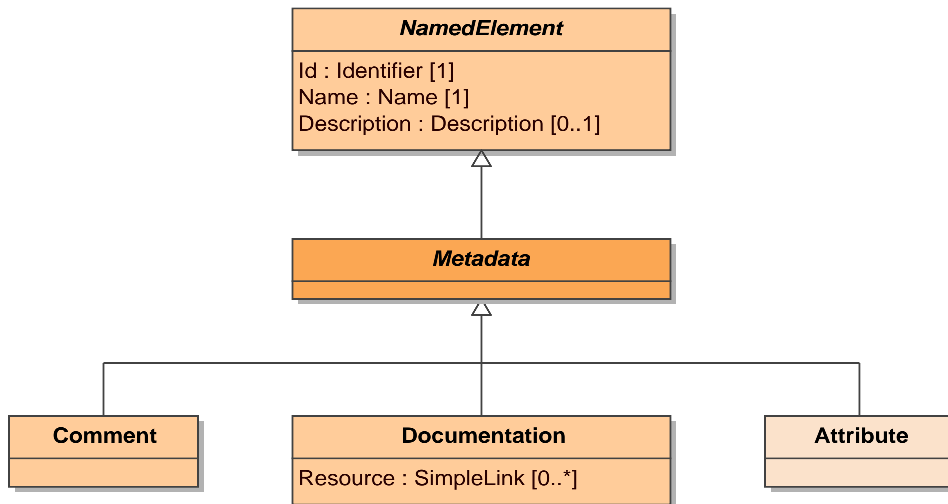

Figure 3-10: Metadata

3.3.1 Metadata

Metadata is additional, named information stored with a named element. It is used to further annotate named elements, as the Description element is typically not sufficient.

3.3.2 Comment

A Comment element holds user comments, e.g. for reviewing models. The Name of a comment should allow to reference the comment (e.g. contain the author's initials and a unique number), while the comment itself is stored in the Description.

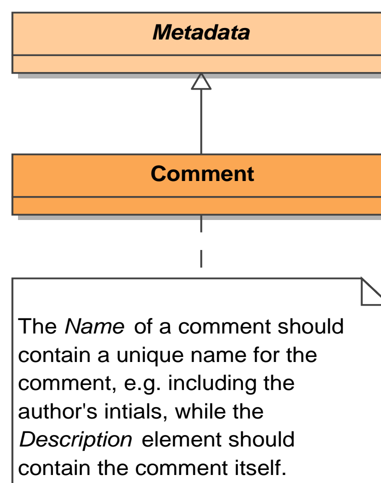

Figure 3-11: Comment

3.3.3 Documentation

A Documentation element holds additional documentation, possibly together with links to external resources. This is done via the Resource element (e.g. holding links to external documentation, 3d animations, technical drawings, CAD models, etc.), based on the XML linking language.

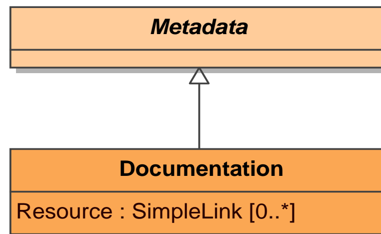

Figure 3-12: Documentation

4
Core Types

This package provides basic types and typing mechanisms, together with appropriate value specification mechanisms.

4.1 Types

Types are used in different contexts. The most common type is a LanguageType, but typing is used as well for other mechanisms, e.g. for Attributes (and later for Events). While this schema introduces basic types, more advanced types used specifically within SMDL Catalogues are detailed later.

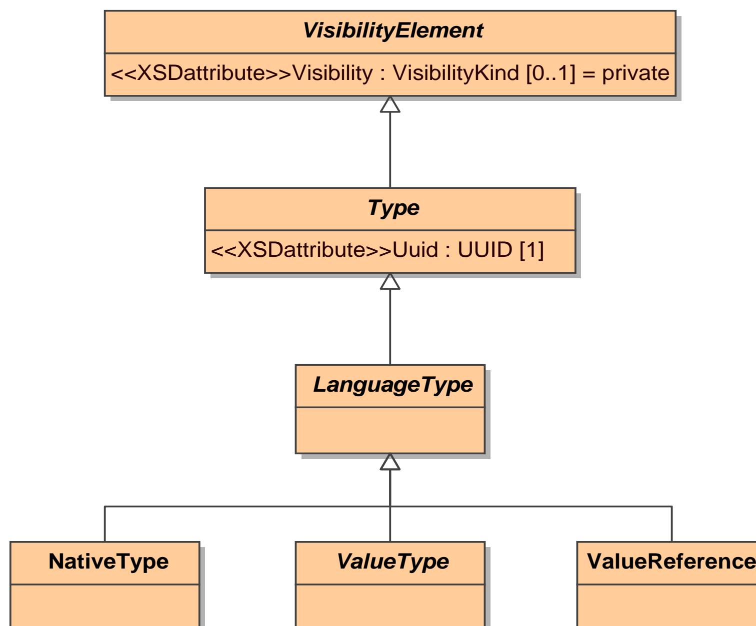

Figure 4-1: Types

4.1.1 Visibility Element

A VisibilityElement is a named element that can be assigned a Visibility attribute to limit its scope of visibility. The visibility may be global (public), local to the parent (private), local to the parent and derived types thereof (protected), or package global (package).

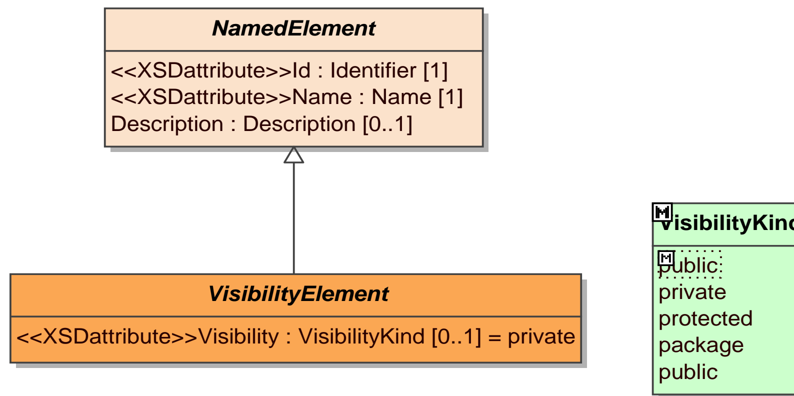

Figure 4-2: Visibility Element

4.1.2 Visibility Kind

This enumeration defines the possible values for an element's visibility.

Table 4-1: Enumeration Literals of VisibilityKind

Name	Description
private	The element is visible only within its containing classifier.
protected	The element is visible within its containing classifier and derived classifiers thereof.
package	The element is globally visible inside the package.
public	The element is globally visible.

4.1.3 Type

A Type is the abstract base metaclass for all type definition constructs specified by SMDL. A type must have a Uuid attribute representing a Universally Unique Identifier (UUID) as defined above. This is needed such that implementations may reference back to their specification without the need to directly reference an XML element in the catalogue.

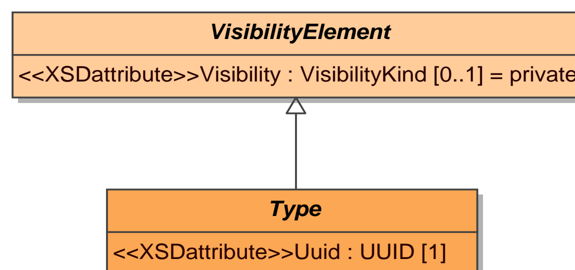

Figure 4-3: Type

4.1.4 Language Type

A LanguageType is the abstract base metaclass for value types (where instances are defined by their value), and references to value types. Also the Smdl Catalogue schema defines reference types (where instances are defined by their reference, i.e. their location in memory) which are derived from LanguageType as well.

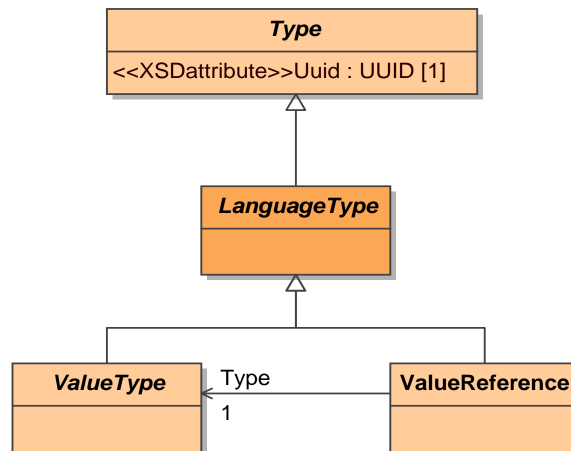

Figure 4-4: Language Type

4.1.5 Value Type

An instance of a ValueType is uniquely determined by its value. Two instances of a value type are said to be equal if they have equal values. Value types include simple types like enumerations and integers, and composite types like structures and arrays.

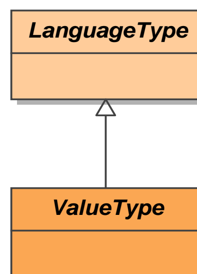

Figure 4-5: Value Type

4.1.6 Value Reference

A ValueReference is a type that references a specific value type. It is the "missing link" between value types and reference types.

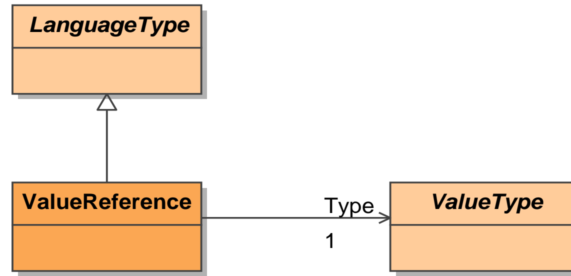

Figure 4-6: Value Reference

4.1.7 Native Type

A NativeType specifies a type with any number of platform mappings. It is used to anchor existing or user-defined types into different target platforms. This mechanism is used within the specification to define the SMDL primitive types with respect to the Metamodel, but it can also be used to define native types within an arbitrary SMDL catalogue for use by models. In the latter case, native types are typically used to bind a model to some external library or existing Application Programming Interface (API).

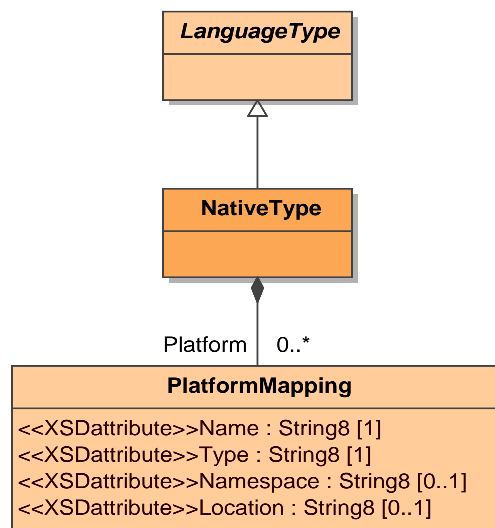

Figure 4-7: Native Type

4.1.8 Platform Mapping

A PlatformMapping defines the mapping of a native type into a target platform. The Name attribute specifies the platform name (see below), the Type attribute specifies the type name on the platform, the Namespace attribute specifies the type's namespace (if any) on the target platform, and the Location attribute specifies where the type is located. Note that the interpretation of these values is platform specific.

The platform name shall be specified using the pattern <platform><environment>, where the environment is optional and may further detail the platform. Some examples are:

- cpp: Standard ANSI/ISO C++ (for all environments)
- cpp__linux__: C++ under the Linux Operating System environment
- idl: CORBA IDL
- xsd: XML Schema
- java: Java language

Basically, any platform mapping may be specified in SMDL as long as the tools - typically code generators working on SMDL Catalogue(s) - have an understanding of their meaning.

The interpretation of the <environment> string may vary between different platforms, and is detailed in each platform mapping document.

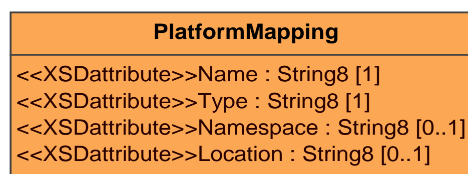

Figure 4-8: Platform Mapping

4.2 Value Types

This package provides mechanisms to specify value types. The shown metaclasses are not the value types themselves, but rather represent language elements (i.e. mechanisms) that can be applied to define actual value types. Please note that the PrimitiveType metaclass has been introduced to allow defining the available base types of SMDL, and is only used internally. Further, the NativeType metaclass provides a generic mechanism to specify native platform specific types, which is also used to define the platform mappings of the SMP primitive types within the Component Model.

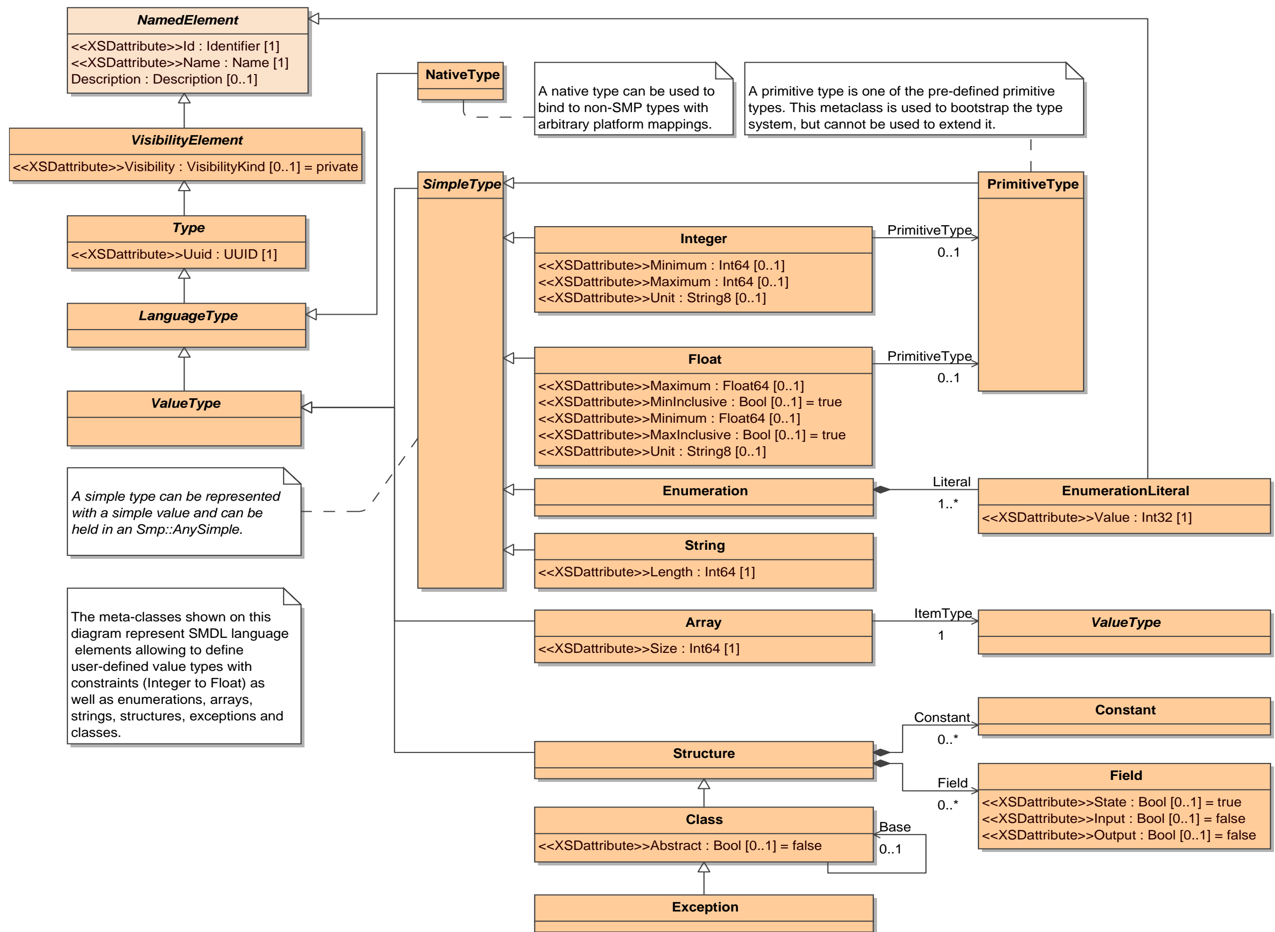

Figure 4-9: Value Types

4.2.1 Simple Type

A simple type is a type that can be represented by a simple value. Simple types include primitive types as well as user-defined Enumeration, Integer, Float and String types.

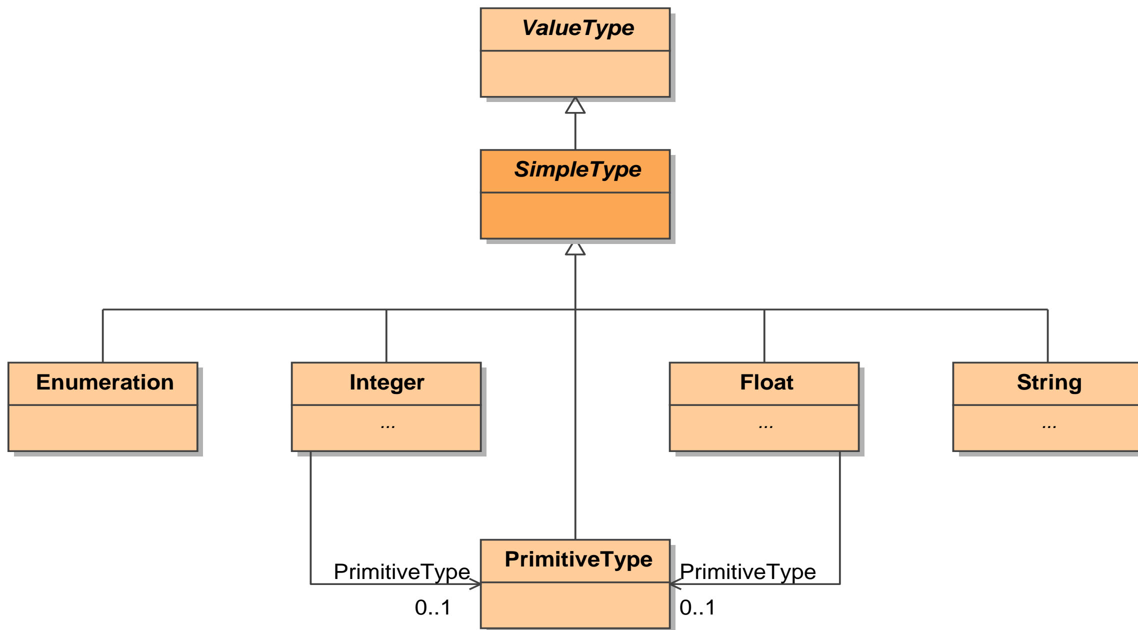

Figure 4-10: Simple Type

4.2.2 Primitive Type

A number of pre-defined types are needed in order to bootstrap the type system. These pre-defined value types are represented by instances of the metaclass PrimitiveType.

This mechanism is only used in order to bootstrap the type system and may not be used to define new types for modelling. This is an important restriction, as all values of primitive types may be held in a SimpleValue. The metaclasses derived from SimpleValue, however, are pre-defined and cannot be extended.

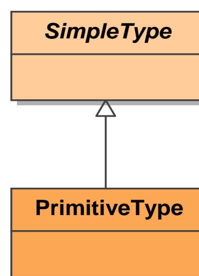

Figure 4-11: Primitive Type

4.2.3 Enumeration

An Enumeration type represents one of a number of pre-defined enumeration literals. The Enumeration language element can be used to create user-defined enumeration types. An enumeration must always contain at least one EnumerationLiteral, each having a name and an integer Value attached to it.

All enumeration literals of an enumeration type must have unique names and values, respectively.

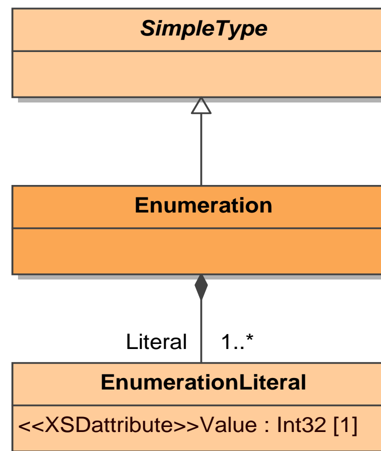

Figure 4-12: Enumeration

4.2.4 Enumeration Literal

An EnumerationLiteral assigns a Name (inherited from NamedElement) to an integer Value.

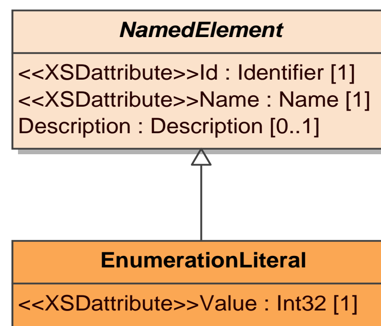

Figure 4-13: Enumeration Literal

4.2.5 Integer

An Integer type represents integer values with a given range of valid values (via the Minimum and Maximum attributes). The Unit element can hold a physical unit that can be used by applications to ensure physical unit integrity across models.

Optionally, the PrimitiveType used to encode the integer value may be specified (one of Int8, Int16, Int32, Int64, UIn8, UInt16, UInt32, where the default is Int32).

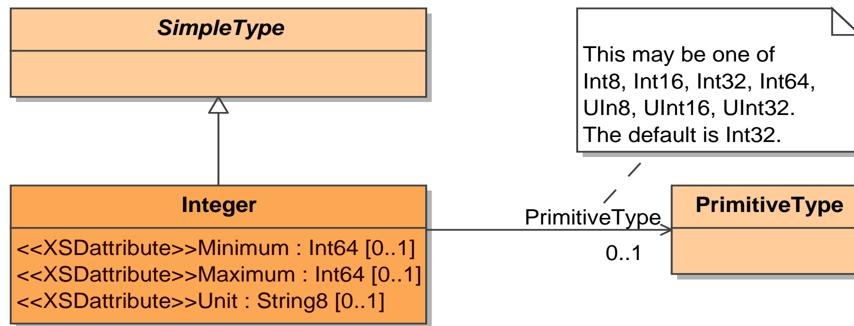

Figure 4-14: Integer

4.2.6 Float

A Float type represents floating-point values with a given range of valid values (via the Minimum and Maximum attributes). The MinInclusive and MaxInclusive attributes determine whether the boundaries are included in the range or not. The Unit element can hold a physical unit that can be used by applications to ensure physical unit integrity across models.

Optionally, the PrimitiveType used to encode the floating-point value may be specified (one of Float32 or Float64, where the default is Float64).

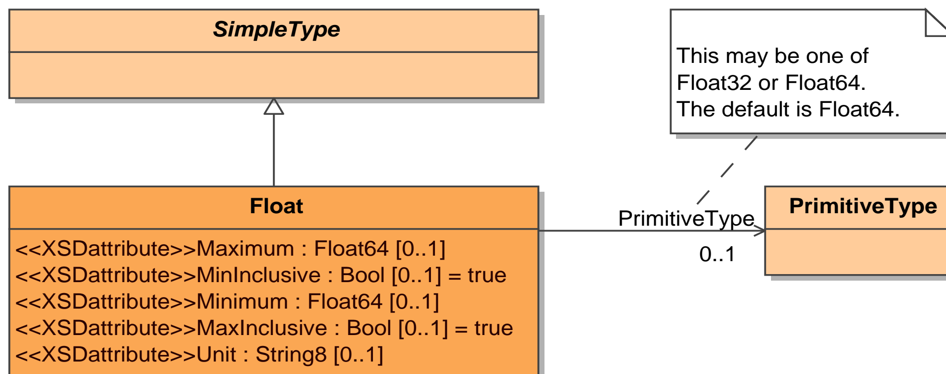

Figure 4-15: Float

4.2.7 String

A String type represents fixed Length string values base on Char8. The String language element defines an Array of Char8 values, but allows a more natural handling of it, e.g. by storing a string value as one string, not as an array of individual characters.

As with arrays, SMDL does not allow defining variable-sized strings, as these have the same problems as dynamic arrays (e.g. their size is not know up-front, and their use requires memory allocation).

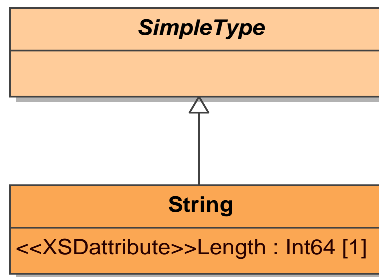

Figure 4-16: String

4.2.8 Array

An Array type defines a fixed-size array of identically typed elements, where ItemType defines the type of the array items, and Size defines the number of array items.

Multi-dimensional arrays are defined when ItemType is an Array type as well.

Dynamic arrays are not supported by SMDL, as they are not supported by some potential target platforms, and introduce various difficulties in memory management.

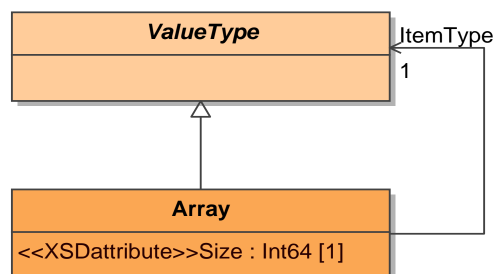

Figure 4-17: Array

Remarks: Nevertheless, specific mechanisms are available to allow dynamic collections of components, either for containment (composition) or references (aggregation).

4.2.9 Structure

A Structure type collects an arbitrary number of Fields representing the state of the structure.

Within a structure, each field needs to be given a unique name. In order to arrive at semantically correct (data) type definitions, a structure type may not be recursive, i.e. a structure may not have a field that is typed by the structure itself.

A structure can also serve as a namespace to define an arbitrary number of Constants.

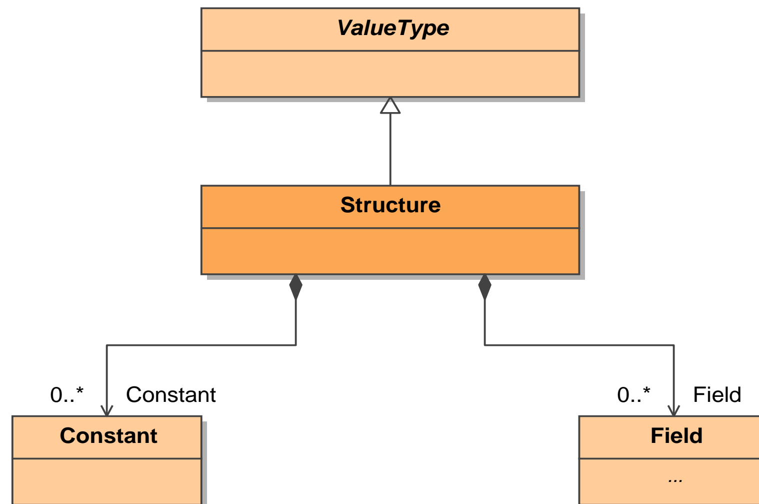

Figure 4-18: Structure

4.2.10 Exception

An Exception represents a non-recoverable error that can occur when calling into an Operation or Property getter/setter (within an Operation this is represented by the RaisedException links and within a Property this is represented by the GetRaises and SetRaises links, respectively).

An Exception can contain constants and fields (from Structure) as well as operations, properties and associations (from Class). The fields represent the state variables of the exception which carry additional information when the exception is raised.

Furthermore, an Exception may be Abstract (from Class), and it may inherit from a single base exception (implementation inheritance), which is represented by the Base link (from Class).

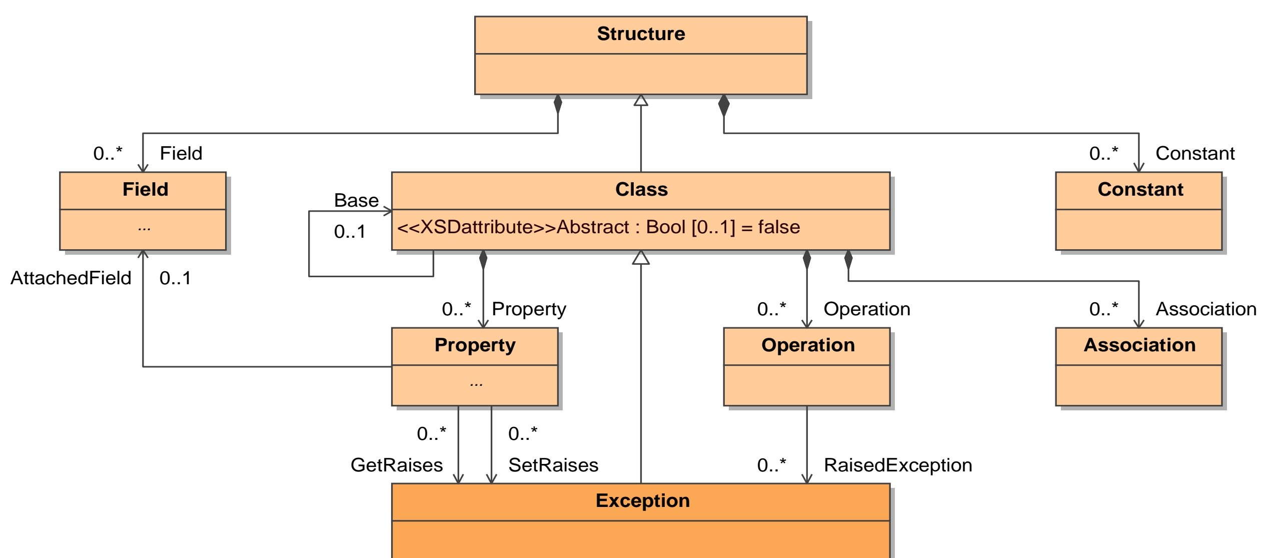

Figure 4-19: Exception

4.2.11 Class

The Class metaclass is derived from Structure. A class may be abstract (attribute Abstract), and it may inherit from a single base class (implementation inheritance), which is represented by the Base link.

As the Class metaclass is derived from Structure it can contain constants and fields. Further, it can have arbitrary numbers of properties (Property elements), operations (Operation elements), and associations (Association elements).

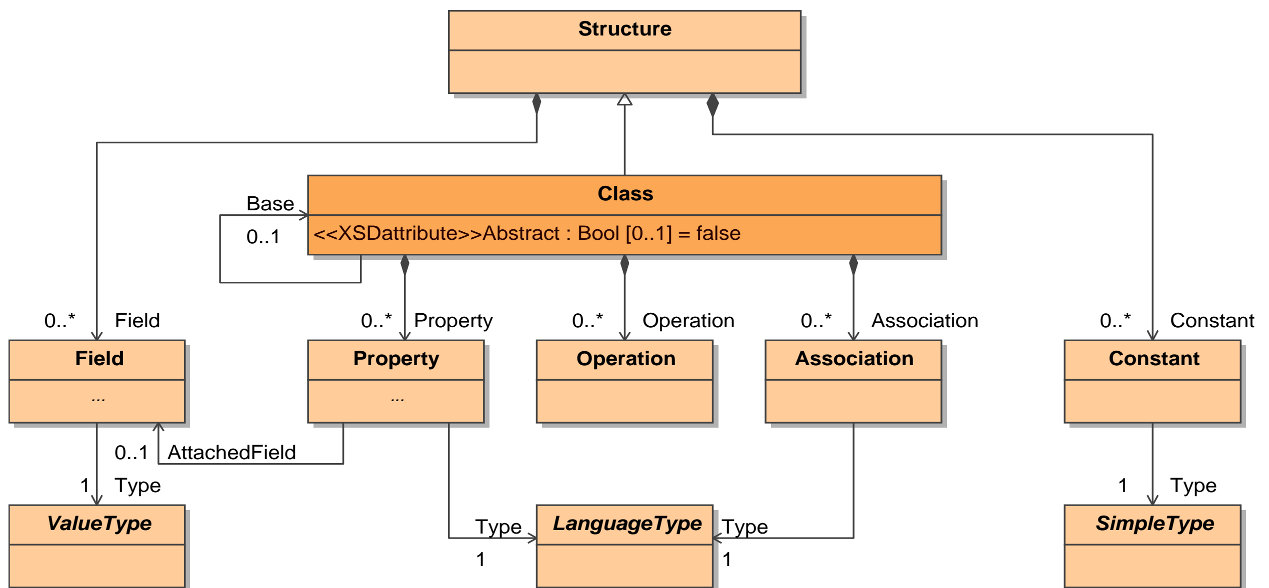

Figure 4-20: Class

4.3 Features

A feature is an element that is contained in a type and that typically refers to a type. Additionally, some features have (default) values.

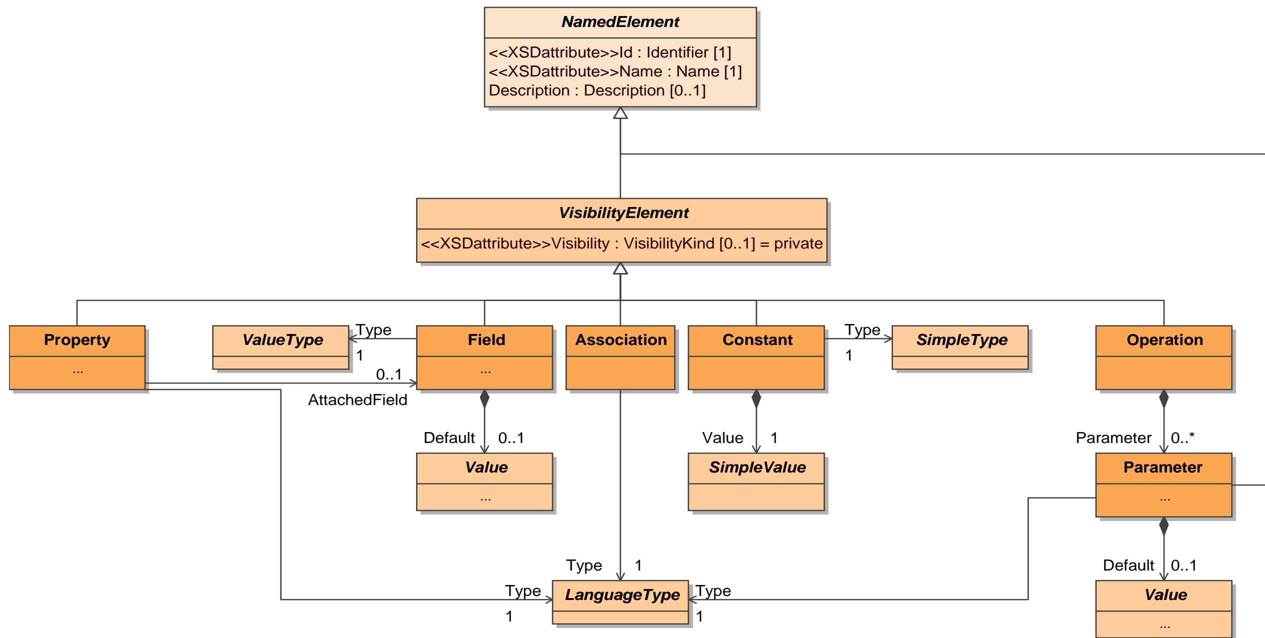

Figure 4-21: Features

4.3.1 Constant

A Constant is a feature that is typed by a simple type and that must have a Value.

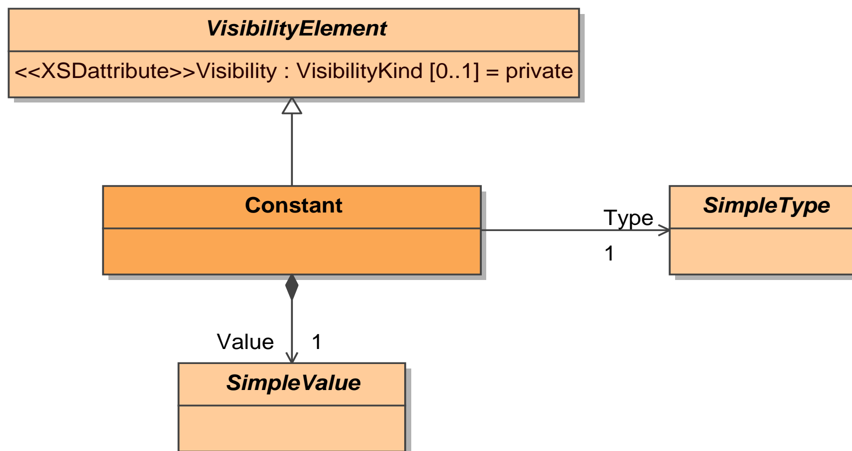

Figure 4-22: Constant

4.3.2 Field

A Field is a feature that is typed by any value type but String8, and that may have a Default value.

The State attribute defines how the field is published to the simulation environment. Only fields with a State of true are stored using external persistence. The visibility to the user within the simulation environment can be

controlled via the standard SMP attribute "View". By default, the State flag is set to true and the View attribute defaults to "None" when not applied.

The Input and Output attributes define whether the field value is an input for internal calculations (i.e. needed in order to perform these calculations), or an output of internal calculations (i.e. modified when performing these calculations). These flags default to false, but can be changed from their default value to support dataflow-based design.

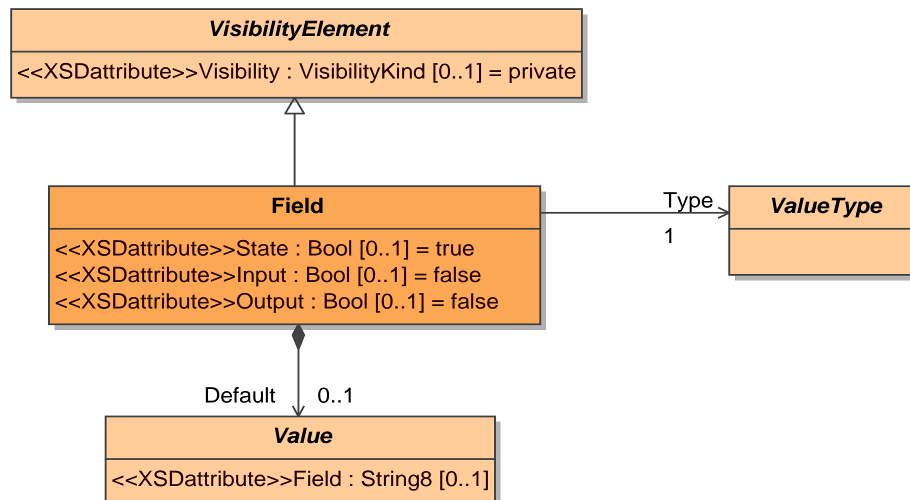

Figure 4-23: Field

4.3.3 Property

A Property has a similar syntax as a Field: It is a feature that references a language type. However, the semantics is different in that a property does not represent a state and that it can be assigned an Access attribute to specify how the property can be accessed (either readWrite, readOnly, or writeOnly, see AccessKind).

Furthermore, a property can be assigned a Category attribute to help grouping the properties within its owning type, and a property may specify an arbitrary number of exceptions that it can raise in its getter (GetRaises) and/or setter (SetRaises).

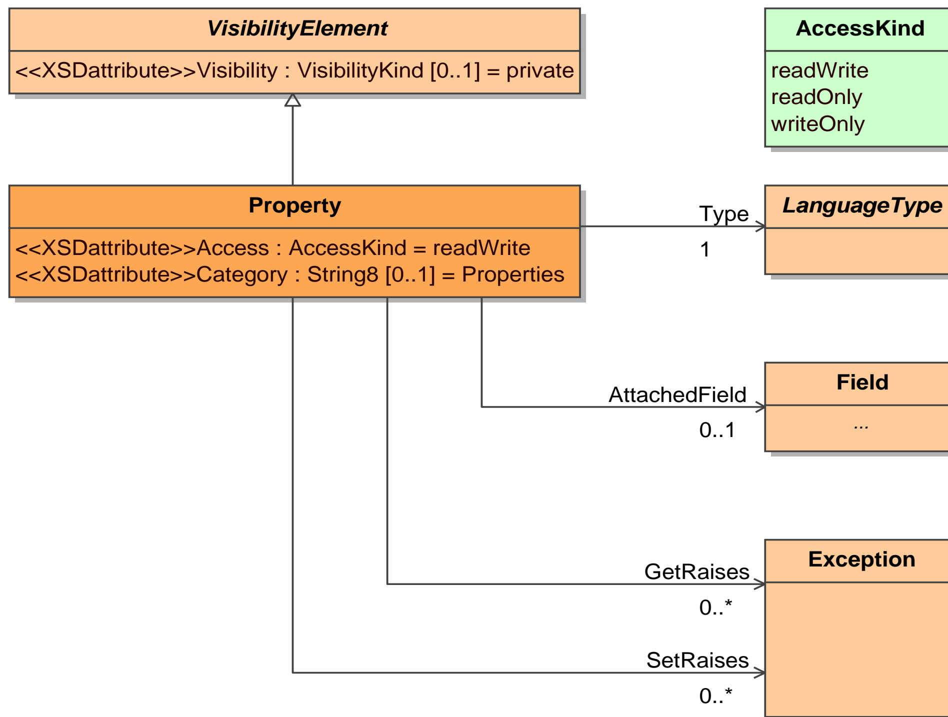

Figure 4-24: Property

Remark: The category can be used in applications as ordering or filtering criterion, for example in a property grid. The term "property" used here closely corresponds in its semantics to the same term in the Java Beans specification and in the Microsoft .NET framework. That is, a property formally represents a "getter" or a "setter" operation or both which allow accessing state or configuration information (or derived information thereof) in a controlled way and which can also be exposed via interfaces (in contrast to fields).

4.3.4 Access Kind

This enumeration defines how a property can be accessed.

Table 4-2: Enumeration Literals of AccessKind

Name	Description
readWrite	Specifies a property, which has both a getter and a setter.
readOnly	Specifies a property, which only has a getter.
writeOnly	Specifies a property, which only has a setter.

4.3.5 Field versus Property

The semantics of a property is very different from the semantics of a field. A field always has a memory location holding its value, while a property is a convenience mechanism to represent one or two access operations, namely the

setter and/or the getter. If a property is read-only, there is no setter, if it is write-only, there is to getter. The actual implementation depends on the target platform and language.

Compared to fields, properties have the advantage that there is no direct memory access, but every access is operation-based. This allows mapping them to distributed platforms (e.g. CORBA), and ensures that the containing type always has knowledge about changes of its state (e.g. to support range checking in the setter).

On implementation level, properties are frequently bound to a specific field. This can be expressed by linking to a field (of the same type) via the AttachedField link.

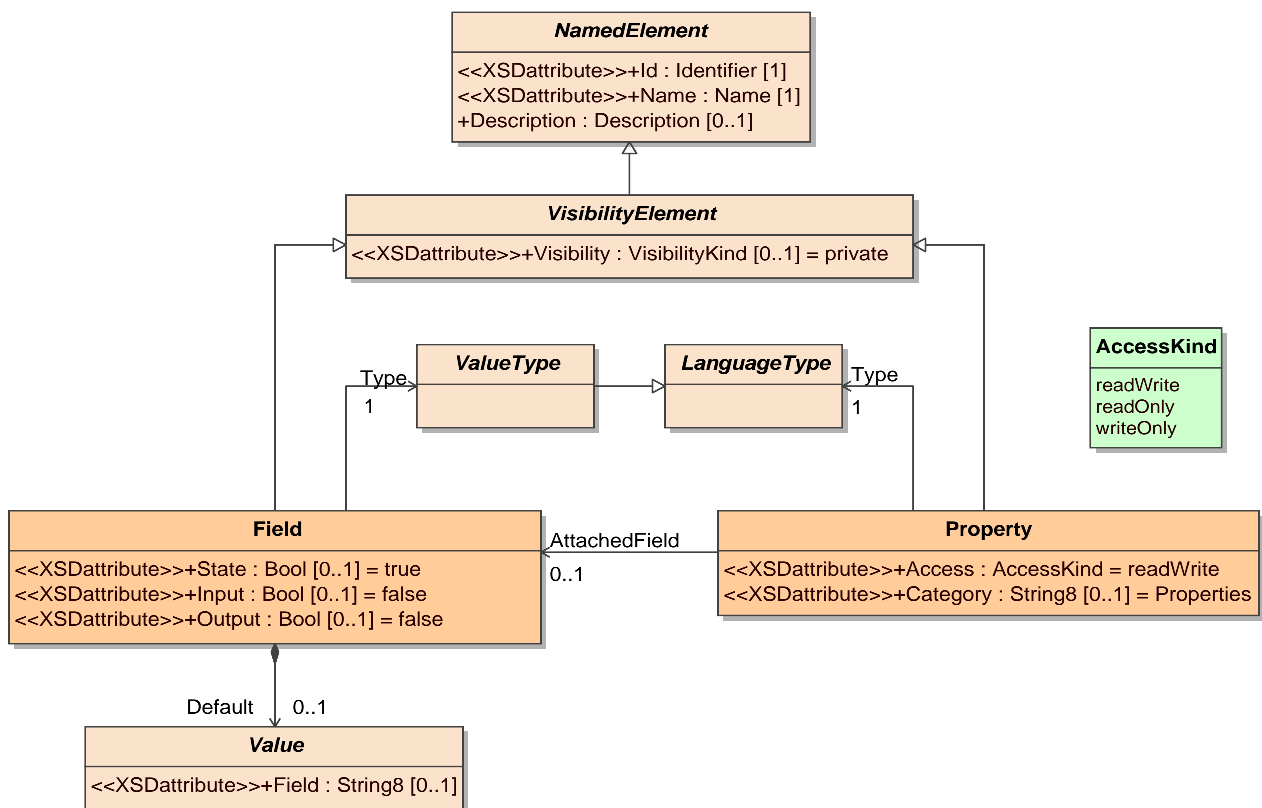

Figure 4-25: Field versus Property

Remark: For example, this information can be utilised by a code generator to generate the relevant binding from the setter and/or the getter to the attached field in the code.

4.3.6 Association

An Association is a feature that is typed by a language type (Type link). An association always expresses a reference to an instance of the referenced language type. This reference is either another model (if the Type link refers to a Model or Interface), or it is a field contained in another model (if the Type link refers to a ValueType).

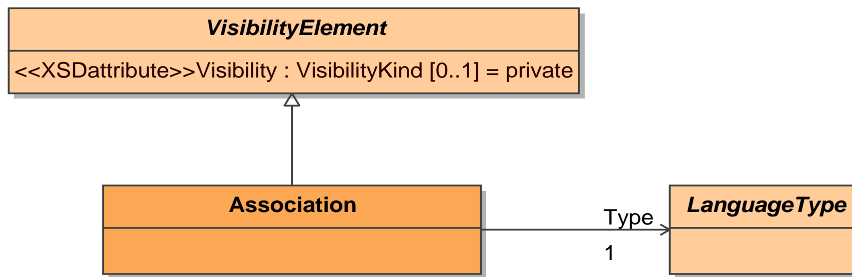

Figure 4-26: Association

4.3.7 Operation

An Operation may have an arbitrary number of parameters, where at most one of the parameters may be of Direction = ParameterDirectionKind.return. If such a parameter is absent, the operation is a void function (procedure) without return value.

An Operation may specify an arbitrary number of exceptions that it can raise (RaisedException).

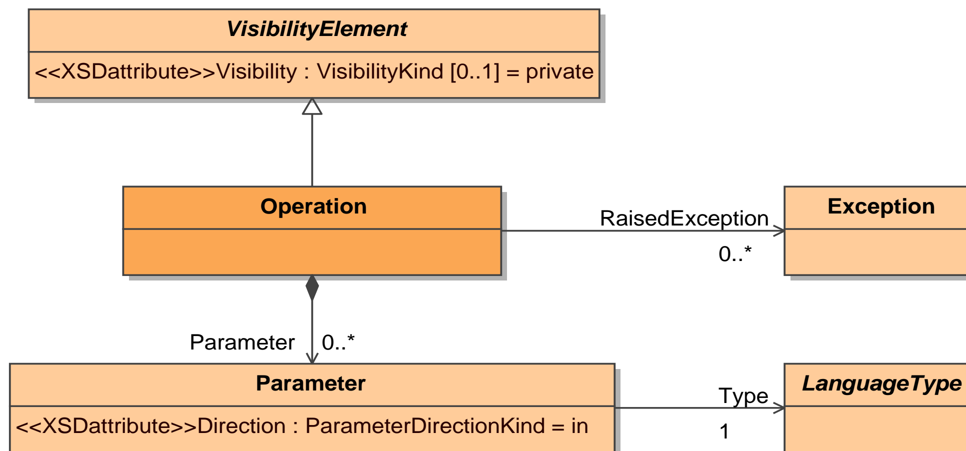

Figure 4-27: Operation

4.3.8 Parameter

A Parameter has a Type and a Direction, where the direction may have the values in, out, inout or return (see ParameterDirectionKind).

When referencing a value type, a parameter may have an additional Default value, which can be used by languages that support default values for parameters.

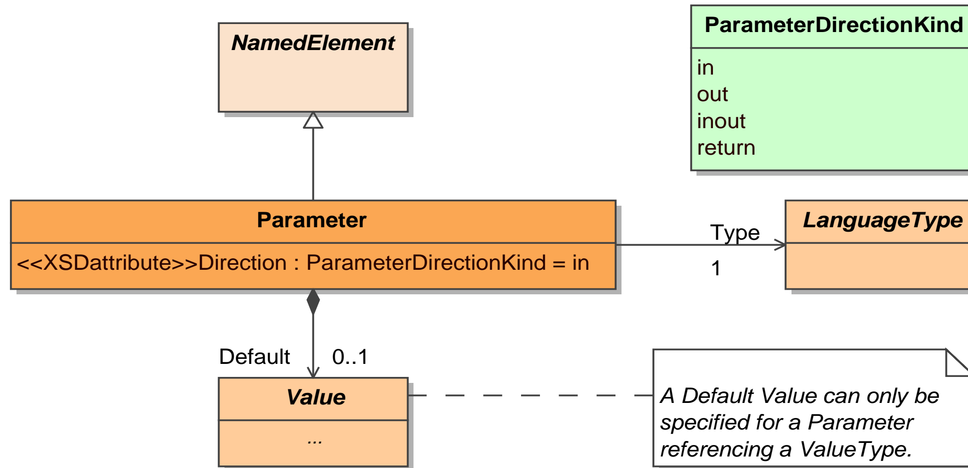

Figure 4-28: Parameter

4.3.9 Parameter Direction Kind

This enumeration defines the possible parameter directions.

Table 4-3: Enumeration Literals of ParameterDirectionKind

Name	Description
in	The parameter is read-only to the operation, i.e. its value must be specified on call, and cannot be changed inside the operation.
out	The parameter is write-only to the operation, i.e. its value is unspecified on call, and must be set by the operation.
inout	The parameter must be specified on call, and may be changed by the operation.
return	The parameter represents the operation's return value.

4.4 Values

A Value represents the state of a ValueType. For each metaclass derived from ValueType, a corresponding metaclass derived from Value is defined. Values are used in various places. Within the Core Types schema, they are used for the Default value of a Field, Parameter and AttributeType and for the Value of a Constant.

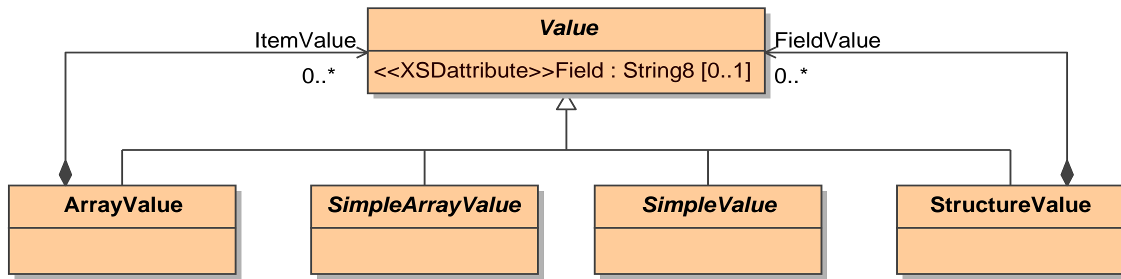

Figure 4-29: Values

4.4.1 Value

The Value metaclass is an abstract base class for specialised values.

The Field attribute specifies the reference to the corresponding field via its name or its locally qualified path. This attribute can be omitted in cases where no field reference needs to be specified (e.g. on a default value of a Parameter).

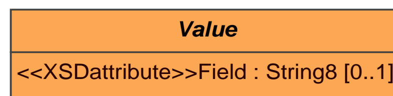

Figure 4-30: Value

4.4.2 Simple Value

A SimpleValue represents a value that is of simple type (this includes all SMP primitive types as well as user-defined Integer, Float, String and Enumeration types).

To ensure type safety, specific sub-metaclasses are introduced, which specify the type of the Value attribute.

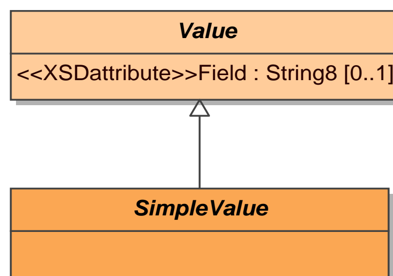

Figure 4-31: Simple Value

4.4.3 Simple Array Value

A SimpleArrayValue represents an array of values that are of (the same) simple type.

To ensure type safety, specific sub-metaclasses are introduced, which specify the type of the contained ItemValue elements.

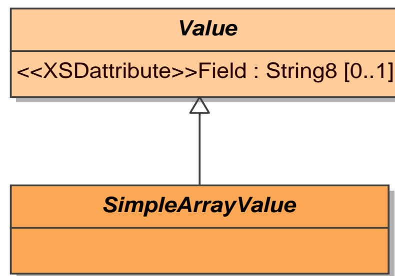

Figure 4-32: Simple Array Value

4.4.4 Array Value

An ArrayValue holds values for each array item, represented by the ItemValue elements. The corresponding array type defines the number of item values.

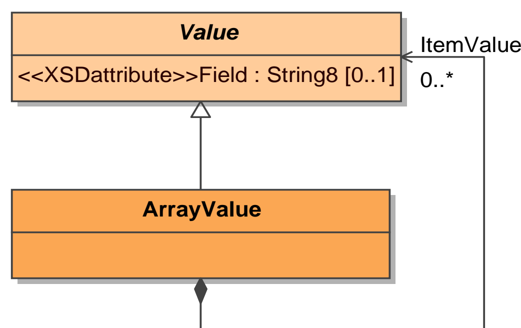

Figure 4-33: Array Value

4.4.5 Structure Value

A StructureValue holds field values for all fields of the corresponding structure type. Thereby, the Field attribute of each contained value specifies the local field name within the structure.

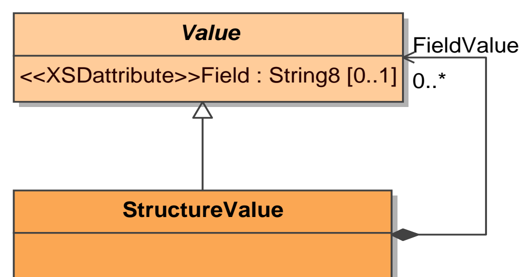

Figure 4-34: Structure Value

4.4.6 Simple Values

Values corresponding to simple types are introduced as specialized metaclasses in order to allow type-safe and efficient XML serialization. A specific metaclass is introduced for each SMP primitive type and to specify enumeration values.

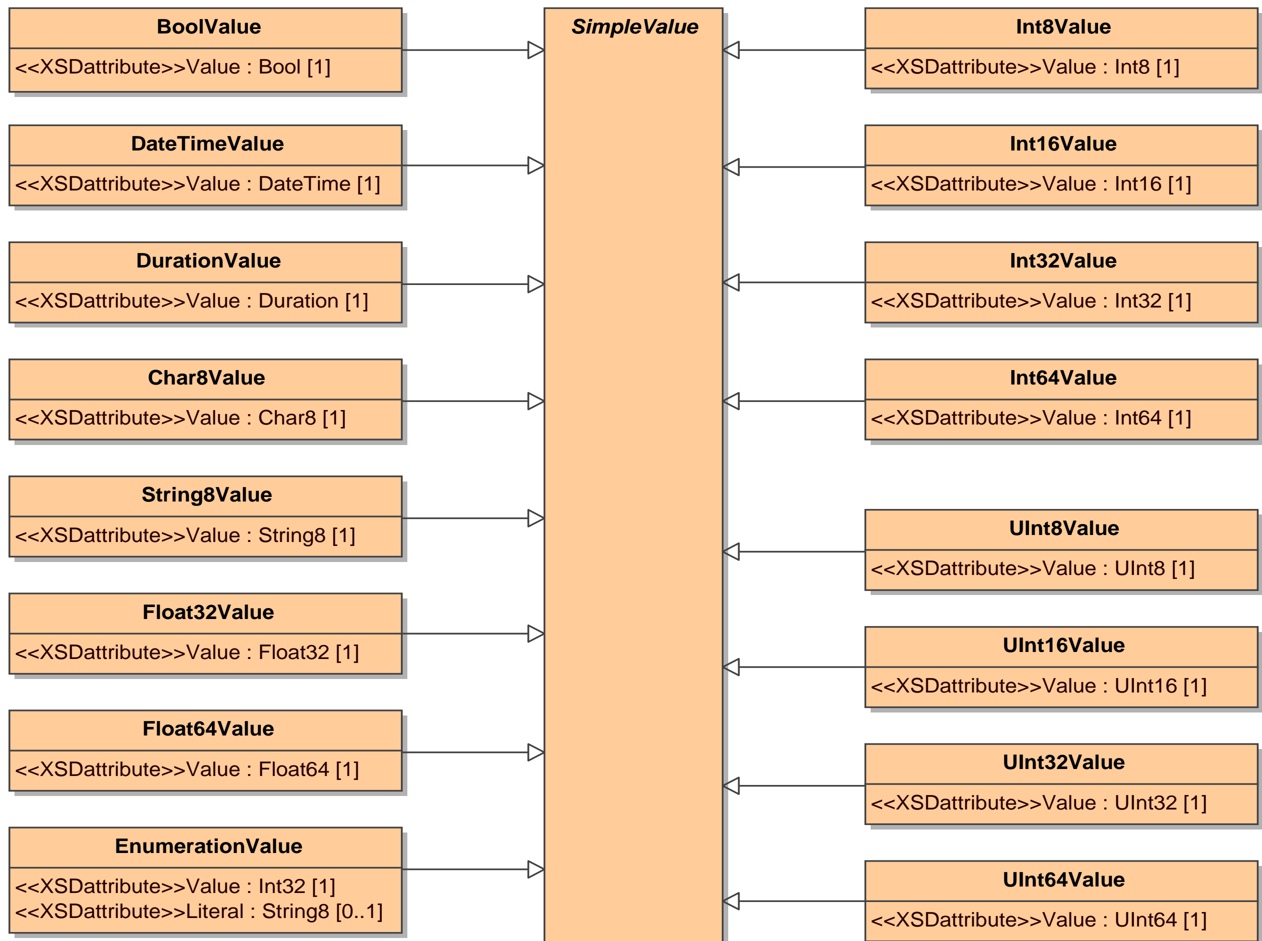

Figure 4-35: Simple Values

4.4.6.1 Bool Value

A BoolValue holds a value for an item of type Bool.

4.4.6.2 Char8Value

A Char8Value holds a value for an item of type Char8.

4.4.6.3 Date Time Value

A DateTimeValue holds a value for an item of type DateTime.

4.4.6.4 Duration Value

A DurationValue holds a value for an item of type Duration.

4.4.6.5 Enumeration Value

An EnumerationValue holds a value for an item of an Enumeration type.

4.4.6.6 Float32Value

A Float32Value holds a value for an item of type Float32.

4.4.6.7 Float64Value

A Float64Value holds a value for an item of type Float64.

4.4.6.8 Int16Value

An Int16Value holds a value for an item of type Int16.

4.4.6.9 Int32Value

An Int32Value holds a value for an item of type Int32.

4.4.6.10 Int64Value

An Int64Value holds a value for an item of type Int64.

4.4.6.11 Int8Value

An Int8Value holds a value for an item of type Int8.

4.4.6.12 String8Value

A String8Value holds a value for an item of type String8, or for an item of a String type.

4.4.6.13 UInt16Value

A UInt16Value holds a value for an item of type UInt16.

4.4.6.14 UInt32Value

A UInt32Value holds a value for an item of type UInt32.

4.4.6.15 UInt64Value

A UInt64Value holds a value for an item of type UInt64.

4.4.6.16 UInt8Value

A UInt8Value holds a value for an item of type UInt8.

4.4.7 Simple Array Values

Values of arrays with items of simple type are introduced as specialized metaclasses in order to allow type-safe and efficient XML serialization. A

specific metaclass is introduced for each SMP primitive type and to specify enumeration values.

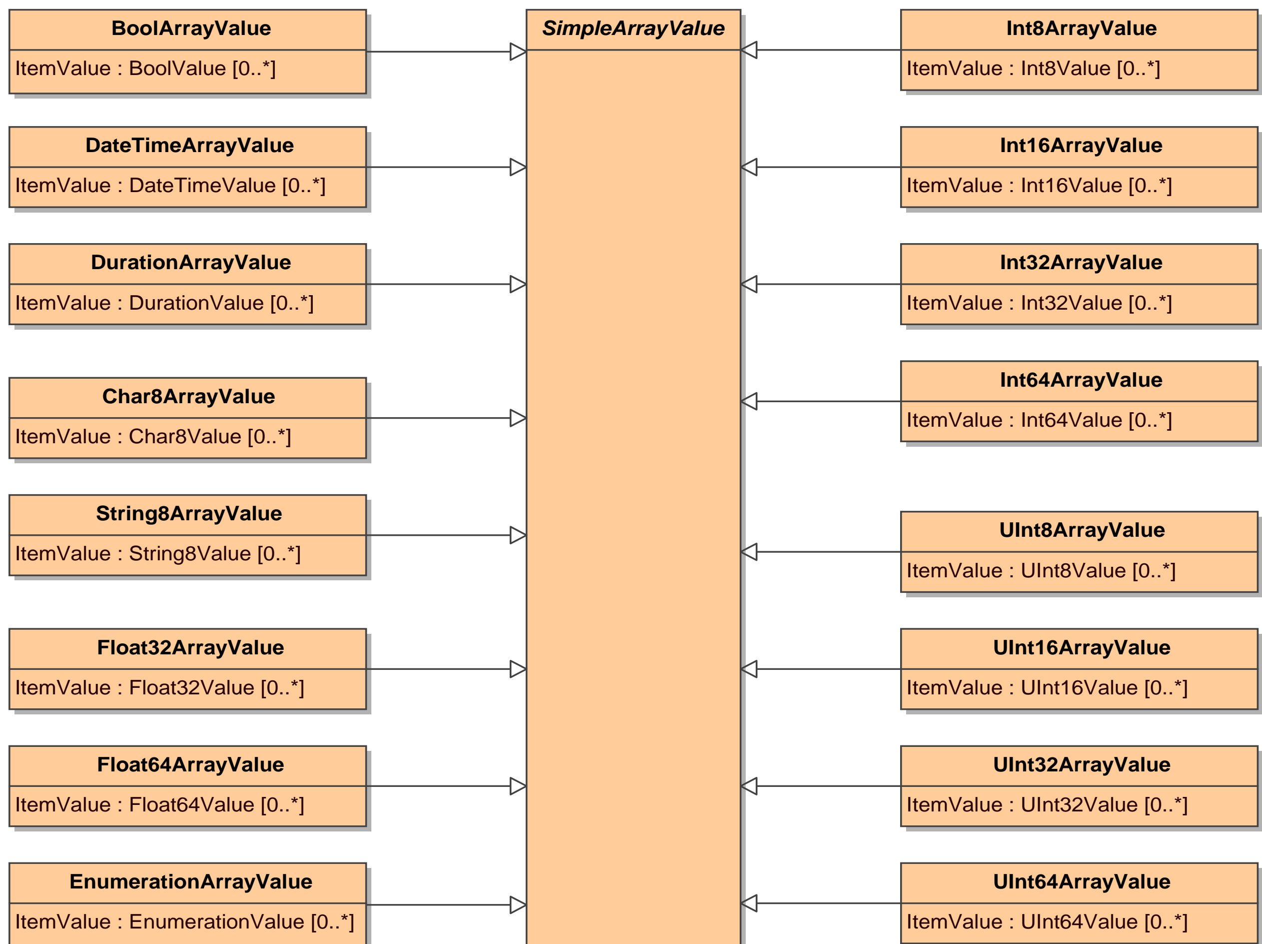

Figure 4-36: Simple Array Values

4.4.7.1 Bool Array Value

The BoolArrayValue holds an array of BoolValue items for an array of type Bool.

4.4.7.2 Char8Array Value

The Char8ArrayValue holds an array of Char8Value items for an array of type Char8.

4.4.7.3 Date Time Array Value

The DateTimeArrayValue holds an array of DateTimeValue items for an array of type DateTime.

4.4.7.4 Duration Array Value

The DurationArrayValue holds an array of DurationValue items for an array of type Duration.

4.4.7.5 Enumeration Array Value

The EnumerationArrayValue holds an array of EnumerationValue items for an array of an Enumeration type.

4.4.7.6 Float32Array Value

The Float32ArrayValue holds an array of Float32Value items for an array of type Float32.

4.4.7.7 Float64Array Value

The Float64ArrayValue holds an array of Float64Value items for an array of type Float64.

4.4.7.8 Int16Array Value

The Int16ArrayValue holds an array of Int16Value items for an array of type Int16.

4.4.7.9 Int32Array Value

The Int32ArrayValue holds an array of Int32Value items for an array of type Int32.

4.4.7.10 Int64Array Value

The Int64ArrayValue holds an array of Int64Value items for an array of type Int64.

4.4.7.11 Int8Array Value

The Int8ArrayValue holds an array of Int8Value items for an array of type Int8.

4.4.7.12 String8Array Value

The String8ArrayValue holds an array of String8Value items for an array of type String8, or for an array of a String type.

4.4.7.13 UInt16Array Value

The UInt16ArrayValue holds an array of UInt16Value items for an array of type UInt16.

4.4.7.14 UInt32Array Value

The UInt32ArrayValue holds an array of UInt32Value items for an array of type UInt32.

4.4.7.15 UInt64Array Value

The UInt64ArrayValue holds an array of UInt64Value items for an array of type UInt64.

4.4.7.16 UInt8Array Value

The UInt8ArrayValue holds an array of UInt8Value items for an array of type UInt8.

4.5 Attributes

This package defines the SMDL attribute mechanism which allows extending SMDL semantics via standard or user-defined attributes.

Remark: In SMDL, the term attribute is used to denote user-defined metadata, as in the .NET framework. In contrast, an attribute in UML denotes a non-functional member of a class, which corresponds to a field or property in SMDL.

4.5.1 Attribute Type

An AttributeType defines a new type available for adding attributes to elements. The AllowMultiple attribute specifies if a corresponding Attribute may be attached more than once to a language element, while the Usage element defines to which language elements attributes of this type can be attached. An attribute type always references a value type, and specifies a Default value.

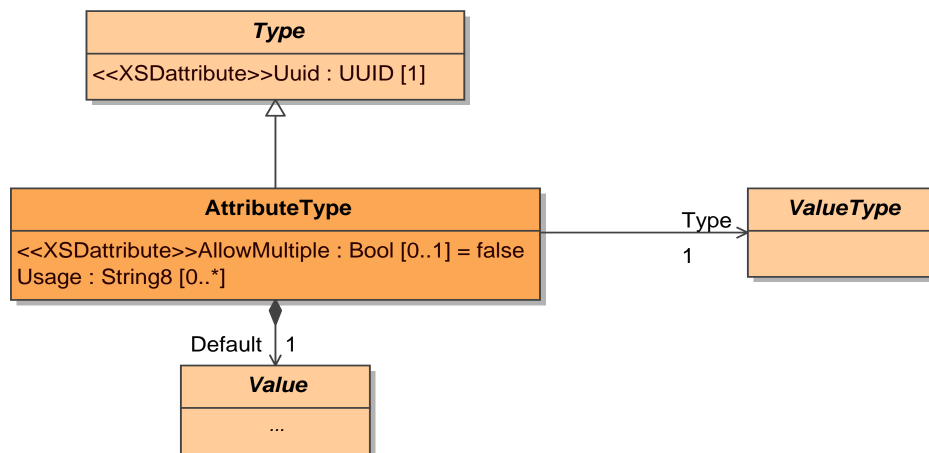

Figure 4-37: Attribute Type

4.5.2 Attribute

An Attribute element holds name-value pairs allowing to attach user-defined metadata to any Element.

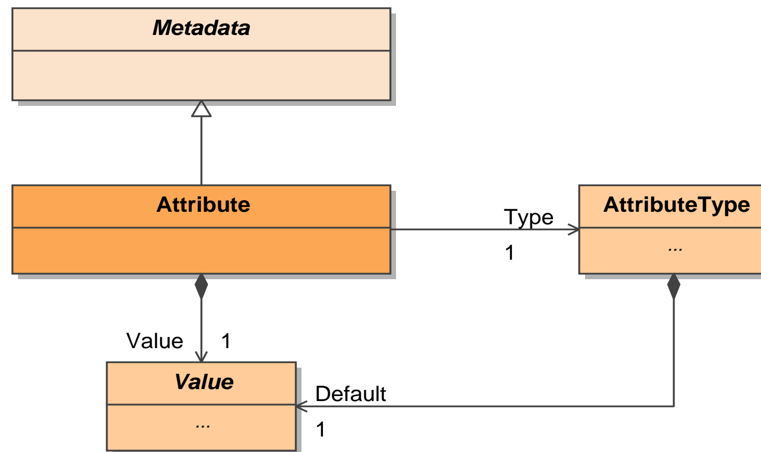

Figure 4-38: Attribute

Remark: This provides a similar mechanism as tagged values in UML, xsd:appinfo in XML Schema, annotations in Java 5.0 or attributes in the .NET framework.

Remark: A possible application of using attributes could be to decorate an SMDL model with information needed to guide a code generator, for example to tailor the mapping to C++.

5 Smdl Catalogue

This package describes all metamodel elements that are needed in order to define models in a catalogue. Catalogues make use of the mechanisms defined in Core Types, e.g. enumerations and structures, and they add reference types (interfaces and components), events and a hierarchical grouping mechanism (namespaces).

5.1 Catalogue

A catalogue contains namespaces, which themselves contain other namespaces and types. Types can either be language types, attribute types (both defined in Core Types) or event types. Further, catalogues extend the available language types by reference types (interfaces and components).

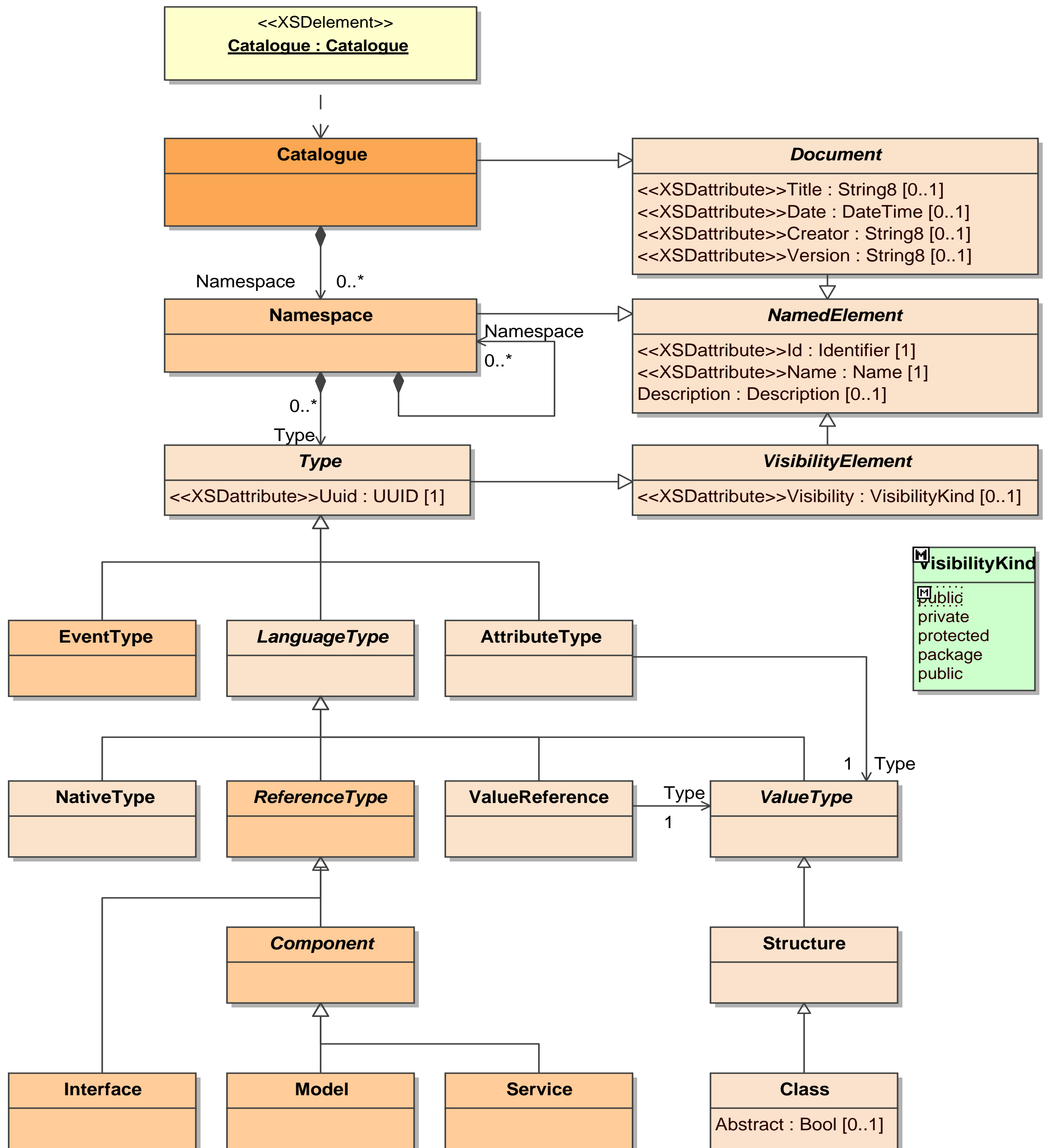

Figure 5-1: Catalogue

5.1.1 Catalogue

A Catalogue is a document that defines types. It contains namespaces as a primary ordering mechanism. The names of these namespaces need to be unique within the catalogue.

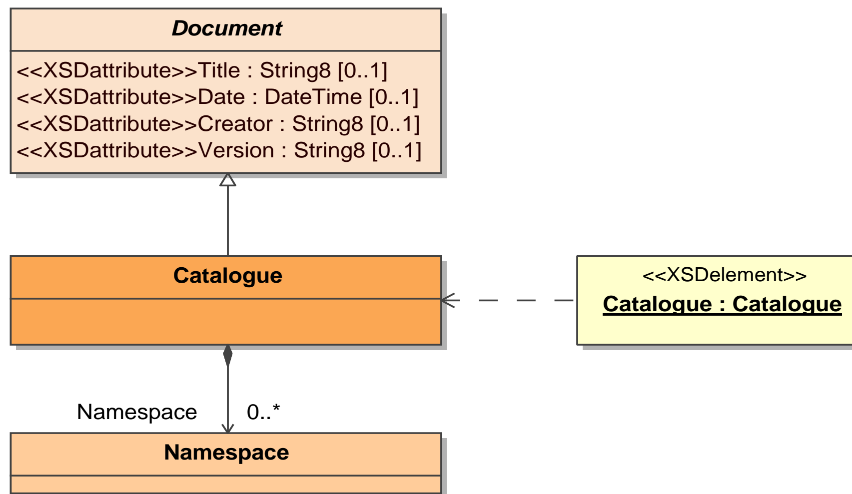

Figure 5-2: Catalogue

5.1.2 Namespace

A Namespace is a primary ordering mechanism. A namespace may contain other namespaces (nested namespaces), and does typically contain types. In SMDL, namespaces are contained within a Catalogue (either directly, or within another namespace in a catalogue).

All sub-elements of a namespace (namespaces and types) must have unique names.

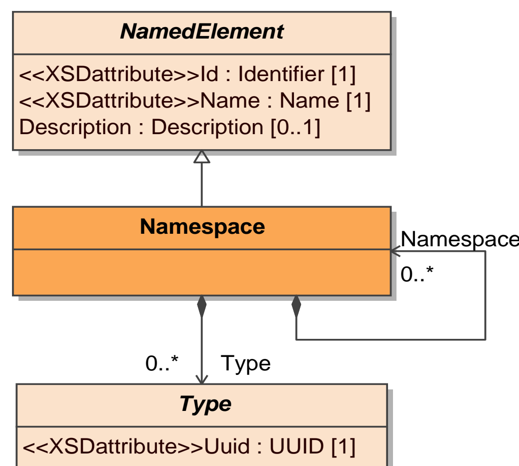

Figure 5-3: Namespace

5.2 Reference Types

An instance of a reference type is uniquely determined by a reference to it, and may have internal state. Two instances of a reference type are equal if and only if they occupy the same (memory) location, i.e. if the references to them are identical. Two instances with equal values may therefore not be equal if they

occupy different (memory) locations. Reference types include interfaces and components (models and services).

Every reference type supports properties and operations. A component adds a number of features for different purposes. First, it adds fields to store an internal state, and provided interfaces for interface-based programming. Further, it adds mechanisms to describe dependencies on other reference types, event sources and sinks for event-based programming, and entry points to allow calling void functions e.g. from a scheduler.

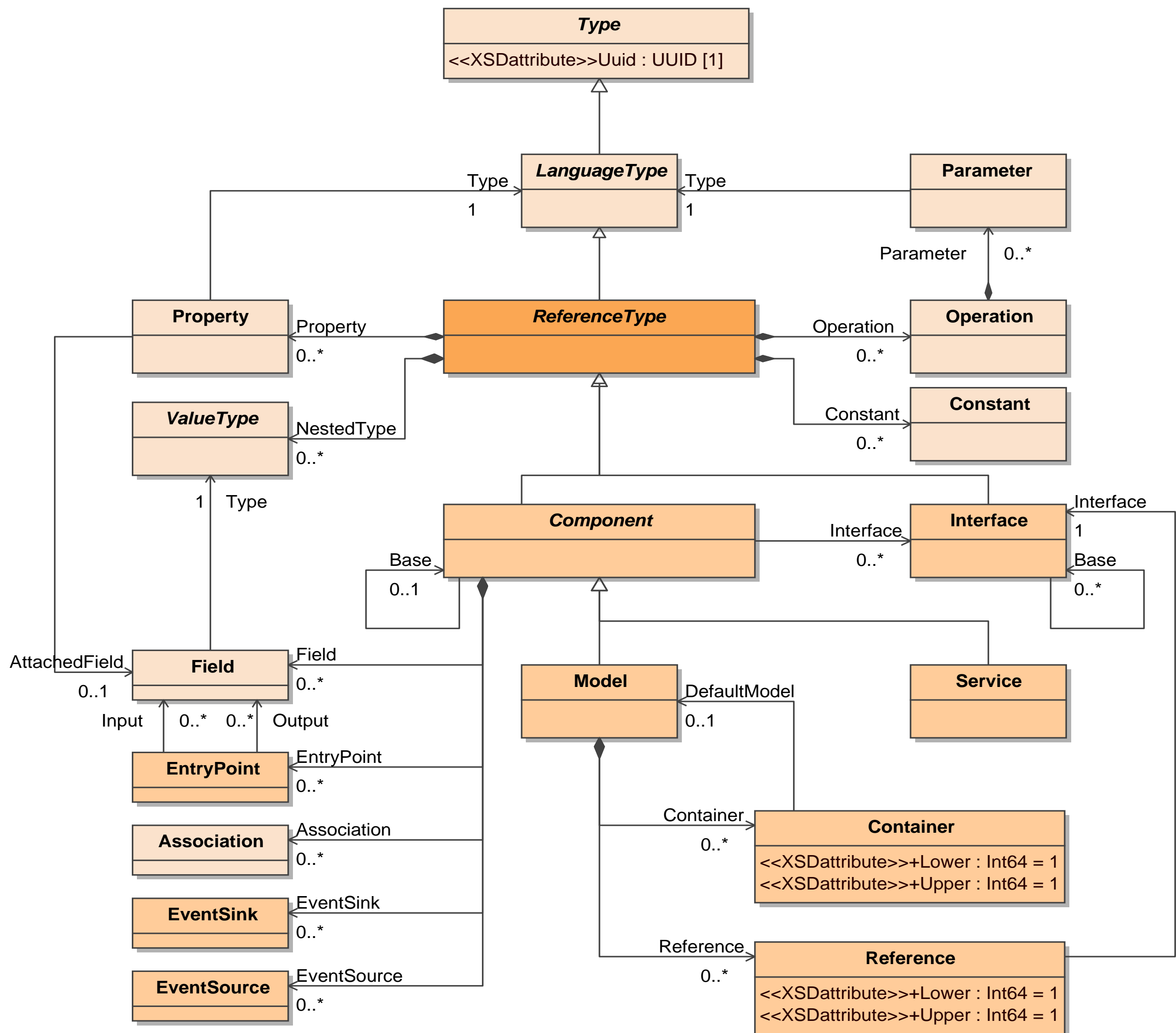

Figure 5-4: Reference Types

5.2.1 Reference Type

ReferenceType serves as an abstract base metaclass for Interface and Component. An instance of a ReferenceType is identified by a reference to it - as opposed to an instance of a ValueType which is identified by its value. A reference type may have various optional elements:

- Constant elements specify constants defined in the reference type's name scope;
- NestedType elements specify (local) value types defined in the reference type's name scope;
- Property elements specify the reference type's properties; and
- Operation elements specify the reference type's operations.

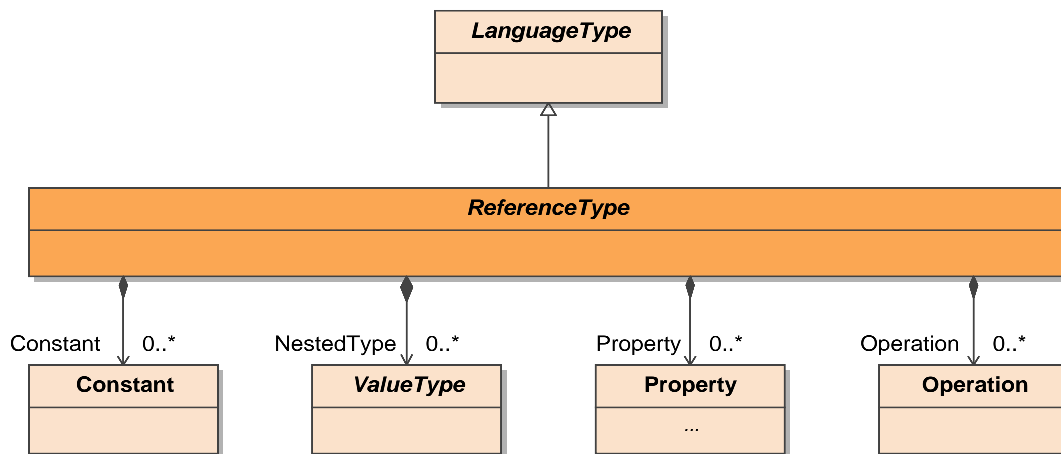

Figure 5-5: Reference Type

5.2.2 Component

A Component is a reference type and hence inherits the ability to hold constants, nested types, properties, and operations. As a Component semantically forms a deployable unit, it may use the available component mechanisms as specified in the SMP Component Model. Apart from the ability to specify a Base component (single implementation inheritance), a component may have various optional elements:

- Interface links specify interfaces that the component provides (in SMP this implies that the component implements these interfaces);
- EntryPoint elements allow the component to be scheduled (via the Scheduler service) and to listen to global events (via the EventManager service);
- EventSink elements specify which events the component can receive (these may be registered with other components' event sources);
- EventSource elements specify events that the component can emit (other components may register their associated event sink(s) with these);
- Field elements define a component's internal state; and
- Association elements express associations to other components or fields of other components.

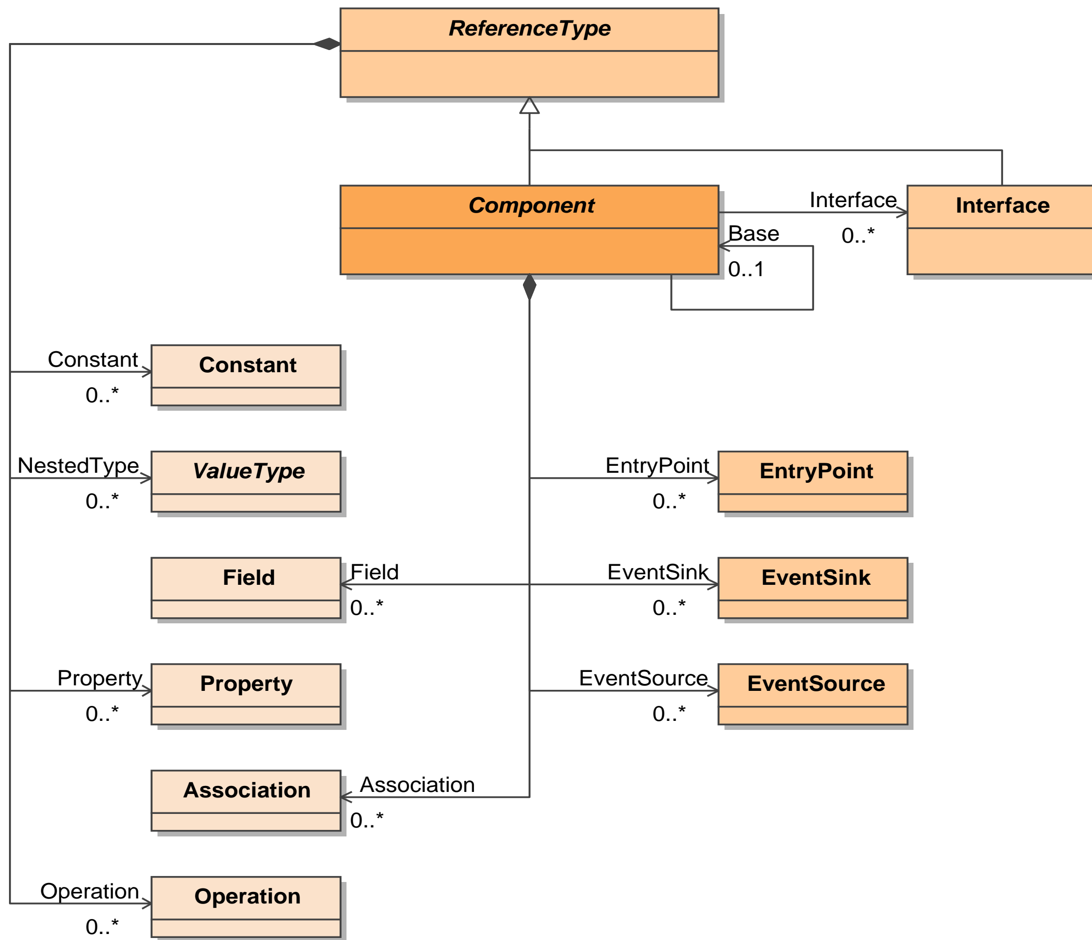

Figure 5-6: Component

5.2.3 Interface

An Interface is a reference type that serves as a contract in a loosely coupled architecture. It has the ability to contain constants, nested types, properties and operations (from ReferenceType). An Interface may inherit from other interfaces (interface inheritance), which is represented via the Base links.

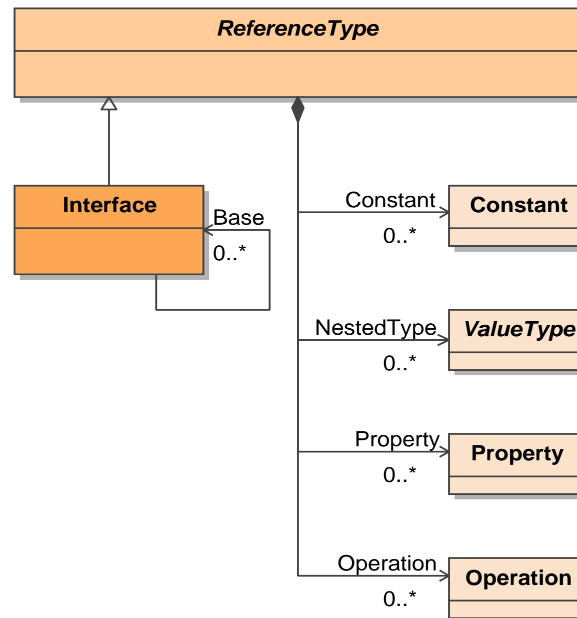

Figure 5-7: Interface

Remark: It is strongly recommended to only use value types, references and other interfaces in the properties and operations of an interface (i.e. not to use models). Otherwise, a dependency between a model implementing the interface, and other models referenced by this interface is introduced, which is against the idea of interface-based or component-based design.

5.2.4 Model

The Model metaclass is a component and hence inherits all component mechanisms.

These mechanisms allow using various different modelling approaches.

For a class-based design, a Model may provide a collection of Field elements to define its internal state. For scheduling and global events, a Model may provide a collection of EntryPoint elements that can be registered with the Scheduler or EventManager services of a Simulation Environment.

For an interface-based design, a Model may provide (i.e. implement) an arbitrary number of interfaces, which is represented via the Interface links.

For a component-based design, a Model may provide Container elements to contain other models (composition), and Reference elements to reference other components (aggregation). These components can either be models or services.

For an event-based design, a Model may support inter-model events via the EventSink and EventSource elements.

For a dataflow-based design, the fields of a Model can be tagged as Input or Output fields.

In addition, a Model may have Association elements to express associations to other models or fields of other models, and it may define its own value types, which is represented by the NestedType elements.

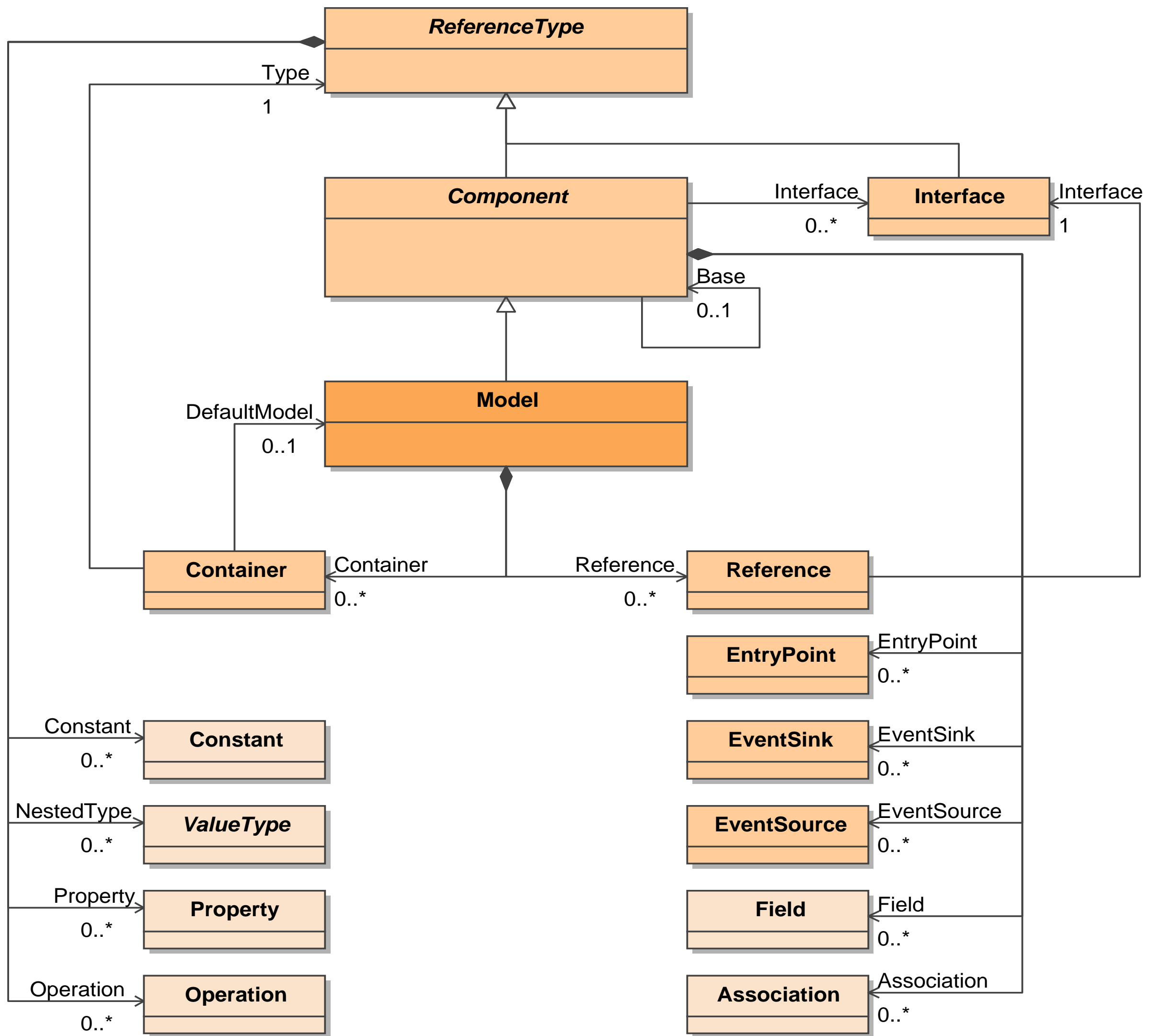

Figure 5-8: Model

Remark: The use of nested types is not recommended, as they may not map to all platform specific models.

5.2.5 Service

The Service metaclass is a component and hence inherits all component mechanisms. A Service can reference one or more interfaces via the Interface links (inherited from Component), where at least one of them must be derived from Smp::IService (see SMP Component Model), which qualifies it as a service interface.

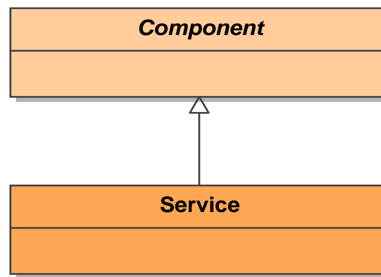

Figure 5-9: Service

5.3 Modelling Mechanisms

This section describes the different modelling mechanisms.

5.3.1 Class based Design

This section describes modelling mechanisms for a class based design.

5.3.1.1 Entry Point

An EntryPoint is a named element of a Component (Model or Service). It corresponds to a void operation taking no parameters that can be called from an external client (e.g. the Scheduler or Event Manager services). An Entry Point can reference both Input fields (which must have their Input attribute set to true) and Output fields (which must have their Output attribute set to true). These links can be used to ensure that all input fields are updated before the entry point is called, or that all output fields can be used after the entry point has been called.

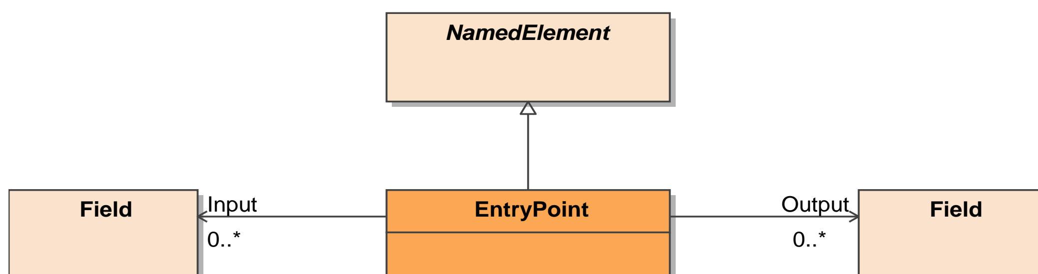

Figure 5-10: Entry Point

5.3.2 Component based Design

This section describes modelling mechanisms for a component based design.

5.3.2.1 Container

A Container defines the rules of composition (containment of children) for a Model.

The type of components that can be contained is specified via the Type link.

The Lower and Upper attributes specify the multiplicity, i.e. the number of possibly stored components. Therein the upper bound may be unlimited, which is represented by Upper=-1.

Furthermore, a container may specify a default implementation of the container type via the DefaultModel link.

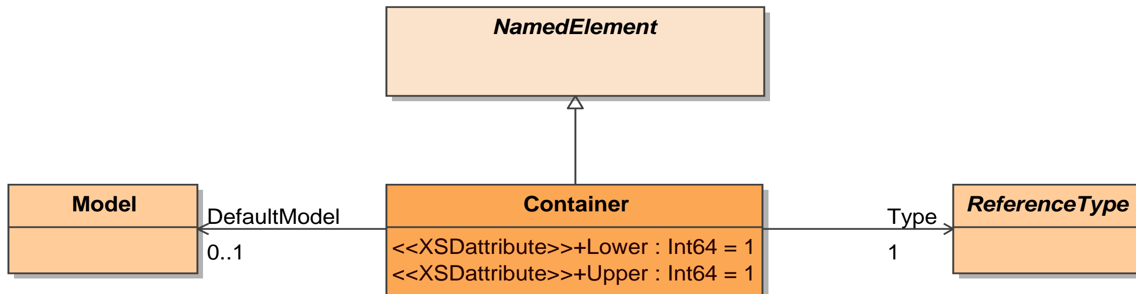

Figure 5-11: Container

Remark: SMDL support tools may use this during instantiation (i.e. creation of an assembly) to select an initial implementation for newly created contained components.

5.3.2.2 Reference

A Reference defines the rules of aggregation (links to components) for a Model.

The type of components (models or services) that can be referenced is specified by the Interface link. Thereby, a service reference is characterized by an interface that is derived from Smp::IService (see SMP Component Model).

The Lower and Upper attributes specify the multiplicity, i.e. the number of possibly held references to components implementing this interface. Therein the upper bound may be unlimited, which is represented by Upper=-1.

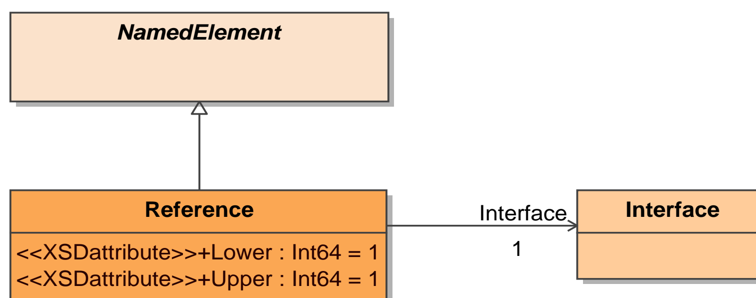

Figure 5-12: Reference

5.3.3 Event based Design

This section describes modelling mechanisms for an event based design.

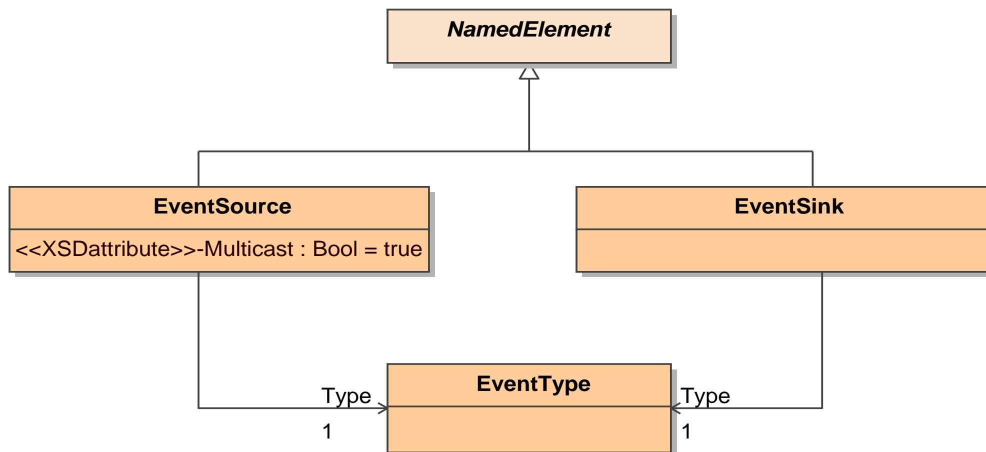

Figure 5-13: Event based Design

5.3.3.1 Event Type

An EventType is used to specify the type of an event. This can be used not only to give a meaningful name to an event type, but also to link it to an existing simple type (via the EventArgs attribute) that is passed as an argument with every invocation of the event.

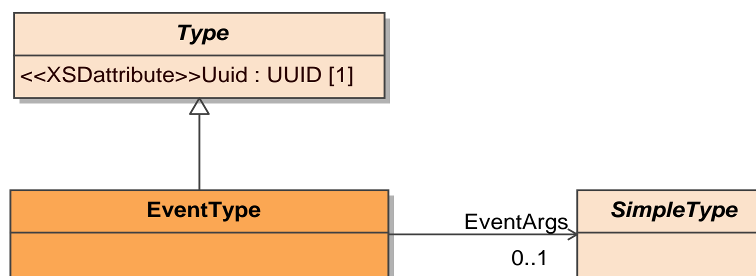

Figure 5-14: Event Type

5.3.3.2 Event Source

An EventSource is used to specify that a Component publishes a specific event under a given name. The Multicast attribute can be used to specify whether any number of sinks can connect to the source (the default), or only a single sink can connect (Multicast=false).

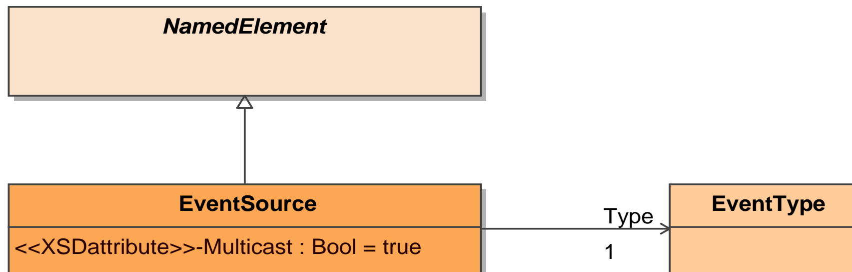

Figure 5-15: Event Source

Remark: An assembly editor can use the Multicast attribute to configure the user interface accordingly to ease the specification of event links.

5.3.3.3 Event Sink

An EventSink is used to specify that a Component can receive a specific event using a given name. An EventSink can be connected to any number of EventSource instances (e.g. in an Assembly using EventLink instances).

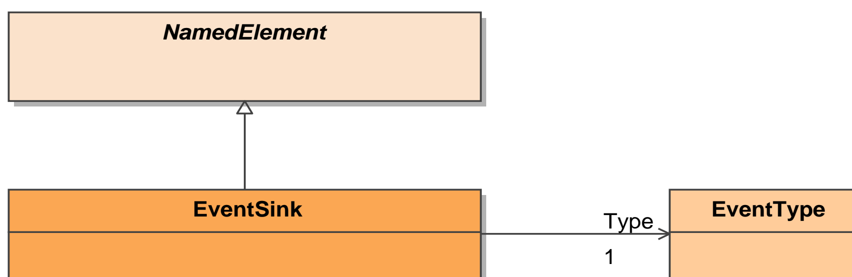

Figure 5-16: Event Sink

5.4 Catalogue Attributes

This section summarises pre-defined SMDL attributes that can be attached to elements in a Catalogue.

5.4.1 Fallible

The Fallible attribute marks a model as being fallible. A fallible model provides support for model failure status, i.e. the model can be assigned one or more failures that can be switched and queried during runtime. The default value for this attribute is false, which corresponds to no support for model failures.

Fallible : AttributeType

Usage = "Model"
AllowMultiple = false
Description = "Mark model as being fallible (enable model failure status support)"
Default = "false"
Id = "Smp.Attributes.Fallible"
Uuid = "8596d697-fb84-41ce-a685-6912006ed660"
Type = Bool

Figure 5-17: Fallible

5.4.2 Forcible

The Forcible attribute marks a field as being forcible. A forcible field provides support for forcing, i.e. the field value can be forced to a specified value during runtime. The following semantics apply for the different kinds of fields:

- Input field: The forced value is provided to the model by the simulation environment. The containing model must internally operate on the field value obtained via the IField.GetValue() method, which ensures that the forced value is obtained when forcing is enabled; otherwise the real value is obtained.
- Ouput field or Input/Output field: The forced value is provided to all connected input fields by the simulation environment (using the IForcibleField interface) according to the Field Links specified in the SMDL Assembly. The containing model must internally operate on the real (unforced) field value.

The default value for this attribute is false, which corresponds to no support for forcing.

Forcible : AttributeType

Usage = "Field"
AllowMultiple = false
Description = "Mark field as being forcible (enable support for forcing the field value)"
Default = "false"
Id = "Smp.Attributes.Forcible"
Uuid = "8596d697-fb84-41ce-a685-6912006ed661"
Type = Bool

Figure 5-18: Forcible

5.4.3 Minimum

The Minimum attribute type defines a duration for an Entry Point specifying the minimum duration between two consecutive calls of the Entry Point. The default value for this attribute is 0, which corresponds to no minimum value.

Minimum : AttributeType
Type = Duration Usage = "EntryPoint" AllowMultiple = false Description = "Minimum time between calls (e.g. time step)" Default = "0" Id = "Smp.Attributes.Minimum" Uuid = "8596d697-fb84-41ce-a685-6912006ed662"

Figure 5-19: Minimum

Remark: This attribute can be used by supporting tools (e.g. a Schedule editor) to check scheduling constraints imposed by the model design.

5.4.4 Maximum

The Maximum attribute type defines a duration for an Entry Point specifying the maximum duration between two consecutive calls of the Entry Point. The default value for this attribute is -1, which corresponds to no maximum value.

Maximum : AttributeType
Type = Duration Usage = "EntryPoint" AllowMultiple = false Description = "Maximum time between calls (e.g. time step)" Default = "-1" Id = "Smp.Attributes.Maximum" Uuid = "8596d697-fb84-41ce-a685-6912006ed663"

Figure 5-20: Maximum

Remark: This attribute can be used by supporting tools (e.g. a Schedule editor) to check scheduling constraints imposed by the model design.

5.4.5 View

The View attribute can be applied to a Field, Property, Operation or Entry Point to specify the element's visibility during publication.

View : AttributeType

Usage = "Field" , "Property", "Operation", "EntryPoint"
AllowMultiple = false
Description = "Control if and how the element is made visible when published to the simulation infrastructure"
Default = "None"
Id = "Smp.Attributes.View"
Uuid = "8596d697-fb84-41ce-a685-6912006ed664"
Type = ViewKind

Figure 5-21: View

5.4.6 View Kind

This enumeration defines possible options for the View attribute, which can be used to control if and how an element is made visible when published to the Simulation Environment.

The Simulation Environment *must* at least support the "None" and the "All" roles (i.e. hidden or always visible).

The Simulation Environment *may* support the selection of different user roles ("Debug", "Expert", "End User"), in which case the "Debug" and the "Expert" role must also be supported as described.

ViewKind

None
Debug
Expert
All

Figure 5-22: View Kind

Table 5-1: Enumeration Literals of ViewKind

Name	Description
None	The element is not made visible to the user (this is the default).
Debug	**Semantics:** The element is made visible for debugging purposes. The element is not visible to end users, neither in the simulation tree nor via scripts. If the simulation infrastructure supports the selection of different user roles, the element shall be visible by "Debug" users only.
Expert	**Semantics:** The element is made visible for expert users. The element is not visible to end users, neither in the simulation tree nor via scripts. If the simulation infrastructure supports the selection of different user roles, the element shall be visible by "Debug" and "Expert" users.
All	**Semantics:** The element is made visible to all users. The element is visible to end users, both in the simulation tree and via scripts. If the simulation infrastructure supports the selection of different user roles, the element shall be visible by all users.

6 Smdl Assembly

This package describes all metamodel elements that are needed in order to define an assembly of model instances. This includes mechanisms for creating links between model instances, namely between references and implemented interfaces, between event sources and event sinks, and between output and input fields.

6.1 Assembly

An assembly contains model instances, where each instance references a Model within a catalogue. Further, an assembly may also contain assembly instances to support sub-assemblies, and service proxies to access services. Each model instance may contain further model instances as children (as defined via the Containers of the corresponding Model), and links to connect references and interfaces (InterfaceLink), event sources and event sinks (EventLink), or output and input fields (FieldLink).

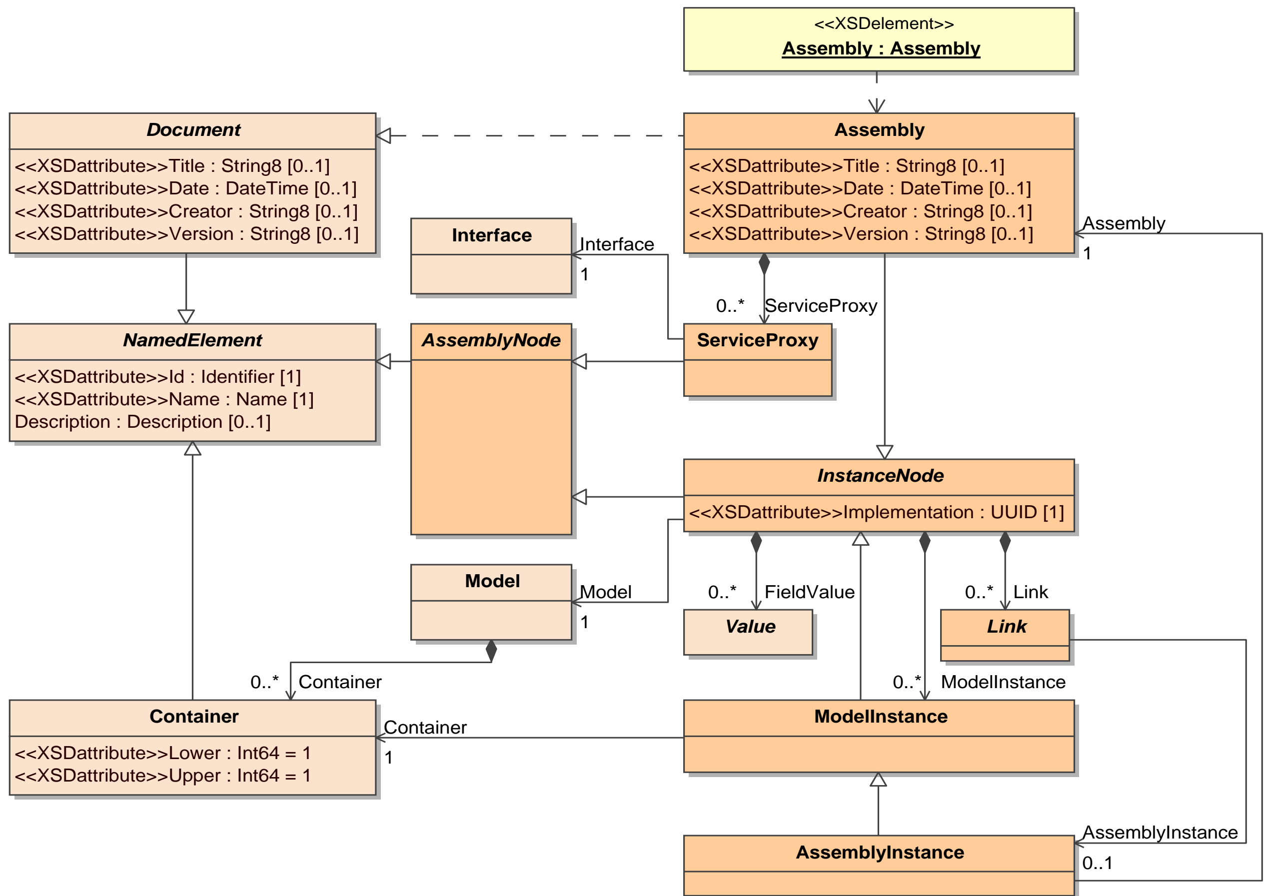

Figure 6-1: Assembly

6.1.1 Assembly Node

The AssemblyNode metaclass is an abstract base class for all nodes in the assembly tree and serves as provider type for interface links. This ensures that the same linking mechanism can be used to resolve model references (via an interface link to an InstanceNode, i.e. Assembly, ModelInstance or AssemblyInstance) and service references (via an interface link to a ServiceProxy).

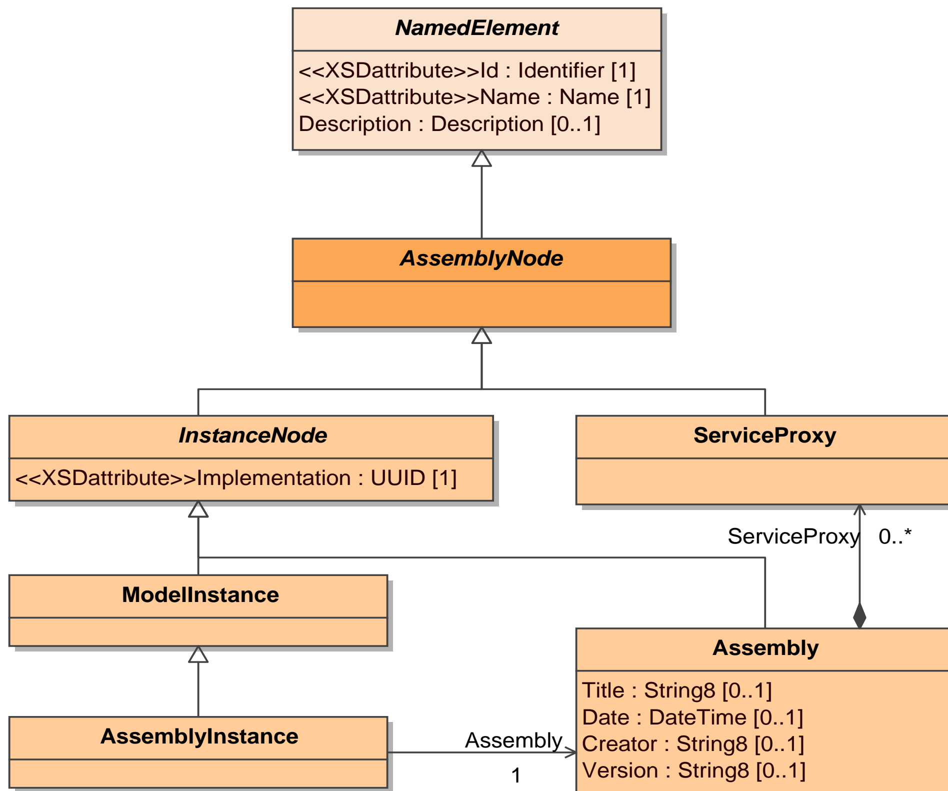

Figure 6-2: Assembly Node

6.1.2 Instance Node

The InstanceNode metaclass is an abstract base class for both Assembly and ModelInstance that provides the common features of both metaclasses.

An InstanceNode represents an instance of a model. Therefore, it has to link to a Model. To allow creating a run-time model instance, the instance node needs to specify as well which Implementation shall be used when the assembly is instantiated (loaded) in a Simulation Environment. This is done by specifying a Universally Unique Identifier (UUID).

Depending on the containers of the referenced model, the instance node can hold any number of child model instances.

For each field of the referenced model, the instance node can specify a FieldValue to define the initial value of the associated Field.

An InstanceNode can link its references, event sinks and input fields, which are specified by the referenced model, to other model instances via the Link elements.

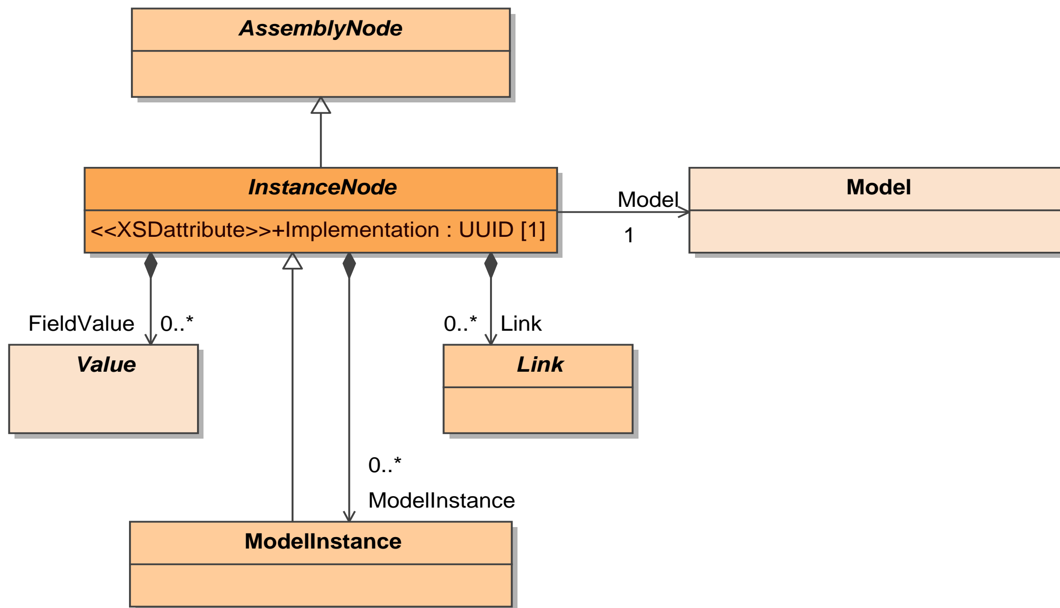

Figure 6-3: Instance Node

6.1.3 Service Proxy

A ServiceProxy represents a service which is specified via the Interface link (service type) and the Name attribute (service name). In the Assembly, a service proxy serves as end point for interface links that resolve service references of a model. When the SMDL Assembly is instantiated (loaded), the Simulation Environment must resolve the service proxy (via ISimulator::GetService()) and provide the resolved service interface to using model instances (i.e. model instances that reference the service proxy via an interface link).

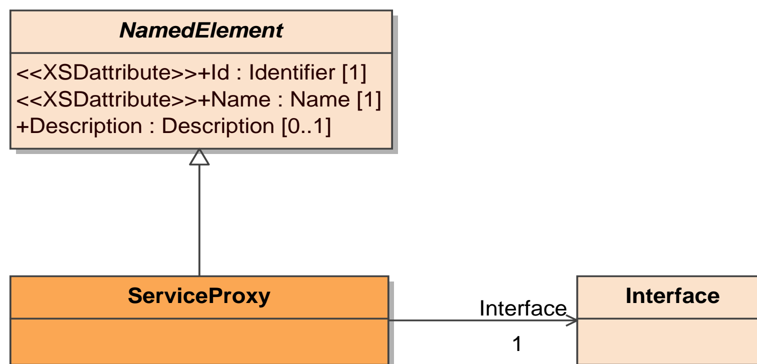

Figure 6-4: Service Proxy

6.1.4 Assembly

An Assembly is an assembly node that acts as the root element (top-level node) in an SMDL Assembly document.

Being an instance node, an Assembly is typed by a Model of a catalogue and can contain model instances, links and field values. Further it can contain

service proxies, for use as end points of service links (interface links to services) within the Assembly.

Being a top-level node, an Assembly does not point to its container (as opposed to a model instance).

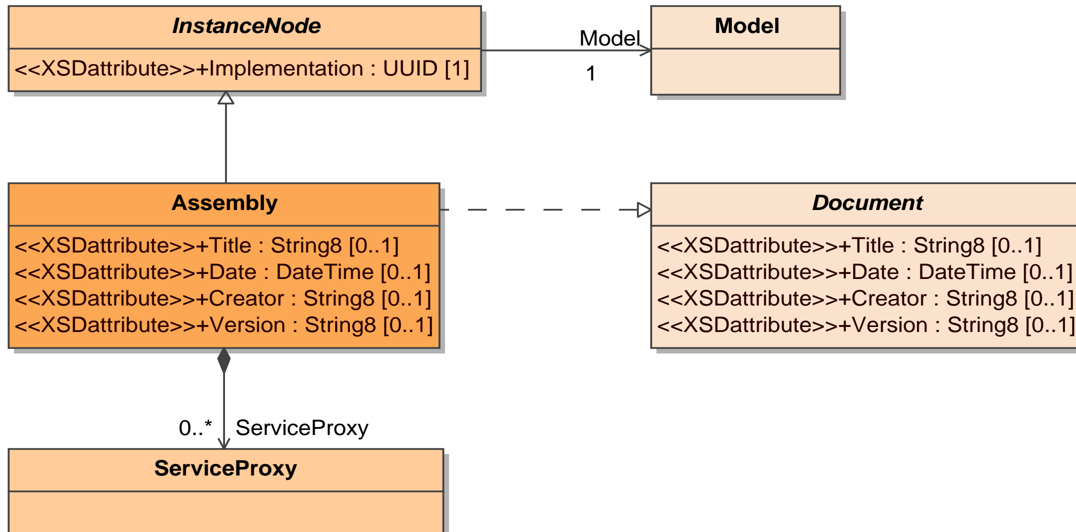

Figure 6-5: Assembly

Remark: As multiple inheritance cannot be used in the metamodel, the Document features are replicated in the Assembly metaclass.

6.1.5 Model Instance

A ModelInstance represents an instance of a model. It is derived from the abstract base metaclass InstanceNode, which provides most of the functionality. As every model instance is either contained in an assembly, or in another model instance, it has to specify as well in which Container of the parent it is stored.

In order to allow an assembly to be interpreted without the associated catalogue, the xlink:title attribute of the Container link shall contain the container name.

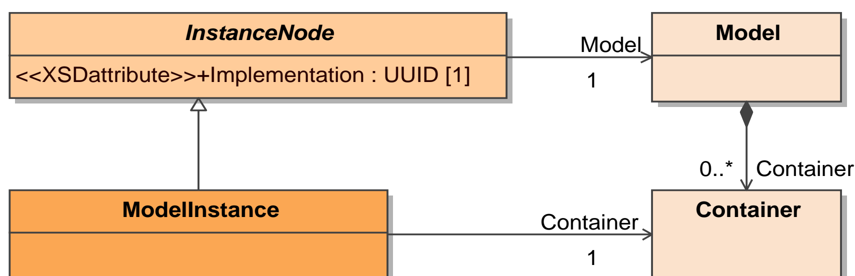

Figure 6-6: Model Instance

6.1.6 Assembly Instance

An AssemblyInstance represents a model instance which is specified by an Assembly in *another* SMDL Assembly document (a sub-assembly).

Semantics and constraints:

- The Assembly link must reference an Assembly node in *another* SMDL Assembly document.
- The FieldValue elements owned by the AssemblyInstance node override any FieldValue elements specified in the referenced Assembly node.
- The Model of both the AssemblyInstance and the referenced Assembly must be identical.
- The Implementation element of an AssemblyInstance node must be identical with the Implementation specified by the referenced Assembly.
- Model instances in the sub-assembly defined by the AssemblyInstance can be referenced from assembly nodes (e.g. model instances) in the using assembly via Links, where the AssemblyInstance link points to this.

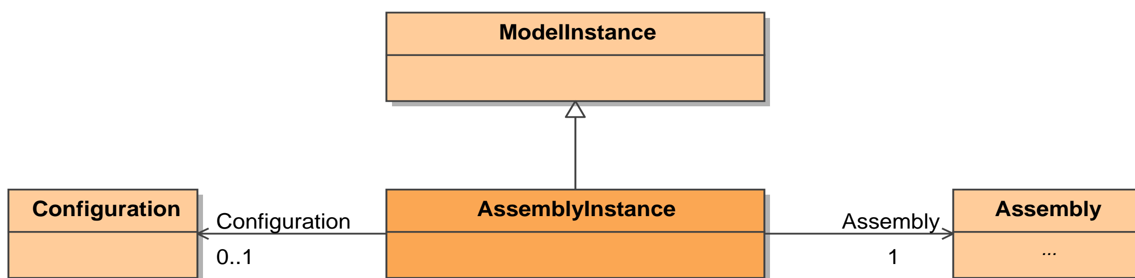

Figure 6-7: Assembly Instance

6.2 Links

Links allow connecting all model instances and the assembly itself together. Four types of links are supported:

- An Interface Link connects a reference to a provided interface of proper type. This is typically used in interface and component-based design.
- An Event Link connects an event sink to an event source of the same type. This is typically used in event-based design.
- A Field Link connects an input field to an output field of the same type. This is typically used in dataflow-based design.

All link metaclasses are derived from a common Link base class.

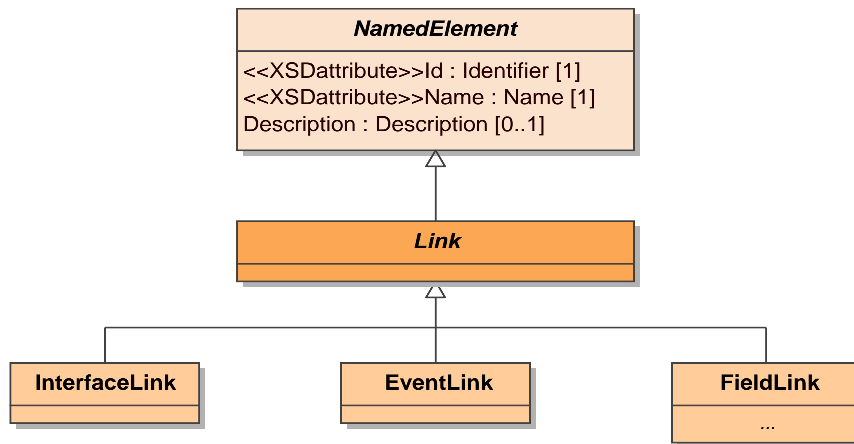

Figure 6-8: Links

6.2.1 Link

The Link metaclass serves as a base metaclass for all links.

The AssemblyInstance link may reference an assembly instance, to support the case where the Link references into a sub-assembly.

If the AssemblyInstance is empty (default), the Link must reference into the containing Assembly, which implicitly defines its assembly instance.

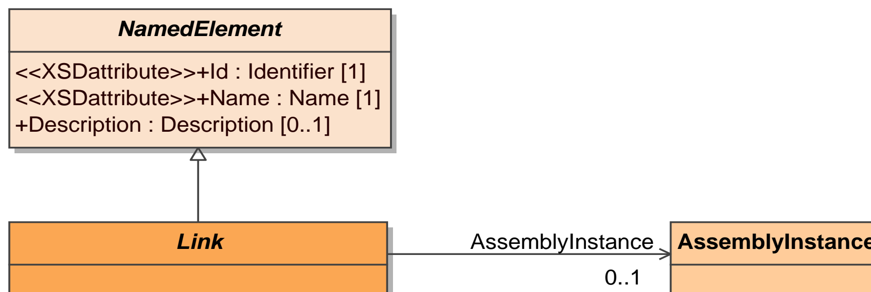

Figure 6-9: Link

6.2.2 Interface Link

An InterfaceLink resolves a reference of an instance node in an assembly.

Therefore, it links to the Reference of the corresponding Model to uniquely identify the link source - together with the knowledge of its parent model instance, which defines the Model.

In order to uniquely identify the link target, the InterfaceLink links to the Provider assembly node, which provides a matching Interface, either via its Model (in case of an InstanceNode) or via its associated service implementation (in case of a ServiceProxy). In the latter case, the Simulation Environment must resolve the service proxy to a service interface when the assembly is instantiated (loaded).

In order to allow an assembly to be interpreted without the associated catalogue, the xlink:title attribute of the Reference link shall contain the name of the associated Reference element in the catalogue.

To be semantically correct, the Interface specified by the Provider must coincide with the Interface that the associated Reference points to, or that is derived from it via interface inheritance.

If the Provider is an instance node, an Interface of the associated Model must match the Reference's Interface.

If the Provider is a service proxy, the Interface of the service proxy must match the Reference's Interface. Note that obviously the associated service implementation must implement such a matching interface, which can only be verified when the Simulation Environment instantiates the assembly (i.e. during runtime) and the service proxy is resolved via its Name using the standard SMP service acquisition mechanism.

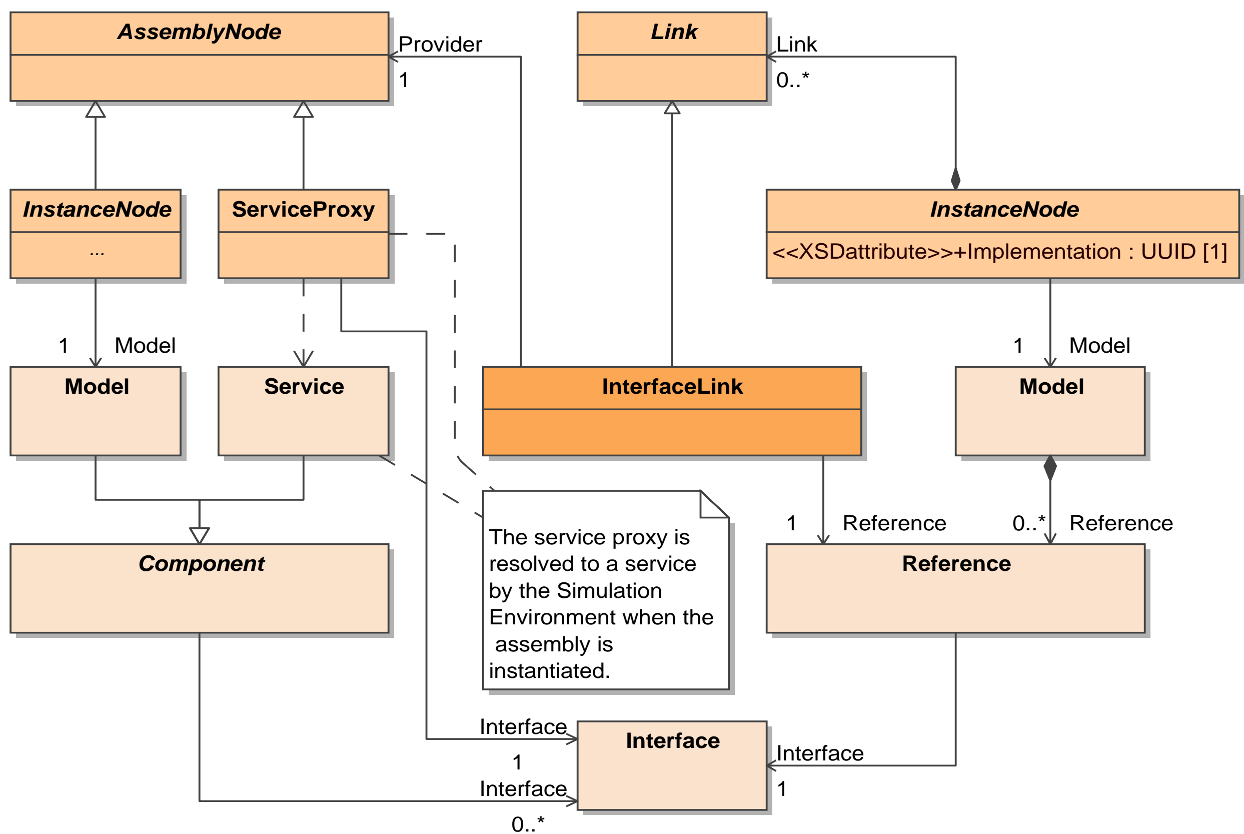

Figure 6-10: Interface Link

6.2.3 Event Link

An EventLink resolves an event sink of an instance node in an assembly. Therefore, it links to the EventSink of the corresponding Model to uniquely identify the link target - together with the knowledge of its parent model instance, which defines the Model.

In order to uniquely identify the link source, the EventLink links to an EventSource of an event Provider model instance.

In order to allow an assembly to be interpreted without the associated catalogue, the xlink:title attribute of the EventSink and EventSource links shall contain the name of the associated EventSink and EventSource elements in the catalogue, respectively.

To be semantically correct, the EventSource of the Provider and the referenced EventSink need to reference the same EventType.

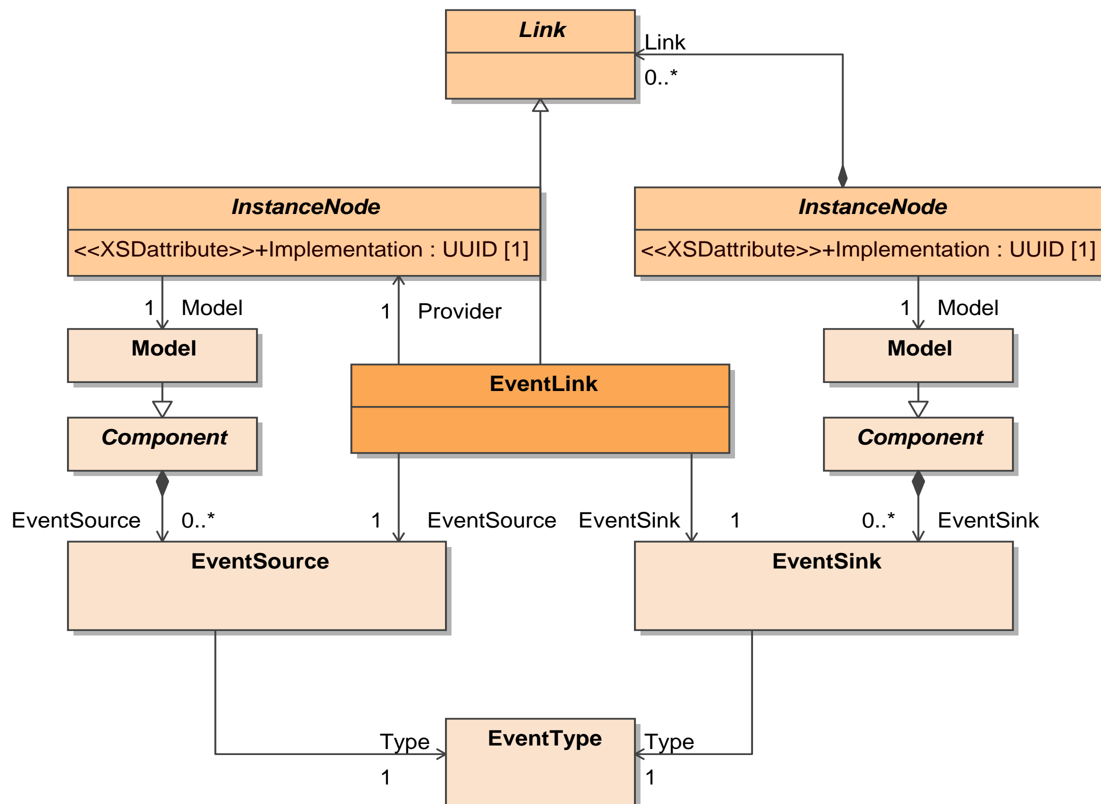

Figure 6-11: Event Link

6.2.4 Field Link

A FieldLink resolves an input field of an instance node in an assembly.

Therefore, it links to the Input of the corresponding Model to uniquely identify the link target - together with the knowledge of its parent model instance, which defines the Model.

In order to uniquely identify the link source, the FieldLink links to an Output field of a Source model instance.

The Factor and Offset attributes specify an affine transformation, which is performed when the field link is "executed" via a Transfer in a Schedule:

- InputValue = OutputValue * Factor + Offset

The default transformation is the identity (Factor=1.0, Offset=0.0).

In order to allow an assembly to be interpreted without the associated catalogue, the xlink:title attribute of the Input and Output links shall contain the name of the associated input and output Field elements in the catalogue, respectively.

To be semantically correct, the Input field needs to have its Input attribute set to true, the Output field of the Source needs to have its Output attribute set to true, and both fields need to reference the same value type.

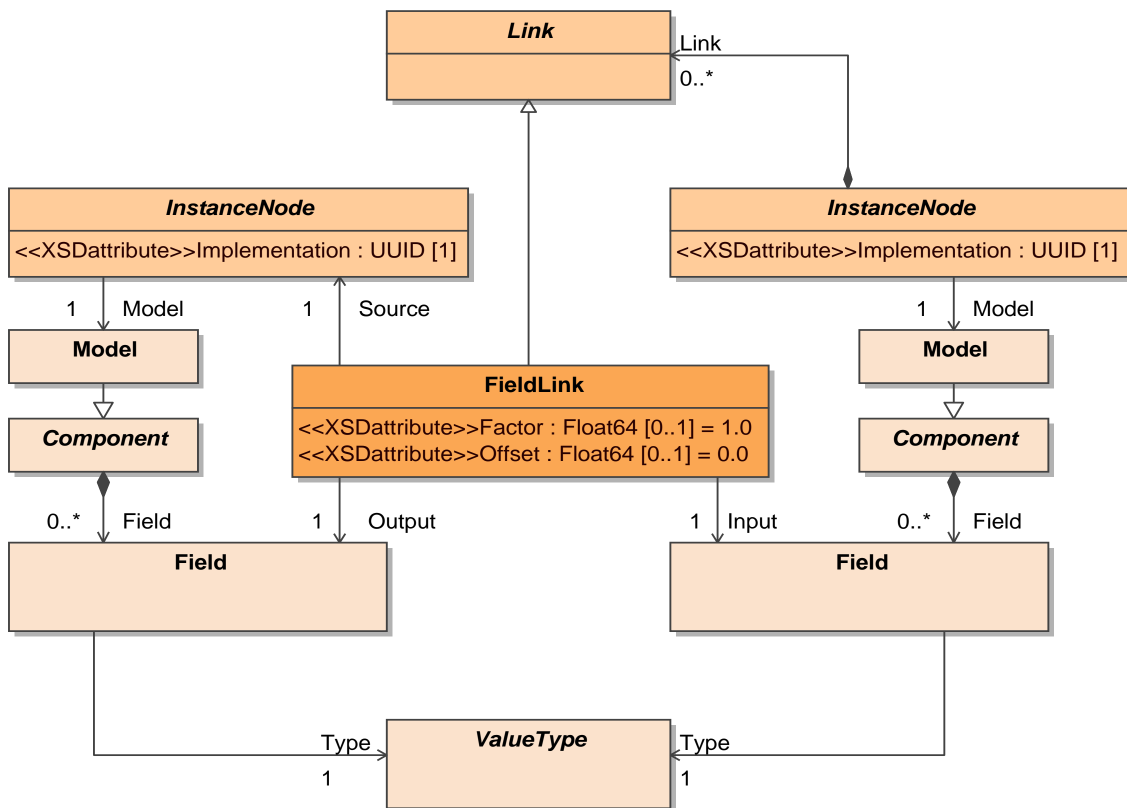

Figure 6-12: Field Link

7 Smdl Schedule

This package describes all metamodel elements that are needed in order to define how the model instances in an assembly are to be scheduled. This includes mechanisms for tasks and events.

7.1 Schedule

The following figure shows the top-level structure of an SMDL Schedule document.

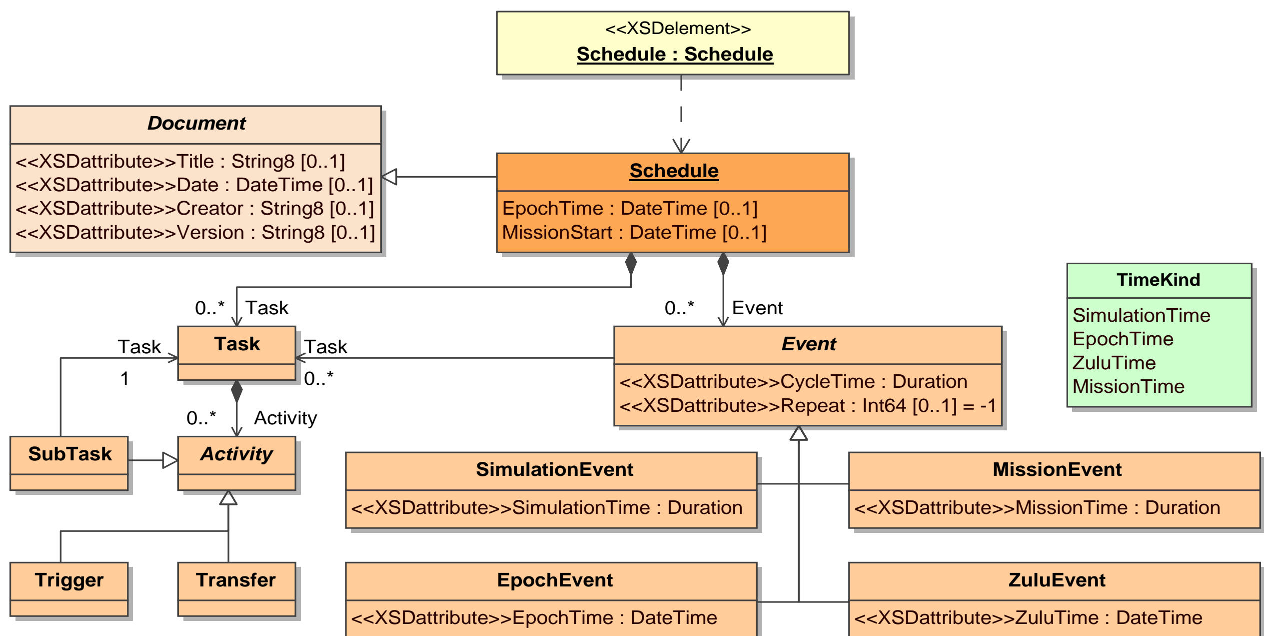

Figure 7-1: Schedule

7.1.1 Schedule

A Schedule is a Document that holds an arbitrary number of tasks (Task elements) and events (Event elements) triggering these tasks. Additionally, it may specify the origins of epoch time and mission time via its EpochTime and MissionStart elements, respectively. These values are used to initialise epoch time and mission time via the SetEpochTime() and SetMissionStart() methods of the TimeKeeper service.

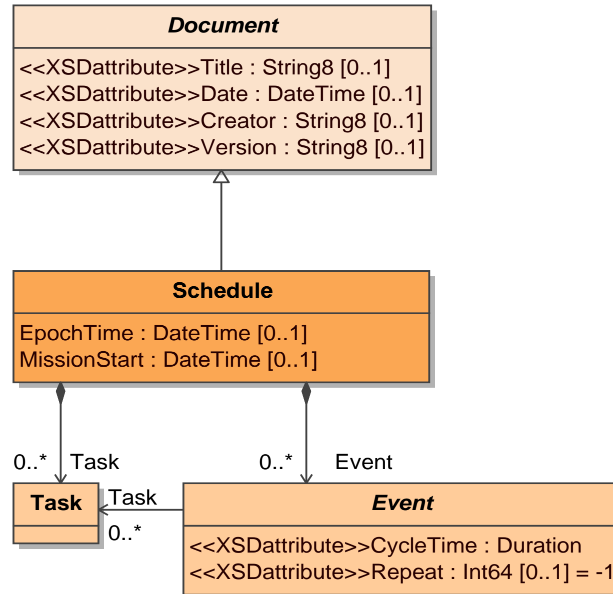

Figure 7-2: Schedule

7.2 Tasks

Tasks are elements that can be triggered by events. The most simple task is one calling a single entry point only, but more complex tasks can call a series of entry points of different providers. In addition, tasks can trigger data transfers of field links in an assembly.

7.2.1 Task

A Task is a container of activities. The order of activities defines in which order the entry points referenced by the Activity elements are called.

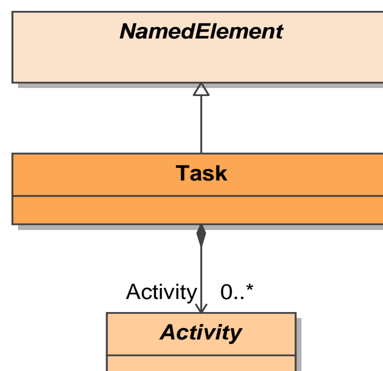

Figure 7-3: Task

7.2.2 Activity

An Activity is the abstract base class for the three different activities that can be contained in a Task.

- A Trigger allows to execute an EntryPoint.
- A Transfer allows to initiate a data transfer as defined in a FieldLink.
- A SubTask allows to execute all activities defined in another Task.

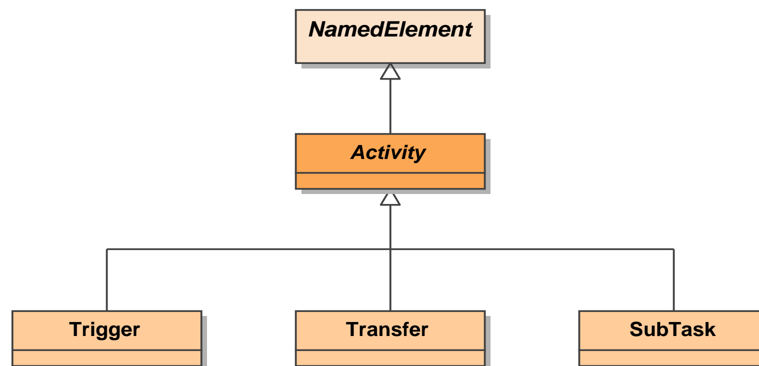

Figure 7-4: Activity

7.2.3 Trigger

A Trigger selects an EntryPoint defined in the corresponding Model of a Catalogue from a Provider instance defined in an Assembly.

In order to allow a schedule to be interpreted without the associated catalogue, the xlink:title attribute of the EntryPoint link shall contain the name of the entry point.

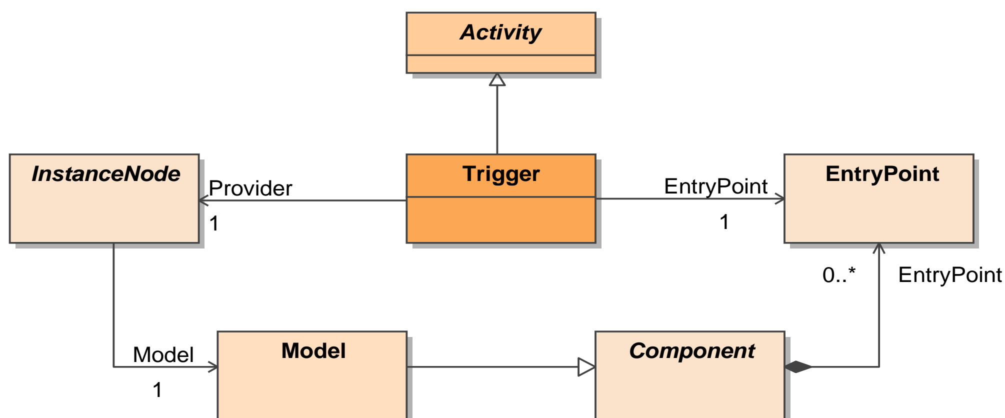

Figure 7-5: Trigger

7.2.4 Transfer

A Transfer selects a FieldLink defined in an Assembly, to initiate its execution to transfer data from the source to the target.

The transformation specified by the Field Link is performed during the transfer of the associated field value.

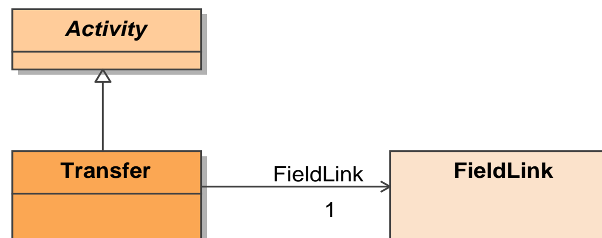

Figure 7-6: Transfer

7.2.5 Sub Task

A SubTask references another Task in a Schedule, to initiate all activities defined in it.

Circular references between tasks are not allowed.

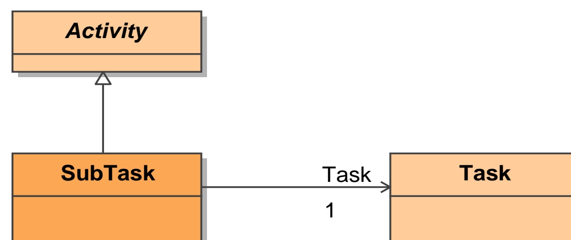

Figure 7-7: Sub Task

7.3 Events

Events are elements in a schedule that can trigger tasks, which themselves reference entry points or field links in assembly nodes. Triggers can either be single events, i.e. they trigger tasks only once, or cyclic events, i.e. they trigger tasks several times. The time when to trigger an event for the first time can be specified in each of the four time kinds supported by SMP. Therefore, four different types of events exist which all derive from a common base class.

7.3.1 Event

An Event defines a point in time when to execute a collection of tasks. Such a scheduled event can either be called only once, or it may be repeated using a given cycle time. While the first type of event is called a single event or timed event, repeated events are called cyclic.

The Repeat attribute defines how often an event shall be repeated.

- An event with Repeat=0 will not be repeated, so it will be only called once. This is a single event.
- An event with Repeat>0 will be called Repeat+1 times, so it is cyclic.
- Finally, a value of Repeat=-1 indicates that an event shall be called forever (unlimited number of repeats). For a cyclic event, the CycleTime needs to specify a positive Duration between two consecutive executions of the event.

Four different time formats are supported to define the time of the (first) execution of the event (see TimeKind). For each of these four time kinds, a dedicated metaclass derived from Event is defined. These are required because relative times (simulation and mission times) are specified using a Duration, while absolute times (epoch and Zulu times) are specified using a DateTime.

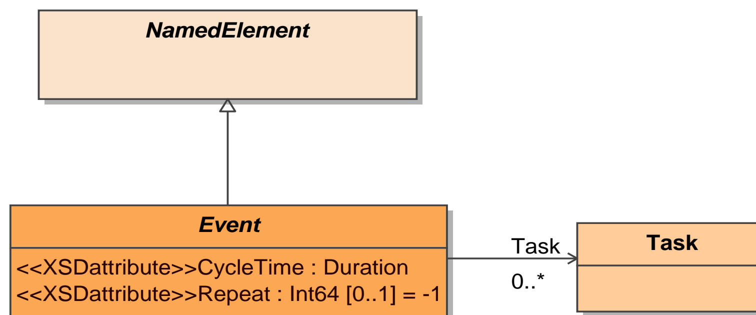

Figure 7-8: Event

7.3.2 Time Kind

This enumeration defines the four SMP time formats.

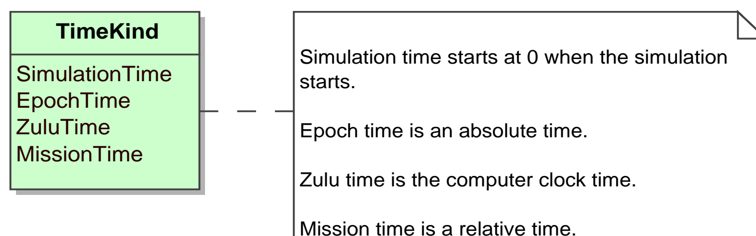

Figure 7-9: Time Kind

Table 7-1: Enumeration Literals of TimeKind

Name	Description
SimulationTime	Simulation Time starts at 0 when the simulation is kicked off. It progresses while the simulation is in Executing state.
EpochTime	Epoch Time is an absolute time and typically progresses with simulation time. The offset between epoch time and simulation time may get changed during a simulation.
ZuluTime	Zulu Time is the computer clock time, also called wall clock time. It progresses whether the simulation is in Executing or Standby state, and is not necessarily related to simulation, epoch or mission time.
MissionTime	Mission Time is a relative time (duration from some start time) and progresses with epoch time.

7.3.3 Simulation Event

A SimulationEvent is derived from Event and adds a SimulationTime attribute.

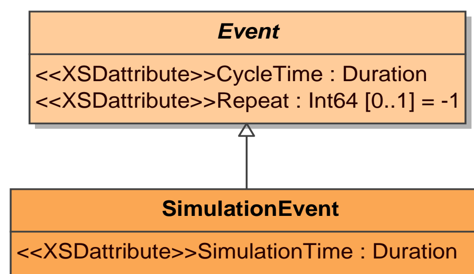

Figure 7-10: Simulation Event

7.3.4 Epoch Event

An EpochEvent is derived from Event and adds an EpochTime attribute.

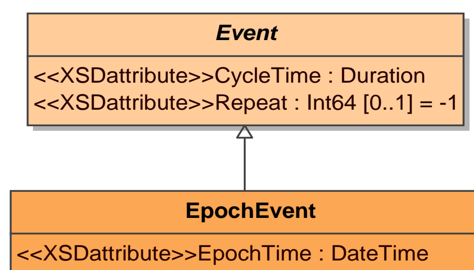

Figure 7-11: Epoch Event

7.3.5 Mission Event

A MissionEvent is derived from Event and adds a MissionTime attribute.

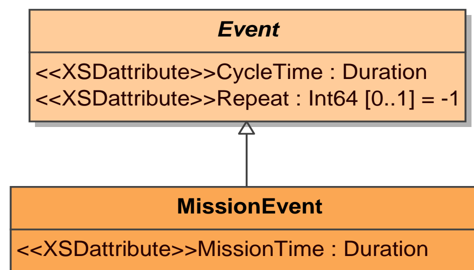

Figure 7-12: Mission Event

7.3.6 Zulu Event

A ZuluEvent is derived from Event and adds a ZuluTime attribute.

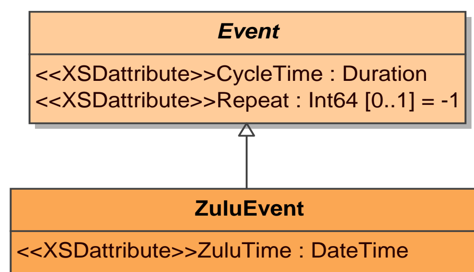

Figure 7-13: Zulu Event

7.4 Schedule Attributes

This section summarises pre-defined SMDL attributes that can be attached to elements in a Schedule.

7.4.1 Initialisation

This attribute can be applied to a Task to specify that it is an initialisation task.

When a schedule file with one or more initialisation tasks is loaded, these tasks are added to the scheduler as initialisation entry points (via IScheduler::AddInitEntryPoint()) and therefore executed in the simulator's Initialising state.

Initialisation : AttributeType
Type = Bool Usage = "Task" AllowMultiple = false Description = "Task is an Initialisation Task" Default = "false" Id = "Smp.Attributes.Initialisation" Uuid = "8596d697-fb84-41ce-a685-6912006ed680"

Figure 7-14: Initialisation

7.4.2 Field Update

This attribute can either be applied to a Trigger to specify the update behaviour for associated fields when the entry point is executed, to a Schedule in order to define the global default behaviour within the schedule or to a Model Instance to specify the execution of the entry points in terms of the referenced input and out fields.

When the default behaviour ("None") is selected, field values are *not* updated automatically when the entry point is executed. In this case, all field updates must be explicitly scheduled via Transfer elements.

When the "Pull" behaviour is selected, all input fields associated with the entry point are updated from the linked outputs *before* the entry point is called.

When the "Push" behaviour is selected, the values of all output fields associated with the entry point are automatically transferred to their linked input fields (in other models) *after* the entry point has been called.

When tbe "PullPush" behaviour is selected, the behaviour is a combination of "Pull" and "Push".

FieldUpdate : AttributeType
Type = FieldUpdateKind Usage = "Trigger", "Schedule", "ModelInstance" AllowMultiple = false Description = "Specifies the field update behaviour" Default = "None" Id = "Smp.Attributes.FieldUpdate" Uuid = "8596d697-fb84-41ce-a685-6912006ed681"

Figure 7-15: Field Update

7.4.3 Field Update Kind

This enumeration allows to specify the behaviour when a Trigger is updated.

Figure 7-16: Field Update Kind

Table 7-2: Enumeration Literals of FieldUpdateKind

Name	Description
None	Field values are not updated automatically when the entry point is executed. In this case, all field updates must be explicitly scheduled via Transfer elements.
Pull	All input fields associated with the entry point are updated from the linked outputs *before* the entry point is called.
Push	The values of all output fields associated with the entry point are automatically transferred to their linked input fields (in other models) *after* the entry point has been called.
PullPush	Combination of Pull and Push. That is, all input fields associated with the entry point are updated from the linked outputs *before* the entry point is called and the values of all output fields associated with the entry point are automatically transferred to their linked input fields (in other models) *after* the entry point has been called.

8 Smdl Package

This package describes all metamodel elements that are needed in order to define how implementations of types defined in catalogues are packaged. This includes not only models, which may have different implementations in different packages, but as well all other user-defined types.

8.1 Package

The following figure shows the top-level structure of an SMDL Package document.

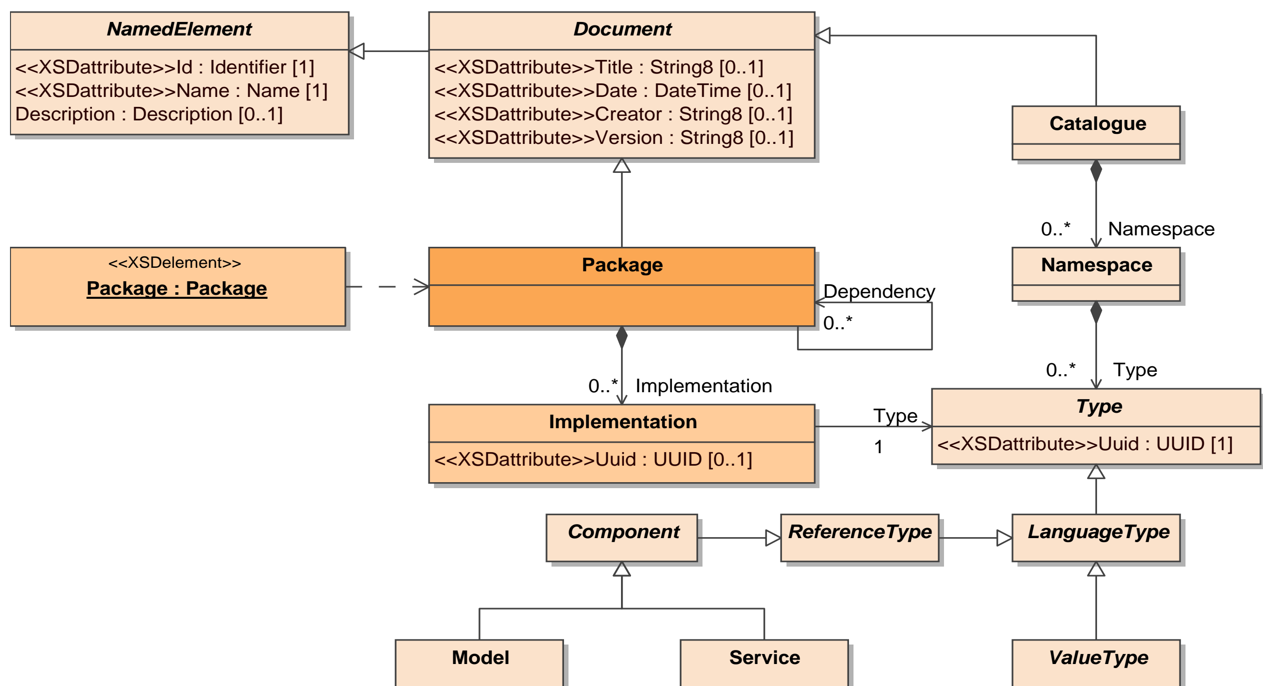

Figure 8-1: Package

8.1.1 Implementation

An Implementation selects a single Type from a catalogue for a package. For the implementation, the Uuid of the type is used, unless the type is a Model: For a

model, a different Uuid for the implementation can be specified, as for a model, different implementations may exist in different packages.

The purpose of the Implementation metaclass is to make a selected type available (as part of a package) for use in a simulation. The detailed meaning is specific to each platform, and detailed in the individual platform mappings.

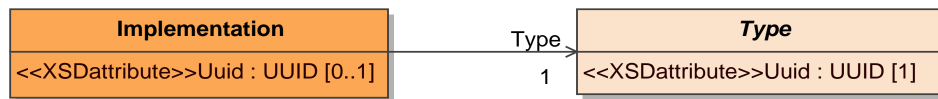

Figure 8-2: Implementation

Remark: As an implementation is not a named element, is has neither a name nor a description. Nevertheless, it can be shown in a tool using name and description of the type it references.

8.1.2 Package

A Package is a Document that holds an arbitrary number of Implementation elements. Each of these implementations references a type in a catalogue that shall be implemented in the package.

In addition, a package may reference other packages as a Dependency. This indicates that a type referenced from an implementation in the package requires a type implemented in the referenced package.

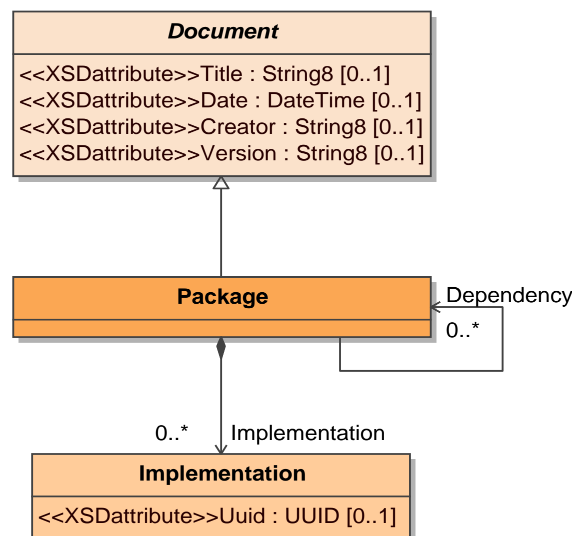

Figure 8-3: Package

9
Smdl Configuration

This package describes all metamodel elements that are needed in order to define an SMDL configuration document. A configuration document allows specifying arbitrary field values of component instances in the simulation hierarchy. This can be used to initialise or reinitialise the simulation.

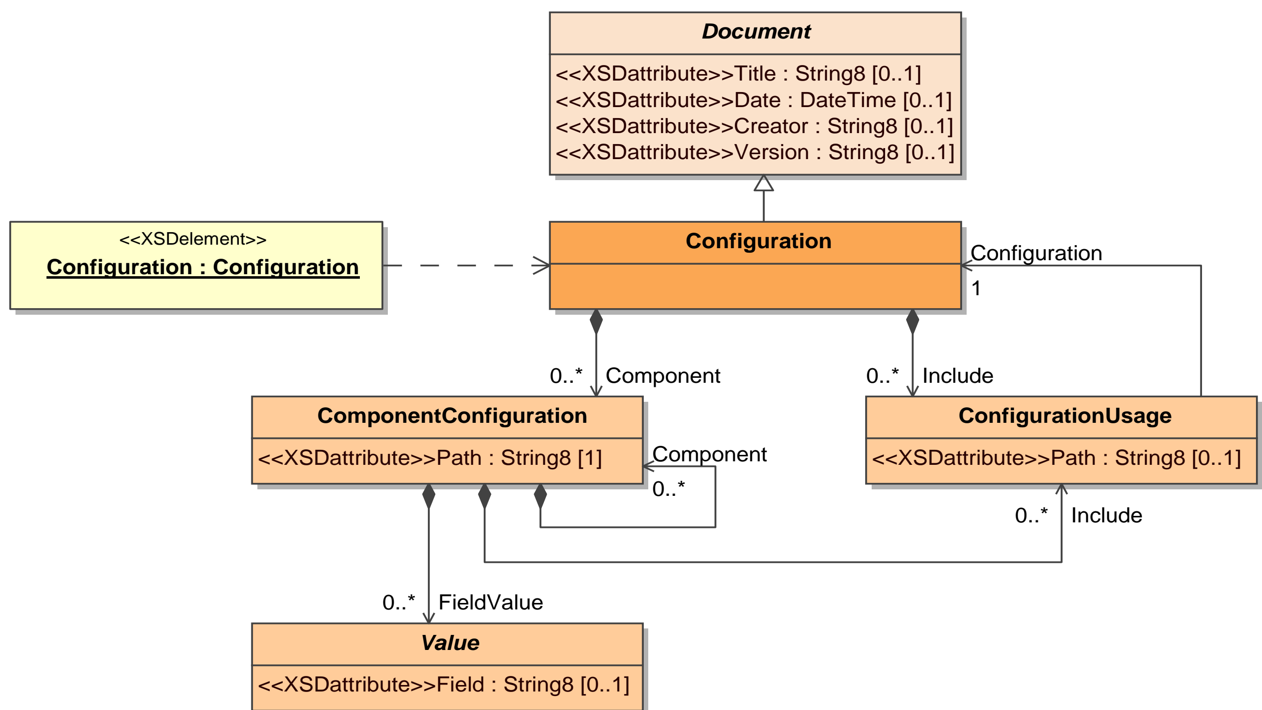

Figure 9-1: Configuration

9.1 Component Configuration

A ComponentConfiguration defines field values of a component instance (model or service instance). The component instance is specified via the Path attribute, which can be

- absolute, i.e. the path starts at the root of the simulation tree (e.g. Path="/Models/Spacecraft/AOCS" or Path="/Services/CustomService"), or
- relative, i.e. the path is appended to the path of the parent component configuration.

Each FieldValue is an instance of one of the available Value metaclasses (section 4.4). A FieldValue has to reference the corresponding field of the component via its Field attribute which specifies the field's local name/path (e.g. Field="field1" or Field="struct1.structField2").

In addition to the ability to define a hierarchy of component configurations via the Component element, the Include element enables to include another Configuration file using a relative or absolute Path for it.

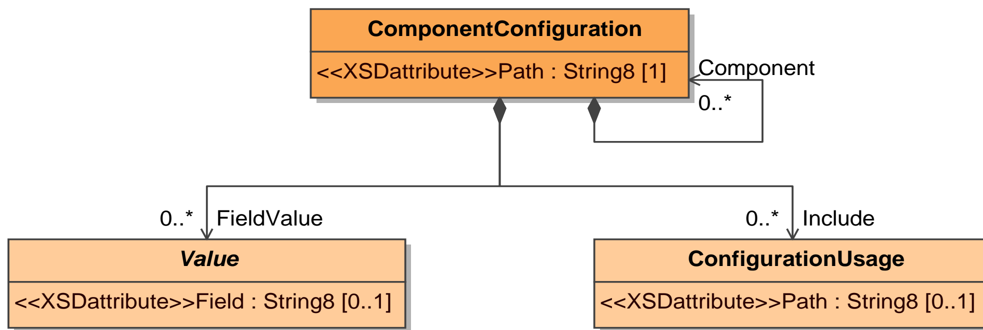

Figure 9-2: Component Configuration

9.2 Configuration

A Configuration acts as the root element (top-level node) in an SMDL Configuration document. A configuration contains a tree of component configurations that define field values of component instances (model or service instances). Further, a configuration may include other configuration documents via the Include element.

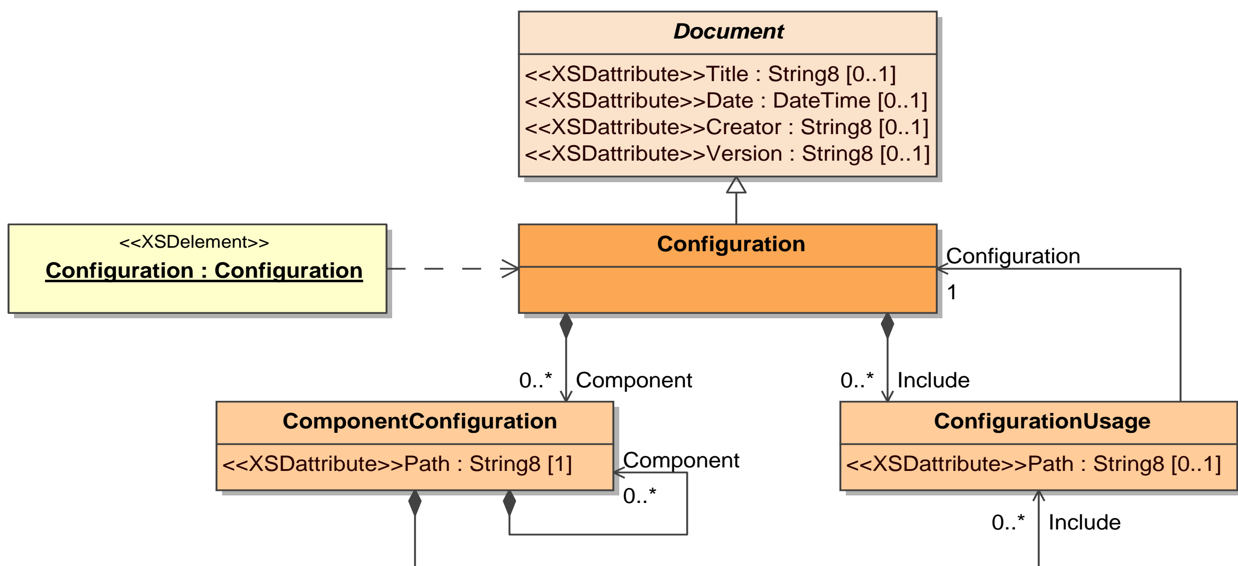

Figure 9-3: Configuration

9.3 Configuration Usage

A ConfigurationUsage allows to include another SMDL Configuration document. The external configuration document is referenced via the Configuration link. The Path specifies the (optional) prefix that shall be applied to all paths in the external configuration document. This allows pre-defined configuration documents to be re-used for the configuration of components in various places in the model hierarchy.

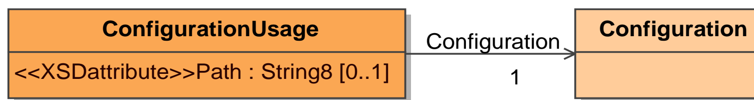

Figure 9-4: Configuration Usage

10 Validation Rules

This section contains validation rules for SMDL files (which are XML files) that go beyond the validation that can be done with the corresponding XML Schema documents (listed in the Annex). as these validation rules cover semantic rules that cannot be expressed in XML Schema. The rules included in this draft have been provided by the SMP2 1.2 implementations of EuroSim and SIMSAT, and don't necessary match the ECSS SMP standard. Further, the list of rules is not complete.

The rules in this section need to be reviewed, completed and merged into a single list. This is expected to be performed during the public review of this technical memorandum.

10.1 EuroSim Validation Rules

No recursive types shall be specified, i.e. no types used as part of a type X may have a type that is equal to X, is a type that uses X as a part of it, or is derived from X.

- Array itemType / Array.
- Class Base / Class
- Model Base / Model
- Field Type / Field's containing type (Model or Structure)
- Interface Base / Interface

Types that are used in another type shall be visible for that type.

- EventArgs / EventType
- Array itemType / Array.
- Class Base / Class
- Model Base / Model
- Field Type / Field's containing type (Model or Structure)
- Interface Base / Interface
- Property Type / Property's containing type (ReferenceType)
- Operation Type / Operation's containing type (Reference type)
- EventSource Type / EventSource's containing type (Model)

- EventSink Type / EventSink's containing type (Model)

A NamedElement Name shall be unique in its context:

- Enumeration Literal / Enumeration
- Field / Structure
- Parameter / Operation
- Namespace / Namespace's containing type (Namespace or Catalogue)
- Namespace + Type / containing Namespace
- Property + Association + Operation + Field / Class, including Base Class.
- Field + EntryPoint + EventSink + NestedType + Association + EventSource + Reference + Container + Operation + Property / Model, including Base Models
- Operation + Property / Interface, including Base Interfaces
- Array size shall be > 0.
- A NamedElement Id shall be unique in its Document.
- A NamedElement Name shall not be a C++ keyword.
- Container lower bound shall be > 0.
- Container lower bound shall be <= container upper bound, if present.
- Container upper bound shall be -1 or >= container lower bound.
- Suffix for catalogue file name shall be "cat".
- Suffix for assembly file name shall be "asm".
- Suffix for schedule file name shall be "sch".
- EnumerationLiteral Names shall be unique within an Enumeration.
- EnumerationLiteral Values shall be unique within an Enumeration.
- EntryPoint Minimum shall be >= 0.0.
- EntryPoint Minimum shall be <= EntryPoint Maximum, if present.
- EnytrPoint Maximum shall be -1.0 or >= EntryPoint Minimum.
- EntryPoint Output fields shall be output type fields.
- EntryPoint Input fields shall be input type fields.
- EntryPoint Input and Output fields shall be located in the same Model or a base model.
- Float Minimum shall be < Float Maximum if MinInclusive is false or MaxInclusive is false.
- Float Minimum shall be <= Float Maximum if MinInclusive is true and MaxInclusive is true.
- Integer Minimum shall be >= Integer Maximum.
- Property AttachedField shall have the type of Property's Type, or a type derived thereof.

- Property AttachedField shall be located in the same Class or a base class.
- Reference lower bound shall be > 0.
- Reference lower bound shall be <= Reference upper bound, if present.
- Reference upper bound shall be -1 or >= Reference lower bound.
- String Length shall be >= 0.
- Type UUID shall be unique.
- XLinks in documents shall not result in recursively linked documents.

10.2 SIMSAT Validation Rules

10.2.1 General Validation Rules

Table 10-1 General Validation rules

ID	Rule Description	Validation Level
G001	The Id must not be empty.	FAILURE
G002	The Name must start with a letter.	FAILURE
G003	The Name must only contain letters, digits and the underscore.	FAILURE
G004	The Name must not exceed 32 characters in length.	FAILURE

10.2.2 Catalogue Validation Rules

Table 10-2: Catalogue Validation Rules

ID	Rule Description	Validation Level
C101	The VisibilityKind 'package' is not supported in all platform mappings.	Recommendation
C102	The Uuid must not be empty.	FAILURE
C103	There must be a ValueType object assigned to the Type.	FAILURE
C104	An enumeration must contain at least one enumeration literal.	FAILURE
C105	The PrimitiveType for an Integer may only point to Int8, Int16, Int32, Int64, UInt8, UInt16, UInt32.	FAILURE
C106	For an Integer, the minimum should not be bigger than the maximum.	Warning
C107	The PrimitiveType for a Float may only point to Float32, Float64.	FAILURE
C108	For a Float, the minimum should not be bigger than the maximum.	Warning
C109	The Size of an array must be a positive number.	FAILURE

ID	Rule Description	Validation Level
C110	The Length of a String must be a positive number.	FAILURE
C111	The Type link for a Field must point to a ValueType.	FAILURE
C112	A Field should have a default value.	Recommendation
C113	A Field of a Structure must be public.	FAILURE
C114	A Field of a Model should not be public	Recommendation
C115	The Type link of an Operation may only point to a LanguageType.	FAILURE
C116	An operation of an Interface must be public.	FAILURE
C117	An Operation of an Interface must not be static.	FAILURE
C118	The Type link of a Parameter must point to a LanguageType.	FAILURE
C119	A Parameter can only have a default value if its Type is a ValueType.	FAILURE
C120	The value is not in the range defined for the corresponding Integer type.	Warning OR FAILURE
C121	The value is not in the range defined for the corresponding Float type.	Warning OR FAILURE
C122	The number of ItemValues in an ArrayValue should match the size of the corresponding Array type.	Warning
C123	The length of a StringValue must not exceed the size of the corresponding String type.	Warning
C124	A StructureValue should have a FieldValue for each Field of the corresponding Structure type.	Warning
C125	The Field link must point to a Field.	FAILURE
C126	The Field value must not be empty.	FAILURE
C127	The type for an AttributeType must point to a ValueType.	FAILURE
C128	The default value of an AttributeType must be not empty.	FAILURE
C129	The Type link for an Attribute must point to an AttributeType.	FAILURE
C130	The Value of an Attribute must not be empty.	FAILURE
C131	The Type link of a Property must point to a LanguageType.	FAILURE
C132	The Type of the AttachedField must match the Type of the Property.	FAILURE
C133	A Property of an Interface must be public.	FAILURE
C134	A Property of an Interface must not be static.	FAILURE
C135	The Type link of an Association must point to a LanguageType.	FAILURE
C136	Use of NestedTypes is not recommended.	Recommendation
C137	The Type link of a Container must point to a ReferenceType.	FAILURE
C138	For a Container, the lower limit should not be bigger than the upper limit.	Warning

ID	Rule Description	Validation Level
C139	The Type link of a Reference must point to an Interface.	FAILURE
C140	For a Reference, the Lower limit should not be bigger than the Upper limit.	Warning
C141	The EventArgs link for an EventType must only point to a SimpleType.	FAILURE
C142	The Type link of an EventSource must point to an EventType.	FAILURE
C143	The Type link of an EventSink must point to an EventType.	FAILURE

10.2.3 Assembly Validation Rules

Table 10-3: Assembly Validation rules

ID	Rule Description	Validation Level
A201	The Model link must point to a Model.	FAILURE
A202	Each Container must contain at least 'Container.Lower' models and must contain at most 'Container.Upper' models	Warning
A203		
A204	Each Reference must have at least 'Reference.Lower' links and must have at most 'Reference.Upper' links	Warning
A205		
A206	The Container link must point to a Container.	FAILURE
A207	The Reference link must point to a Reference.	FAILURE
A208	The Provider link must point to an AssemblyNode.	FAILURE
A209	The EventSink link must point to an EventSink.	FAILURE
A210	The Publisher link must point to an AssemblyNode.	FAILURE
A211	The EventSource link must point to an EventSource.	FAILURE
A212	If the multicast option is turned off, one EventSink at most may be connected to the EventSource.	Warning

10.2.4 Schedule Validation Rules

Table 10-4: Schedule Validation Rules

ID	Rule Description	Validation Level
S301	An event properly references a task	FAILURE
S302	A trigger properly references an entrypoint in the catalogue associated with the model instance in the assembly	FAILURE
S303	A transfer properly references a fieldlink in an assembly	FAILURE
S304	A subtask properly references a task	FAILURE
S305	Sensible duration and datetime values are set in events (eg. after Schedule.MissionStart and compatible with Schedule.EpochTime)	FAILURE

10.2.5 Package Validation Rules

Table 10-5: Package Validation Rules

ID	Rule Description	Validation Level
P401	Only Types from referenced Catalogue files allowed.	FAILURE
P402	No Type name clashes.	FAILURE
P403	For a Model, a different Uuid for the implementation must be specified.	FAILURE
P404		

Annex A
XML Schema Documents

This annex contains the normative XML Schema documents that define the Simulation Model Definition Language.

A.1 Core Elements

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:Elements="http://www.esa.int/2008/02/Core/Elements"
            xmlns:xlink="http://www.w3.org/1999/xlink"
            xmlns="http://www.esa.int/2008/02/Core/Elements"
            targetNamespace="http://www.esa.int/2008/02/Core/Elements"
            elementFormDefault="unqualified"
            attributeFormDefault="unqualified">
   <xsd:annotation>
      <xsd:documentation>
This file is generated by the EGOS-MF XML Schema Generation Tool, version 1.0.1.

UML Model Information:
    UML model file:  file:/F:/VST/ECSSMP/design/gen/xmi/Core.xmi
    UML model name:  Core
    UML metamodel:   http://schema.omg.org/spec/UML/2.1.1
    XMI version:     2.1
    XMI exporter:    EGOS-MF XMI Converter (from MagicDraw UML 12.1), version 1.0.1

XSLT Processing Information:
    Processing date: 2008-03-12T09:24:33.014+01:00
    XSLT processor:  SAXON 8.8 from Saxonica
```

```
    XSLT version:    2.0
    XSLT stylesheet: xmi-to-xsd.xslt
      </xsd:documentation>
   </xsd:annotation>
   <!-- Import UML Component 'xlink' -->
   <xsd:import namespace="http://www.w3.org/1999/xlink" schemaLocation="../xlink.xsd"/>

   <!-- ======================================== -->
   <!-- UML Package 'Elements' -->
   <!-- ======================================== -->
   <!-- This package defines base metaclassesnand annotation mechanisms used throughout the SMP Metamodel. -->

   <!-- ======================================== -->
   <!-- UML Package 'Elements::BasicTypes' -->
   <!-- ======================================== -->
   <!-- The Core Elements schema defines some basic types which are used for attributes of other types. Some string types use a
pattern tag to specify a regular expression that limits their value space. -->

   <!-- UML DataType 'Identifier' -->
   <xsd:simpleType name="Identifier">
      <xsd:annotation>
         <xsd:documentation>An Identifier is a machine-readable identification string for model elements stored in XML documents,
being a possible target for XML links. This type is used in the Id attribute of the NamedElement metaclass (see below).
An identifier must not be empty, which is indicated by the minLength tag.</xsd:documentation>
         <xsd:documentation>Identifier is defined as an XML ID type. It must be unique in the space of identified things.
Identifier must be used as an attribute because it is of the xsd:ID type, and this must exist only in attributes, never elements,
to retain compatibility with XML 1.0 DTD's.</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:ID">
         <xsd:minLength value="1"/>
      </xsd:restriction>
   </xsd:simpleType>

   <!-- UML DataType 'Name' -->
   <xsd:simpleType name="Name">
      <xsd:annotation>
         <xsd:documentation>A Name is a user-readable name for model elements. This type is used in the Name attribute of the
NamedElement metaclass (see below). A name must start with a character, and is limited to characters, digits, and the underscore
('_').
</xsd:documentation>
         <xsd:documentation>&lt;b&gt;&lt;i&gt;Remark&lt;/b&gt;&lt;/i&gt;: It is recommended to limit names to a maximum of 32
characters.</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:string">
```

```
        <xsd:pattern value="[a-zA-Z][a-zA-Z0-9_]*"/>
      </xsd:restriction>
   </xsd:simpleType>

   <!-- UML DataType 'Description' -->
   <xsd:simpleType name="Description">
      <xsd:annotation>
         <xsd:documentation>A Description holds a description for a model element. This type is used in the Description element
of the NamedElement metaclass (see below).</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:string"/>
   </xsd:simpleType>

   <!-- UML DataType 'UUID' -->
   <xsd:simpleType name="UUID">
      <xsd:annotation>
         <xsd:documentation>A UUID is Universally Unique Identifier (UUID) for model elements, which takes the form of
hexadecimal integers separated by hyphens, following the pattern 8-4-4-4-12 as defined by the Open Group. This type is used in
the Uuid attribute of the Type metaclass.</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:string">
         <xsd:pattern value="[0-9a-fA-F]{8}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{4}-[0-9a-fA-F]{12}"/>
      </xsd:restriction>
   </xsd:simpleType>

   <!-- ======================================== -->
   <!-- UML Package 'Elements::Elements' -->
   <!-- ======================================== -->
   <!-- The Core Elements schema defines the common base class NamedElement and the Document class which is the base for all SMP
documents. -->

   <!-- UML Class 'NamedElement' -->
   <xsd:complexType name="NamedElement" abstract="true">
      <xsd:annotation>
         <xsd:documentation>The metaclass NamedElement is the common base for most other language elements. A named element has
an Id attribute for unique identification in an XML file, a Name attribute holding a human-readable name to be used in
applications, and a Description element holding a human-readable description. Furthermore, a named element can hold an arbitrary
number of metadata children.</xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
         <xsd:element name="Description" type="Elements:Description" minOccurs="0">
            <xsd:annotation>
               <xsd:documentation>The description of the element.</xsd:documentation>
            </xsd:annotation>
```

```
        </xsd:element>
        <xsd:element name="Metadata" type="Elements:Metadata" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="Id" type="Elements:Identifier" use="required">
        <xsd:annotation>
          <xsd:documentation>The unique identifier of the named element.

This is typically a machine-readable identification of the element that can be used for referencing the
element.</xsd:documentation>
        </xsd:annotation>
      </xsd:attribute>
      <xsd:attribute name="Name" type="Elements:Name" use="required">
        <xsd:annotation>
          <xsd:documentation>The name of the named element that is suitable for use in programming languages such as C++, Java,
or CORBA IDL.

This is the element's name represented with only a limited character set specified by the Name type.</xsd:documentation>
        </xsd:annotation>
      </xsd:attribute>
    </xsd:complexType>

    <!-- UML Class 'Document' -->
    <xsd:complexType name="Document" abstract="true">
      <xsd:annotation>
        <xsd:documentation>A Document is a named element that can be the root element of an XML document. It therefore adds the
Title, Date, Creator and Version attributes to allow identification of documents.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
        <xsd:extension base="Elements:NamedElement">
          <xsd:attribute name="Title" type="xsd:string" use="optional"/>
          <xsd:attribute name="Date" type="xsd:dateTime" use="optional"/>
          <xsd:attribute name="Creator" type="xsd:string" use="optional"/>
          <xsd:attribute name="Version" type="xsd:string" use="optional"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- ======================================== -->
    <!-- UML Package 'Elements::Metadata' -->
    <!-- ======================================== -->
    <!-- Metadata is additional, named information stored with a named element. It is used to further annotate named elements, as
the Description element is typically not sufficient.
```

```
Metadata can either be a simple Comment, a link to external Documentation, or an Attribute. Please note that the Attribute
metaclass is shown on this diagram already, but not defined in the Core Elements schema. It is added by the Core Types schema
later, because attributes are typed by an attribute type. -->

   <!-- UML Class 'Metadata' -->
   <xsd:complexType name="Metadata" abstract="true">
      <xsd:annotation>
         <xsd:documentation>Metadata is additional, named information stored with a named element. It is used to further annotate
named elements, as the Description element is typically not sufficient.

</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Elements:NamedElement"/>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Comment' -->
   <xsd:complexType name="Comment">
      <xsd:annotation>
         <xsd:documentation>A Comment element holds user comments, e.g. for reviewing models. The Name of a comment should allow
to reference the comment (e.g. contain the author's initials and a unique number), while the comment itself is stored in the
Description.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Elements:Metadata"/>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Documentation' -->
   <xsd:complexType name="Documentation">
      <xsd:annotation>
         <xsd:documentation>A Documentation element holds additional documentation, possibly together with links to external
resources. This is done via the Resource element (e.g. holding links to external documentation, 3d animations, technical
drawings, CAD models, etc.), based on the XML linking language.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Elements:Metadata">
            <xsd:sequence>
               <xsd:element name="Resource" type="xlink:SimpleLink" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
```

```
</xsd:schema>
```

A.2 Core Types

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:Types="http://www.esa.int/2008/02/Core/Types"
            xmlns:Elements="http://www.esa.int/2008/02/Core/Elements"
            xmlns:xlink="http://www.w3.org/1999/xlink"
            xmlns="http://www.esa.int/2008/02/Core/Types"
            targetNamespace="http://www.esa.int/2008/02/Core/Types"
            elementFormDefault="unqualified"
            attributeFormDefault="unqualified">
   <xsd:annotation>
      <xsd:documentation>
This file is generated by the EGOS-MF XML Schema Generation Tool, version 1.0.1.

UML Model Information:
    UML model file:  file:/F:/VST/ECSSMP/design/gen/xmi/Core.xmi
    UML model name:  Core
    UML metamodel:   http://schema.omg.org/spec/UML/2.1.1
    XMI version:     2.1
    XMI exporter:    EGOS-MF XMI Converter (from MagicDraw UML 12.1), version 1.0.1

XSLT Processing Information:
    Processing date: 2008-03-12T09:24:33.014+01:00
    XSLT processor:  SAXON 8.8 from Saxonica
    XSLT version:    2.0
    XSLT stylesheet: xmi-to-xsd.xslt
      </xsd:documentation>
   </xsd:annotation>
   <!-- Import UML Component 'Core::Elements' -->
   <xsd:import namespace="http://www.esa.int/2008/02/Core/Elements"
               schemaLocation="../Core/Elements.xsd"/>
   <!-- Import UML Component 'xlink' -->
   <xsd:import namespace="http://www.w3.org/1999/xlink" schemaLocation="../xlink.xsd"/>

   <!-- ======================================== -->
   <!-- UML Package 'Types' -->
   <!-- ======================================== -->
   <!-- This package provides basic types and typing mechanisms, together with appropriate value specification mechanisms. -->
```

```
   <!-- ======================================== -->
   <!-- UML Package 'Types::Types' -->
   <!-- ======================================== -->
   <!-- Types are used in different contexts. The most common type is a LanguageType, but typing is used as well for other 
mechanisms, e.g. for Attributes (and later for Events). While this schema introduces basic types, more advanced types used 
specifically within SMDL Catalogues are detailed later. -->

   <!-- UML Enumeration 'VisibilityKind' -->
   <xsd:simpleType name="VisibilityKind">
      <xsd:annotation>
         <xsd:documentation>This enumeration defines the possible values for an element's visibility.</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:string">
         <xsd:enumeration value="private">
            <xsd:annotation>
               <xsd:documentation>The element is visible only within its containing classifier.</xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
         <xsd:enumeration value="protected">
            <xsd:annotation>
               <xsd:documentation>The element is visible within its containing classifier and derived classifiers 
thereof.</xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
         <xsd:enumeration value="package">
            <xsd:annotation>
               <xsd:documentation>The element is globally visible inside the package.</xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
         <xsd:enumeration value="public">
            <xsd:annotation>
               <xsd:documentation>The element is globally visible.</xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
      </xsd:restriction>
   </xsd:simpleType>

   <!-- UML Class 'VisibilityElement' -->
   <xsd:complexType name="VisibilityElement" abstract="true">
      <xsd:annotation>
         <xsd:documentation>A VisibilityElement is a named element that can be assigned a Visibility attribute to limit its scope 
of visibility. The visibility may be global (public), local to the parent (private), local to the parent and derived types 
thereof (protected), or package global (package).</xsd:documentation>
      </xsd:annotation>
```

```
      <xsd:complexContent>
         <xsd:extension base="Elements:NamedElement">
            <xsd:attribute name="Visibility" type="Types:VisibilityKind" use="optional" default="private"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Type' -->
   <xsd:complexType name="Type" abstract="true">
      <xsd:annotation>
         <xsd:documentation>A Type is the abstract base metaclass for all type definition constructs specified by SMDL. A type
must have a Uuid attribute representing a Universally Unique Identifier (UUID) as defined above. This is needed such that
implementations may reference back to their specification without the need to directly reference an XML element in the
catalogue.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:VisibilityElement">
            <xsd:attribute name="Uuid" type="Elements:UUID" use="required"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'LanguageType' -->
   <xsd:complexType name="LanguageType" abstract="true">
      <xsd:annotation>
         <xsd:documentation>A LanguageType is the abstract base meta0u108 ?ass for value types (where instances are defined by
their value), and references to value types. Also the Smdl Catalogue schema defines reference types (where instances are defined
by their reference, i.e. their location in memory) which are derived from LanguageType as well.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:Type"/>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'ValueType' -->
   <xsd:complexType name="ValueType" abstract="true">
      <xsd:annotation>
         <xsd:documentation>An instance of a ValueType is uniquely determined by its value. Two instances of a value type are
said to be equal if they have equal values. Value types include simple types like enumerations and integers, and composite types
like structures and arrays.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:LanguageType"/>
      </xsd:complexContent>
```

```
   </xsd:complexType>

   <!-- UML Class 'ValueReference' -->
   <xsd:complexType name="ValueReference">
      <xsd:annotation>
         <xsd:documentation>A ValueReference is a type that references a specific value type. It is the "missing link" between
value types and reference types.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:LanguageType">
            <xsd:sequence>
               <xsd:element name="Type" type="xlink:SimpleLink">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Types:ValueType</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Types:ValueType</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'NativeType' -->
   <xsd:complexType name="NativeType">
      <xsd:annotation>
         <xsd:documentation>A NativeType specifies a type with any number of platform mappings. It is used to anchor existing or
user-defined types into different target platforms. This mechanism is used within the specification to define the SMDL primitive
types with respect to the Metamodel, but it can also be used to define native types within an arbitrary SMDL catalogue for use by
models. In the latter case, native types are typically used to bind a model to some external library or existing Application
Programming Interface (API).</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:LanguageType">
            <xsd:sequence>
               <xsd:element name="Platform" type="Types:PlatformMapping" minOccurs="0"
                            maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'PlatformMapping' -->
   <xsd:complexType name="PlatformMapping">
      <xsd:annotation>
```

```
       <xsd:documentation>&lt;html&gt;
&lt;head&gt;

&lt;/head&gt;
&lt;body&gt;
  &lt;p&gt;
    A PlatformMapping defines the mapping of a native type into a target
    platform. The Name attribute specifies the platform name (see below),
    the Type attribute specifies the type name on the platform, the
    Namespace attribute specifies the type's namespace (if any) on the
    target platform, and the Location attribute specifies where the type is
    located. Note that the interpretation of these values is platform
    specific.
  &lt;/p&gt;
  &lt;p&gt;
    The platform name shall be specified using the pattern &lt;language&gt;&lt;environment&gt;,
    where the environment is optional and may further detail the platform.
    Some examples are:
  &lt;/p&gt;
  &lt;ul&gt;
    &lt;li&gt;
      cpp: Standard ANSI/ISO C++ (for all environments)
    &lt;/li&gt;
    &lt;li&gt;
      cpp__linux__: C++ under Linux Operating System environment
    &lt;/li&gt;
    &lt;li&gt;
      idl: CORBA IDL
    &lt;/li&gt;
    &lt;li&gt;
      xsd: XML Schema
    &lt;/li&gt;
    &lt;li&gt;
      java: Java language
    &lt;/li&gt;
  &lt;/ul&gt;
  &lt;p&gt;
    Basically, any platform mapping may be specified in SMDL as long as the
    tools &amp;#8211; typically code generators working on SMDL Catalogue(s) &amp;#8211; have an
    understanding of their meaning.&lt;br&gt;&lt;br&gt;The interpretation of the
  &lt;/p&gt;
  &lt;p&gt;
    &lt;environment&gt;
```

```
    &lt;/p&gt;
    &lt;p&gt;
      string may vary between different platforms, and is detailed in each
      platform mapping document.
    &lt;/p&gt;
  &lt;/body&gt;
&lt;/html&gt;
</xsd:documentation>
      </xsd:annotation>
      <xsd:attribute name="Name" type="xsd:string" use="required">
         <xsd:annotation>
            <xsd:documentation>Name of the platform using the following pattern:
&lt;language&gt;_&lt;environment&gt;, where the environment may be split into &lt;os&gt;_&lt;compiler&gt;. Examples are:
cpp_windows_vc71 - C++ using Microsoft VC++ 7.1 under Windows
cpp_linux_gcc33  - C++ using GNU gcc 3.3 under Linux</xsd:documentation>
         </xsd:annotation>
      </xsd:attribute>
      <xsd:attribute name="Type" type="xsd:string" use="required">
         <xsd:annotation>
            <xsd:documentation>Name of the type on the platform.</xsd:documentation>
         </xsd:annotation>
      </xsd:attribute>
      <xsd:attribute name="Namespace" type="xsd:string" use="optional">
         <xsd:annotation>
            <xsd:documentation>Namespace on the platform.</xsd:documentation>
         </xsd:annotation>
      </xsd:attribute>
      <xsd:attribute name="Location" type="xsd:string" use="optional">
         <xsd:annotation>
            <xsd:documentation>Location on the platform.
- In C++, this may be a required include file.
- In Java, this may be a jar file to reference.
- In C#, this may be an assembly to reference.</xsd:documentation>
         </xsd:annotation>
      </xsd:attribute>
   </xsd:complexType>

   <!-- ======================================== -->
   <!-- UML Package 'Types::ValueTypes' -->
   <!-- ======================================== -->
   <!-- This package provides mechanisms to specify value types. The shown metaclasses are not the value types themselves, but
rather represent language elements (i.e. mechanisms) that can be applied to define actual value types. Please note that the
PrimitiveType metaclass has been introduced to allow defining the available base types of SMDL, and is only used internally.
```

```
Further, the NativeType metaclass provides a generic mechanism to specify native platform specific types, which is also used to
define the platform mappings of the SMP primitive types within the Component Model. -->

   <!-- UML Class 'SimpleType' -->
   <xsd:complexType name="SimpleType" abstract="true">
      <xsd:annotation>
         <xsd:documentation>A simple type is a type that can be represented by a simple value. Simple types include primitive
types as well as user-defined Enumeration, Integer, Float and String types.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:ValueType"/>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'PrimitiveType' -->
   <xsd:complexType name="PrimitiveType">
      <xsd:annotation>
         <xsd:documentation>A number of pre-defined types are needed in order to bootstrap the type system. These pre-defined
value types are represented by instances of the metaclass PrimitiveType.
This mechanism is only used in order to bootstrap the type system and may not be used to define new types for modelling. This is
an important restriction, as all values of primitive types may be held in a SimpleValue. The metaclasses derived from
SimpleValue, however, are pre-defined and cannot be extended.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:SimpleType"/>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Enumeration' -->
   <xsd:complexType name="Enumeration">
      <xsd:annotation>
         <xsd:documentation>An Enumeration type represents one of a number of pre-defined enumeration literals. The Enumeration
language element can be used to create user-defined enumeration types. An enumeration must always contain at least one
EnumerationLiteral, each having a name and an integer Value attached to it.
All enumeration literals of an enumeration type must have unique names and values, respectively.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:SimpleType">
            <xsd:sequence>
               <xsd:element name="Literal" type="Types:EnumerationLiteral" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
```

```
   <!-- UML Class 'EnumerationLiteral' -->
   <xsd:complexType name="EnumerationLiteral">
      <xsd:annotation>
         <xsd:documentation>An EnumerationLiteral assigns a Name (inherited from NamedElement) to an integer
Value.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Elements:NamedElement">
            <xsd:attribute name="Value" type="xsd:int" use="required"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Integer' -->
   <xsd:complexType name="Integer">
      <xsd:annotation>
         <xsd:documentation>An Integer type represents integer values with a given range of valid values (via the Minimum and
Maximum attributes). The Unit element can hold a physical unit that can be used by applications to ensure physical unit integrity
across models.
Optionally, the PrimitiveType used to encode the integer value may be specified (one of Int8, Int16, Int32, Int64, UIn8, UInt16,
UInt32, where the default is Int32).</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:SimpleType">
            <xsd:sequence>
               <xsd:element name="PrimitiveType" type="xlink:SimpleLink" minOccurs="0">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Types:PrimitiveType</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Types:PrimitiveType</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
            <xsd:attribute name="Minimum" type="xsd:long" use="optional"/>
            <xsd:attribute name="Maximum" type="xsd:long" use="optional"/>
            <xsd:attribute name="Unit" type="xsd:string" use="optional"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Float' -->
   <xsd:complexType name="Float">
      <xsd:annotation>
```

```
        <xsd:documentation>A Float type represents floating-point values with a given range of valid values (via the Minimum and
Maximum attributes). The MinInclusive and MaxInclusive attributes determine whether the boundaries are included in the range or
not. The Unit element can hold a physical unit that can be used by applications to ensure physical unit integrity across models.
Optionally, the PrimitiveType used to encode the floating-point value may be specified (one of Float32 or Float64, where the
default is Float64).</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
        <xsd:extension base="Types:SimpleType">
          <xsd:sequence>
            <xsd:element name="PrimitiveType" type="xlink:SimpleLink" minOccurs="0">
              <xsd:annotation>
                <xsd:documentation>Link destination type: Types:PrimitiveType</xsd:documentation>
                <xsd:appinfo source="http://www.w3.org/1999/xlink">Types:PrimitiveType</xsd:appinfo>
              </xsd:annotation>
            </xsd:element>
          </xsd:sequence>
          <xsd:attribute name="Maximum" type="xsd:double" use="optional"/>
          <xsd:attribute name="MinInclusive" type="xsd:boolean" use="optional" default="true"/>
          <xsd:attribute name="Minimum" type="xsd:double" use="optional"/>
          <xsd:attribute name="MaxInclusive" type="xsd:boolean" use="optional" default="true"/>
          <xsd:attribute name="Unit" type="xsd:string" use="optional"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- UML Class 'String' -->
    <xsd:complexType name="String">
      <xsd:annotation>
        <xsd:documentation>A String type represents fixed Length string values base on Char8. The String language element
defines an Array of Char8 values, but allows a more natural handling of it, e.g. by storing a string value as one string, not as
an array of individual characters.
As with arrays, SMDL does not allow defining variable-sized strings, as these have the same problems as dynamic arrays (e.g.
their size is not know up-front, and their use requires memory allocation).
</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
        <xsd:extension base="Types:SimpleType">
          <xsd:attribute name="Length" type="xsd:long" use="required"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- UML Class 'Array' -->
    <xsd:complexType name="Array">
```

```
      <xsd:annotation>
         <xsd:documentation>An Array type defines a fixed-size array of identically typed elements, where ItemType defines the type of the array items, and Size defines the number of array items.
Multi-dimensional arrays are defined when ItemType is an Array type as well.
Dynamic arrays are not supported by SMDL, as they are not supported by some potential target platforms, and introduce various difficulties in memory management. </xsd:documentation>
         <xsd:documentation>&lt;b&gt;&lt;i&gt;Remarks&lt;/i&gt;&lt;/b&gt;: Nevertheless, specific mechanisms are available to allow dynamic collections of components, either for containment (composition) or references (aggregation).</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:ValueType">
            <xsd:sequence>
               <xsd:element name="ItemType" type="xlink:SimpleLink">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Types:ValueType</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Types:ValueType</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
            <xsd:attribute name="Size" type="xsd:long" use="required"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Structure' -->
   <xsd:complexType name="Structure">
      <xsd:annotation>
         <xsd:documentation>A Structure type collects an arbitrary number of Fields representing the state of the structure.
Within a structure, each field needs to be given a unique name. In order to arrive at semantically correct (data) type definitions, a structure type may not be recursive, i.e. a structure may not have a field that is typed by the structure itself.
A structure can also serve as a namespace to define an arbitrary number of Constants.
</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:ValueType">
            <xsd:sequence>
               <xsd:element name="Constant" type="Types:Constant" minOccurs="0" maxOccurs="unbounded"/>
               <xsd:element name="Field" type="Types:Field" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Exception' -->
```

```
   <xsd:complexType name="Exception">
      <xsd:annotation>
         <xsd:documentation>An Exception represents a non-recoverable error that can occur when calling into an Operation or 
Property getter/setter (within an Operation this is represented by the RaisedException links and within a Property this is 
represented by the GetRaises and SetRaises links, respectively).
An Exception can contain constants and fields (from Structure) as well as operations, properties and associations (from Class).
The fields represent the state variables of the exception which carry additional information when the exception is raised.
Furthermore, an Exception may be Abstract (from Class), and it may inherit from a single base exception (implementation 
inheritance), which is represented by the Base link (from Class).</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:Class"/>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Class' -->
   <xsd:complexType name="Class">
      <xsd:annotation>
         <xsd:documentation>The Class metaclass is derived from Structure. A class may be abstract (attribute Abstract), and it 
may inherit from a single base class (implementation inheritance), which is represented by the Base link.
As the Class metaclass is derived from Structure it can contain constants and fields. Further, it can have arbitrary numbers of 
properties (Property elements), operations (Operation elements), and associations (Association elements).</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:Structure">
            <xsd:sequence>
               <xsd:element name="Base" type="xlink:SimpleLink" minOccurs="0">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Types:Class</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Types:Class</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
               <xsd:element name="Property" type="Types:Property" minOccurs="0" maxOccurs="unbounded"/>
               <xsd:element name="Operation" type="Types:Operation" minOccurs="0" maxOccurs="unbounded"/>
               <xsd:element name="Association" type="Types:Association" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="Abstract" type="xsd:boolean" use="optional" default="false"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- ======================================== -->
   <!-- UML Package 'Types::Features' -->
   <!-- ======================================== -->
```

```
   <!-- A feature is an element that is contained in a type and that typically refers to a type. Additionally, some features have (default) values. -->

   <!-- UML Enumeration 'AccessKind' -->
   <xsd:simpleType name="AccessKind">
      <xsd:annotation>
         <xsd:documentation>This enumeration defines how a property can be accessed.</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:string">
         <xsd:enumeration value="readWrite">
            <xsd:annotation>
               <xsd:documentation>Specifies a property, which has both a getter and a setter.</xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
         <xsd:enumeration value="readOnly">
            <xsd:annotation>
               <xsd:documentation>Specifies a property, which only has a getter.</xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
         <xsd:enumeration value="writeOnly">
            <xsd:annotation>
               <xsd:documentation>Specifies a property, which only has a setter.</xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
      </xsd:restriction>
   </xsd:simpleType>

   <!-- UML Enumeration 'ParameterDirectionKind' -->
   <xsd:simpleType name="ParameterDirectionKind">
      <xsd:annotation>
         <xsd:documentation>This enumeration defines the possible parameter directions.</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:string">
         <xsd:enumeration value="in">
            <xsd:annotation>
               <xsd:documentation>The parameter is read-only to the operation, i.e. its value must be specified on call, and cannot be changed inside the operation.</xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
         <xsd:enumeration value="out">
            <xsd:annotation>
               <xsd:documentation>The parameter is write-only to the operation, i.e. its value is unspecified on call, and must be set by the operation.</xsd:documentation>
            </xsd:annotation>
```

```
        </xsd:enumeration>
        <xsd:enumeration value="inout">
          <xsd:annotation>
            <xsd:documentation>The parameter must be specified on call, and may be changed by the
operation.</xsd:documentation>
          </xsd:annotation>
        </xsd:enumeration>
        <xsd:enumeration value="return">
          <xsd:annotation>
            <xsd:documentation>The parameter represents the operation's return value.</xsd:documentation>
          </xsd:annotation>
        </xsd:enumeration>
      </xsd:restriction>
    </xsd:simpleType>

    <!-- UML Class 'Constant' -->
    <xsd:complexType name="Constant">
      <xsd:annotation>
        <xsd:documentation>A Constant is a feature that is typed by a simple type and that must have a
Value.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
        <xsd:extension base="Types:VisibilityElement">
          <xsd:sequence>
            <xsd:element name="Type" type="xlink:SimpleLink">
              <xsd:annotation>
                <xsd:documentation>Link destination type: Types:SimpleType</xsd:documentation>
                <xsd:appinfo source="http://www.w3.org/1999/xlink">Types:SimpleType</xsd:appinfo>
              </xsd:annotation>
            </xsd:element>
            <xsd:element name="Value" type="Types:SimpleValue"/>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- UML Class 'Field' -->
    <xsd:complexType name="Field">
      <xsd:annotation>
        <xsd:documentation>A Field is a feature that is typed by any value type but String8, and that may have a Default value.
The State attribute defines how the field is published to the simulation environment. Only fields with a State of true are stored
using external persistence. The visibility to the user within the simulation environment can be controlled via the standard SMP
attribute "View". By default, the State flag is set to true and the View attribute defaults to "None" when not applied.
```

```
The Input and Output attributes define whether the field value is an input for internal calculations (i.e. needed in order to
perform these calculations), or an output of internal calculations (i.e. modified when performing these calculations). These
flags default to false, but can be changed from their default value to support dataflow-based design.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:VisibilityElement">
            <xsd:sequence>
               <xsd:element name="Type" type="xlink:SimpleLink">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Types:ValueType</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Types:ValueType</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
               <xsd:element name="Default" type="Types:Value" minOccurs="0"/>
            </xsd:sequence>
            <xsd:attribute name="State" type="xsd:boolean" use="optional" default="true"/>
            <xsd:attribute name="Input" type="xsd:boolean" use="optional" default="false"/>
            <xsd:attribute name="Output" type="xsd:boolean" use="optional" default="false"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Property' -->
   <xsd:complexType name="Property">
      <xsd:annotation>
         <xsd:documentation>A Property has a similar syntax as a Field: It is a feature that references a language type. However,
the semantics is different in that a property does not represent a state and that it can be assigned an Access attribute to
specify how the property can be accessed (either readWrite, readOnly, or writeOnly, see AccessKind).
Furthermore, a property can be assigned a Category attribute to help grouping the properties within its owning type, and a
property may specify an arbitrary number of exceptions that it can raise in its getter (GetRaises) and/or setter
(SetRaises).</xsd:documentation>
         <xsd:documentation>&lt;b&gt;&lt;i&gt;Remark&lt;/i&gt;&lt;/b&gt;: The category can be used in applications as ordering or
filtering criterion, for example in a property grid. The term "property" used here closely corresponds in its semantics to the
same term in the Java Beans specification and in the Microsoft .NET framework. That is, a property formally represents a "getter"
or a "setter" operation or both which allow accessing state or configuration information (or derived information thereof) in a
controlled way and which can also be exposed via interfaces (in contrast to fields).</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:VisibilityElement">
            <xsd:sequence>
               <xsd:element name="Type" type="xlink:SimpleLink">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Types:LanguageType</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Types:LanguageType</xsd:appinfo>
```

```
          </xsd:annotation>
        </xsd:element>
        <xsd:element name="AttachedField" type="xlink:SimpleLink" minOccurs="0">
          <xsd:annotation>
            <xsd:documentation>Link destination type: Types:Field</xsd:documentation>
            <xsd:appinfo source="http://www.w3.org/1999/xlink">Types:Field</xsd:appinfo>
          </xsd:annotation>
        </xsd:element>
        <xsd:element name="GetRaises" type="xlink:SimpleLink" minOccurs="0" maxOccurs="unbounded">
          <xsd:annotation>
            <xsd:documentation>Link destination type: Types:Exception</xsd:documentation>
            <xsd:appinfo source="http://www.w3.org/1999/xlink">Types:Exception</xsd:appinfo>
          </xsd:annotation>
        </xsd:element>
        <xsd:element name="SetRaises" type="xlink:SimpleLink" minOccurs="0" maxOccurs="unbounded">
          <xsd:annotation>
            <xsd:documentation>Link destination type: Types:Exception</xsd:documentation>
            <xsd:appinfo source="http://www.w3.org/1999/xlink">Types:Exception</xsd:appinfo>
          </xsd:annotation>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="Access" type="Types:AccessKind" default="readWrite"/>
      <xsd:attribute name="Category" type="xsd:string" use="optional" default="Properties"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<!-- UML Class 'Association' -->
<xsd:complexType name="Association">
  <xsd:annotation>
    <xsd:documentation>An Association is a feature that is typed by a language type (Type link). An association always
expresses a reference to an instance of the referenced language type. This reference is either another model (if the Type link
refers to a Model or Interface), or it is a field contained in another model (if the Type link refers to a
ValueType).</xsd:documentation>
  </xsd:annotation>
  <xsd:complexContent>
    <xsd:extension base="Types:VisibilityElement">
      <xsd:sequence>
        <xsd:element name="Type" type="xlink:SimpleLink">
          <xsd:annotation>
            <xsd:documentation>Link destination type: Types:LanguageType</xsd:documentation>
            <xsd:appinfo source="http://www.w3.org/1999/xlink">Types:LanguageType</xsd:appinfo>
          </xsd:annotation>
        </xsd:element>
```

```
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Operation' -->
   <xsd:complexType name="Operation">
      <xsd:annotation>
         <xsd:documentation>An Operation may have an arbitrary number of parameters, where at most one of the parameters may be
of Direction = ParameterDirectionKind.return. If such a parameter is absent, the operation is a void function (procedure) without
return value.
An Operation may specify an arbitrary number of exceptions that it can raise (RaisedException).</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:VisibilityElement">
            <xsd:sequence>
               <xsd:element name="Parameter" type="Types:Parameter" minOccurs="0" maxOccurs="unbounded"/>
               <xsd:element name="RaisedException" type="xlink:SimpleLink" minOccurs="0"
                            maxOccurs="unbounded">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Types:Exception</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Types:Exception</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Parameter' -->
   <xsd:complexType name="Parameter">
      <xsd:annotation>
         <xsd:documentation>A Parameter has a Type and a Direction, where the direction may have the values in, out, inout or
return (see ParameterDirectionKind).
When referencing a value type, a parameter may have an additional Default value, which can be used by languages that support
default values for parameters.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Elements:NamedElement">
            <xsd:sequence>
               <xsd:element name="Type" type="xlink:SimpleLink">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Types:LanguageType</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Types:LanguageType</xsd:appinfo>
```

```
            </xsd:annotation>
          </xsd:element>
          <xsd:element name="Default" type="Types:Value" minOccurs="0"/>
        </xsd:sequence>
        <xsd:attribute name="Direction" type="Types:ParameterDirectionKind" default="in"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- ======================================== -->
  <!-- UML Package 'Types::Features::Field versus Property' -->
  <!-- ======================================== -->
  <!-- The semantics of a property is very different from the semantics of a field. A field always has a memory location holding
its value, while a property is a convenience mechanism to represent one or two access operations, namely the setter and/or the
getter. If a property is read-only, there is no setter, if it is write-only, there is to getter. The actual implementation
depends on the target platform and language.
Compared to fields, properties have the advantage that there is no direct memory access, but every access is operation-based.
This allows mapping them to distributed platforms (e.g. CORBA), and ensures that the containing type always has knowledge about
changes of its state (e.g. to support range checking in the setter).
On implementation level, properties are frequently bound to a specific field. This can be expressed by linking to a field (of the
same type) via the AttachedField link.
 -->
  <!-- <b><i>Remark</i></b>: For example, this information can be utilised by a code generator to generate the relevant binding
from the setter and/or the getter to the attached field in the code. -->

  <!-- ======================================== -->
  <!-- UML Package 'Types::Values' -->
  <!-- ======================================== -->
  <!-- A Value represents the state of a ValueType. For each metaclass derived from ValueType, a corresponding metaclass derived
from Value is defined. Values are used in various places. Within the Core Types schema, they are used for the Default value of a
Field, Parameter and AttributeType and for the Value of a Constant. -->

  <!-- UML Class 'Value' -->
  <xsd:complexType name="Value" abstract="true">
    <xsd:annotation>
      <xsd:documentation>The Value metaclass is an abstract base class for specialised values.
The Field attribute specifies the reference to the corresponding field via its name or its locally qualified path. This attribute
can be omitted in cases where no field reference needs to be specified (e.g. on a default value of a
Parameter).</xsd:documentation>
    </xsd:annotation>
    <xsd:attribute name="Field" type="xsd:string" use="optional"/>
  </xsd:complexType>

  <!-- UML Class 'SimpleValue' -->
```

```
   <xsd:complexType name="SimpleValue" abstract="true">
      <xsd:annotation>
         <xsd:documentation>A SimpleValue represents a value that is of simple type (this includes all SMP primitive types as
well as user-defined Integer, Float, String and Enumeration types).
To ensure type safety, specific sub-metaclasses are introduced, which specify the type of the Value
attribute.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:Value"/>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'SimpleArrayValue' -->
   <xsd:complexType name="SimpleArrayValue" abstract="true">
      <xsd:annotation>
         <xsd:documentation>A SimpleArrayValue represents an array of values that are of (the same) simple type.
To ensure type safety, specific sub-metaclasses are introduced, which specify the type of the contained ItemValue
elements.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:Value"/>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'ArrayValue' -->
   <xsd:complexType name="ArrayValue">
      <xsd:annotation>
         <xsd:documentation>An ArrayValue holds values for each array item, represented by the ItemValue elements. The
corresponding array type defines the number of item values.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:Value">
            <xsd:sequence>
               <xsd:element name="ItemValue" type="Types:Value" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'StructureValue' -->
   <xsd:complexType name="StructureValue">
      <xsd:annotation>
         <xsd:documentation>A StructureValue holds field values for all fields of the corresponding structure type. Thereby, the
Field attribute of each contained value specifies the local field name within the structure.</xsd:documentation>
```

```
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:Value">
            <xsd:sequence>
               <xsd:element name="FieldValue" type="Types:Value" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- ======================================== -->
   <!-- UML Package 'Types::Values::SimpleValues' -->
   <!-- ======================================== -->
   <!-- Values corresponding to simple types are introduced as specialized metaclasses in order to allow type-safe and efficient
XML serialization. A specific metaclass is introduced for each SMP primitive type and to specify enumeration values. -->

   <!-- UML Class 'BoolValue' -->
   <xsd:complexType name="BoolValue">
      <xsd:annotation>
         <xsd:documentation>A BoolValue holds a value for an item of type Bool.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:SimpleValue">
            <xsd:attribute name="Value" type="xsd:boolean" use="required"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Char8Value' -->
   <xsd:complexType name="Char8Value">
      <xsd:annotation>
         <xsd:documentation>A Char8Value holds a value for an item of type Char8.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:SimpleValue">
            <xsd:attribute name="Value" type="xsd:string" use="required"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'DateTimeValue' -->
   <xsd:complexType name="DateTimeValue">
      <xsd:annotation>
         <xsd:documentation>A DateTimeValue holds a value for an item of type DateTime.</xsd:documentation>
```

```
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="Types:SimpleValue">
        <xsd:attribute name="Value" type="xsd:dateTime" use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- UML Class 'DurationValue' -->
  <xsd:complexType name="DurationValue">
    <xsd:annotation>
      <xsd:documentation>A DurationValue holds a value for an item of type Duration.</xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="Types:SimpleValue">
        <xsd:attribute name="Value" type="xsd:duration" use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- UML Class 'EnumerationValue' -->
  <xsd:complexType name="EnumerationValue">
    <xsd:annotation>
      <xsd:documentation>An EnumerationValue holds a value for an item of an Enumeration type.</xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="Types:SimpleValue">
        <xsd:attribute name="Value" type="xsd:int" use="required"/>
        <xsd:attribute name="Literal" type="xsd:string" use="optional"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- UML Class 'Float32Value' -->
  <xsd:complexType name="Float32Value">
    <xsd:annotation>
      <xsd:documentation>A Float32Value holds a value for an item of type Float32.</xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="Types:SimpleValue">
        <xsd:attribute name="Value" type="xsd:float" use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
```

```
<!-- UML Class 'Float64Value' -->
<xsd:complexType name="Float64Value">
   <xsd:annotation>
      <xsd:documentation>A Float64Value holds a value for an item of type Float64.</xsd:documentation>
   </xsd:annotation>
   <xsd:complexContent>
      <xsd:extension base="Types:SimpleValue">
         <xsd:attribute name="Value" type="xsd:double" use="required"/>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>

<!-- UML Class 'Int16Value' -->
<xsd:complexType name="Int16Value">
   <xsd:annotation>
      <xsd:documentation>An Int16Value holds a value for an item of type Int16.</xsd:documentation>
   </xsd:annotation>
   <xsd:complexContent>
      <xsd:extension base="Types:SimpleValue">
         <xsd:attribute name="Value" type="xsd:short" use="required"/>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>

<!-- UML Class 'Int32Value' -->
<xsd:complexType name="Int32Value">
   <xsd:annotation>
      <xsd:documentation>An Int32Value holds a value for an item of type Int32.</xsd:documentation>
   </xsd:annotation>
   <xsd:complexContent>
      <xsd:extension base="Types:SimpleValue">
         <xsd:attribute name="Value" type="xsd:int" use="required"/>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>

<!-- UML Class 'Int64Value' -->
<xsd:complexType name="Int64Value">
   <xsd:annotation>
      <xsd:documentation>An Int64Value holds a value for an item of type Int64.</xsd:documentation>
   </xsd:annotation>
   <xsd:complexContent>
      <xsd:extension base="Types:SimpleValue">
```

```
            <xsd:attribute name="Value" type="xsd:long" use="required"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Int8Value' -->
   <xsd:complexType name="Int8Value">
      <xsd:annotation>
         <xsd:documentation>An Int8Value holds a value for an item of type Int8.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:SimpleValue">
            <xsd:attribute name="Value" type="xsd:byte" use="required"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'String8Value' -->
   <xsd:complexType name="String8Value">
      <xsd:annotation>
         <xsd:documentation>A String8Value holds a value for an item of type String8, or for an item of a String
type.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:SimpleValue">
            <xsd:attribute name="Value" type="xsd:string" use="required"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'UInt16Value' -->
   <xsd:complexType name="UInt16Value">
      <xsd:annotation>
         <xsd:documentation>A UInt16Value holds a value for an item of type UInt16.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:SimpleValue">
            <xsd:attribute name="Value" type="xsd:unsignedShort" use="required"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'UInt32Value' -->
   <xsd:complexType name="UInt32Value">
```

```
      <xsd:annotation>
        <xsd:documentation>A UInt32Value holds a value for an item of type UInt32.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
        <xsd:extension base="Types:SimpleValue">
          <xsd:attribute name="Value" type="xsd:unsignedInt" use="required"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- UML Class 'UInt64Value' -->
    <xsd:complexType name="UInt64Value">
      <xsd:annotation>
        <xsd:documentation>A UInt64Value holds a value for an item of type UInt64.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
        <xsd:extension base="Types:SimpleValue">
          <xsd:attribute name="Value" type="xsd:unsignedLong" use="required"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- UML Class 'UInt8Value' -->
    <xsd:complexType name="UInt8Value">
      <xsd:annotation>
        <xsd:documentation>A UInt8Value holds a value for an item of type UInt8.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
        <xsd:extension base="Types:SimpleValue">
          <xsd:attribute name="Value" type="xsd:unsignedByte" use="required"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- ======================================== -->
    <!-- UML Package 'Types::Values::SimpleArrayValues' -->
    <!-- ======================================== -->
    <!-- Values of arrays with items of simple type are introduced as specialized metaclasses in order to allow type-safe and
efficient XML serialization. A specific metaclass is introduced for each SMP primitive type and to specify enumeration values. --
>

    <!-- UML Class 'BoolArrayValue' -->
    <xsd:complexType name="BoolArrayValue">
      <xsd:annotation>
```

```
        <xsd:documentation>The BoolArrayValue holds an array of BoolValue items for an array of type Bool.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
        <xsd:extension base="Types:SimpleArrayValue">
          <xsd:sequence>
            <xsd:element name="ItemValue" type="Types:BoolValue" minOccurs="0" maxOccurs="unbounded"/>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- UML Class 'Char8ArrayValue' -->
    <xsd:complexType name="Char8ArrayValue">
      <xsd:annotation>
        <xsd:documentation>The Char8ArrayValue holds an array of Char8Value items for an array of type
Char8.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
        <xsd:extension base="Types:SimpleArrayValue">
          <xsd:sequence>
            <xsd:element name="ItemValue" type="Types:Char8Value" minOccurs="0" maxOccurs="unbounded"/>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- UML Class 'DateTimeArrayValue' -->
    <xsd:complexType name="DateTimeArrayValue">
      <xsd:annotation>
        <xsd:documentation>The DateTimeArrayValue holds an array of DateTimeValue items for an array of type
DateTime.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
        <xsd:extension base="Types:SimpleArrayValue">
          <xsd:sequence>
            <xsd:element name="ItemValue" type="Types:DateTimeValue" minOccurs="0" maxOccurs="unbounded"/>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- UML Class 'DurationArrayValue' -->
    <xsd:complexType name="DurationArrayValue">
      <xsd:annotation>
```

```
         <xsd:documentation>The DurationArrayValue holds an array of DurationValue items for an array of type
Duration.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:SimpleArrayValue">
            <xsd:sequence>
               <xsd:element name="ItemValue" type="Types:DurationValue" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'EnumerationArrayValue' -->
   <xsd:complexType name="EnumerationArrayValue">
      <xsd:annotation>
         <xsd:documentation>The EnumerationArrayValue holds an array of EnumerationValue items for an array of an Enumeration
type.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:SimpleArrayValue">
            <xsd:sequence>
               <xsd:element name="ItemValue" type="Types:EnumerationValue" minOccurs="0"
                            maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Float32ArrayValue' -->
   <xsd:complexType name="Float32ArrayValue">
      <xsd:annotation>
         <xsd:documentation>The Float32ArrayValue holds an array of Float32Value items for an array of type
Float32.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:SimpleArrayValue">
            <xsd:sequence>
               <xsd:element name="ItemValue" type="Types:Float32Value" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Float64ArrayValue' -->
```

```
   <xsd:complexType name="Float64ArrayValue">
      <xsd:annotation>
         <xsd:documentation>The Float64ArrayValue holds an array of Float64Value items for an array of type
Float64.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:SimpleArrayValue">
            <xsd:sequence>
               <xsd:element name="ItemValue" type="Types:Float64Value" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Int16ArrayValue' -->
   <xsd:complexType name="Int16ArrayValue">
      <xsd:annotation>
         <xsd:documentation>The Int16ArrayValue holds an array of Int16Value items for an array of type
Int16.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:SimpleArrayValue">
            <xsd:sequence>
               <xsd:element name="ItemValue" type="Types:Int16Value" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Int32ArrayValue' -->
   <xsd:complexType name="Int32ArrayValue">
      <xsd:annotation>
         <xsd:documentation>The Int32ArrayValue holds an array of Int32Value items for an array of type
Int32.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:SimpleArrayValue">
            <xsd:sequence>
               <xsd:element name="ItemValue" type="Types:Int32Value" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
```

```
   <!-- UML Class 'Int64ArrayValue' -->
   <xsd:complexType name="Int64ArrayValue">
      <xsd:annotation>
         <xsd:documentation>The Int64ArrayValue holds an array of Int64Value items for an array of type
Int64.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:SimpleArrayValue">
            <xsd:sequence>
               <xsd:element name="ItemValue" type="Types:Int64Value" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Int8ArrayValue' -->
   <xsd:complexType name="Int8ArrayValue">
      <xsd:annotation>
         <xsd:documentation>The Int8ArrayValue holds an array of Int8Value items for an array of type Int8.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:SimpleArrayValue">
            <xsd:sequence>
               <xsd:element name="ItemValue" type="Types:Int8Value" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'String8ArrayValue' -->
   <xsd:complexType name="String8ArrayValue">
      <xsd:annotation>
         <xsd:documentation>The String8ArrayValue holds an array of String8Value items for an array of type String8, or for an
array of a String type.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:SimpleArrayValue">
            <xsd:sequence>
               <xsd:element name="ItemValue" type="Types:String8Value" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
```

```
   <!-- UML Class 'UInt16ArrayValue' -->
   <xsd:complexType name="UInt16ArrayValue">
      <xsd:annotation>
         <xsd:documentation>The UInt16ArrayValue holds an array of UInt16Value items for an array of type 
UInt16.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:SimpleArrayValue">
            <xsd:sequence>
               <xsd:element name="ItemValue" type="Types:UInt16Value" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'UInt32ArrayValue' -->
   <xsd:complexType name="UInt32ArrayValue">
      <xsd:annotation>
         <xsd:documentation>The UInt32ArrayValue holds an array of UInt32Value items for an array of type 
UInt32.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:SimpleArrayValue">
            <xsd:sequence>
               <xsd:element name="ItemValue" type="Types:UInt32Value" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'UInt64ArrayValue' -->
   <xsd:complexType name="UInt64ArrayValue">
      <xsd:annotation>
         <xsd:documentation>The UInt64ArrayValue holds an array of UInt64Value items for an array of type 
UInt64.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:SimpleArrayValue">
            <xsd:sequence>
               <xsd:element name="ItemValue" type="Types:UInt64Value" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
```

```
   <!-- UML Class 'UInt8ArrayValue' -->
   <xsd:complexType name="UInt8ArrayValue">
      <xsd:annotation>
         <xsd:documentation>The UInt8ArrayValue holds an array of UInt8Value items for an array of type 
UInt8.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:SimpleArrayValue">
            <xsd:sequence>
               <xsd:element name="ItemValue" type="Types:UInt8Value" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- ======================================== -->
   <!-- UML Package 'Types::Attributes' -->
   <!-- ======================================== -->
   <!-- This package defines the SMDL attribute mechanism which allows extending SMDL semantics via standard or user-defined 
attributes. -->
   <!-- <b><i>Remark</i></b>: In SMDL, the term attribute is used to denote user-defined metadata, as in the .NET framework. In 
contrast, an attribute in UML denotes a non-functional member of a class, which corresponds to a field or property in SMDL. -->

   <!-- UML Class 'AttributeType' -->
   <xsd:complexType name="AttributeType">
      <xsd:annotation>
         <xsd:documentation>An AttributeType defines a new type available for adding attributes to elements. The AllowMultiple 
attribute specifies if a corresponding Attribute may be attached more than once to a language element, while the Usage element 
defines to which language elements attributes of this type can be attached. An attribute type always references a value type, and 
specifies a Default value.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:Type">
            <xsd:sequence>
               <xsd:element name="Type" type="xlink:SimpleLink">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Types:ValueType</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Types:ValueType</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
               <xsd:element name="Default" type="Types:Value"/>
               <xsd:element name="Usage" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
```

```
            <xsd:attribute name="AllowMultiple" type="xsd:boolean" use="optional" default="false"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Attribute' -->
   <xsd:complexType name="Attribute">
      <xsd:annotation>
         <xsd:documentation>An Attribute element holds name-value pairs allowing to attach user-defined metadata to any
Element.</xsd:documentation>
         <xsd:documentation>&lt;b&gt;&lt;i&gt;Remark&lt;/i&gt;&lt;/b&gt;: This provides a similar mechanism as tagged values in
UML, xsd:appinfo in XML Schema, annotations in Java 5.0 or attributes in the .NET framework. </xsd:documentation>
         <xsd:documentation>&lt;b&gt;&lt;i&gt;Remark&lt;/i&gt;&lt;/b&gt;: A possible application of using attributes could be to
decorate an SMDL model with information needed to guide a code generator, for example to tailor the mappu105 ?ng to
C++.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Elements:Metadata">
            <xsd:sequence>
               <xsd:element name="Type" type="xlink:SimpleLink">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Types:AttributeType</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Types:AttributeType</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
               <xsd:element name="Value" type="Types:Value"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
</xsd:schema>
```

A.3 Smdl Catalogue

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:Catalogue="http://www.esa.int/2008/02/Smdl/Catalogue"
            xmlns:xlink="http://www.w3.org/1999/xlink"
            xmlns:Types="http://www.esa.int/2008/02/Core/Types"
            xmlns:Elements="http://www.esa.int/2008/02/Core/Elements"
            xmlns="http://www.esa.int/2008/02/Smdl/Catalogue"
            targetNamespace="http://www.esa.int/2008/02/Smdl/Catalogue"
```

```
            elementFormDefault="unqualified"
            attributeFormDefault="unqualified">
   <xsd:annotation>
      <xsd:documentation>
This file is generated by the EGOS-MF XML Schema Generation Tool, version 1.0.1.

UML Model Information:
    UML model file:  file:/F:/VST/ECSSMP/design/gen/xmi/Smdl.xmi
    UML model name:  Smdl
    UML metamodel:   http://schema.omg.org/spec/UML/2.1.1
    XMI version:     2.1
    XMI exporter:    EGOS-MF XMI Converter (from MagicDraw UML 12.1), version 1.0.1

XSLT Processing Information:
    Processing date: 2008-03-12T09:26:11.689+01:00
    XSLT processor:  SAXON 8.8 from Saxonica
    XSLT version:    2.0
    XSLT stylesheet: xmi-to-xsd.xslt
      </xsd:documentation>
   </xsd:annotation>
   <!-- Import UML Component 'xlink' -->
   <xsd:import namespace="http://www.w3.org/1999/xlink" schemaLocation="../xlink.xsd"/>
   <!-- Import UML Component 'Core::Types' -->
   <xsd:import namespace="http://www.esa.int/2008/02/Core/Types"
               schemaLocation="../Core/Types.xsd"/>
   <!-- Import UML Component 'Core::Elements' -->
   <xsd:import namespace="http://www.esa.int/2008/02/Core/Elements"
               schemaLocation="../Core/Elements.xsd"/>

   <!-- ======================================== -->
   <!-- UML Package 'Catalogue' -->
   <!-- ======================================== -->
   <!-- This package describes all metamodel elements that are needed in order to define models in a catalogue. Catalogues make
use of the mechanisms defined in Core Types, e.g. enumerations and structures, and they add reference types (interfaces and
components), events and a hierarchical grouping mechanism (namespaces). -->

   <!-- UML InstanceSpecification 'Catalogue' -->
   <xsd:element name="Catalogue" type="Catalogue:Catalogue">
      <xsd:annotation>
         <xsd:documentation>The Catalogue element (of type Catalogue) is the root element of an SMDL
catalogue.</xsd:documentation>
      </xsd:annotation>
   </xsd:element>
```

```
   <!-- ======================================== -->
   <!-- UML Package 'Catalogue::Catalogue' -->
   <!-- ======================================== -->
   <!-- A catalogue contains namespaces, which themselves contain other namespaces and types. Types can either be language types, 
attribute types (both defined in Core Types) or event types. Further, catalogues extend the available language types by reference 
types (interfaces and components). -->

   <!-- UML Class 'Catalogue' -->
   <xsd:complexType name="Catalogue">
      <xsd:annotation>
         <xsd:documentation>A Catalogue is a document that defines types. It contains namespaces as a primary ordering mechanism. 
The names of these namespaces need to be unique within the catalogue.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Elements:Document">
            <xsd:sequence>
               <xsd:element name="Namespace" type="Catalogue:Namespace" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Namespace' -->
   <xsd:complexType name="Namespace">
      <xsd:annotation>
         <xsd:documentation>A Namespace is a primary ordering mechanism. A namespace may contain other namespaces (nested 
namespaces), and does typically contain types. In SMDL, namespaces are contained within a Catalogue (either directly, or within 
another namespace in a catalogue).
All sub-elements of a namespace (namespaces and types) must have unique names.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Elements:NamedElement">
            <xsd:sequence>
               <xsd:element name="Namespace" type="Catalogue:Namespace" minOccurs="0" maxOccurs="unbounded"/>
               <xsd:element name="Type" type="Types:Type" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- ======================================== -->
   <!-- UML Package 'Catalogue::Reference Types' -->
   <!-- ======================================== -->
```

```
   <!-- An instance of a reference type is uniquely determined by a reference to it, and may have internal state. Two instances
of a reference type are equal if and only if they occupy the same (memory) location, i.e. if the references to them are
identical. Two instances with equal values may therefore not be equal if they occupy different (memory) locations. Reference
types include interfaces and components (models and services).
Every reference type supports properties and operations. A component adds a number of features for different purposes. First, it
adds fields to store an internal state, and provided interfaces for interface-based programming. Further, it adds mechanisms to
describe dependencies on other reference types, event sources and sinks for event-based programming, and entry points to allow
calling void functions e.g. from a scheduler. -->

   <!-- UML Class 'ReferenceType' -->
   <xsd:complexType name="ReferenceType" abstract="true">
      <xsd:annotation>
         <xsd:documentation>ReferenceType serves as an abstract base metaclass for Interface and Component. An instance of a
ReferenceType is identified by a reference to it - as opposed to an instance of a ValueType which is identified by its value. A
reference type may have various optional elements:
&lt;ul&gt;
&lt;li&gt;Constant elements specify constants defined in the reference type's name scope;&lt;/li&gt;
&lt;li&gt;NestedType elements specify (local) value types defined in the reference type's name scope;&lt;/li&gt;
&lt;li&gt;Property elements specify the reference type's properties; and&lt;/li&gt;
&lt;li&gt;Operation elements specify the reference type's operations.&lt;/li&gt;
&lt;/ul&gt;</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:LanguageType">
            <xsd:sequence>
               <xsd:element name="Constant" type="Types:Constant" minOccurs="0" maxOccurs="unbounded"/>
               <xsd:element name="NestedType" type="Types:ValueType" minOccurs="0" maxOccurs="unbounded"/>
               <xsd:element name="Property" type="Types:Property" minOccurs="0" maxOccurs="unbounded"/>
               <xsd:element name="Operation" type="Types:Operation" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Component' -->
   <xsd:complexType name="Component" abstract="true">
      <xsd:annotation>
         <xsd:documentation>A Component is a reference type and hence inherits the ability to hold constants, nested types,
properties, and operations. As a Component semantically forms a deployable unit, it may use the available component mechanisms as
specified in the SMP Component Model.  part from the ability to specify a Base component (single implementation inheritance), a
component may have various optional elements:
&lt;ul&gt;
&lt;li&gt;Interface links specify interfaces that the component provides (in SMP this implies that the component implements these
interfaces);&lt;/li&gt;
```

```
&lt;li&gt;EntryPoint elements allow the component to be scheduled (via the Scheduler service) and to listen to global events (via
the EventManager service);&lt;/li&gt;
&lt;li&gt;EventSink elements specify which events the component can receive (these may be registered with other components' event
sources);&lt;/li&gt;
&lt;li&gt;EventSource elements specify events that the component can emit (other components may register their associated event
sink(s) with these);&lt;/li&gt;
&lt;li&gt;Field elements define a component's internal state; and&lt;/li&gt;
&lt;li&gt;Association elements express associations to other components or fields of other components.&lt;/li&gt;
&lt;/ul&gt;</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Catalogue:ReferenceType">
            <xsd:sequence>
               <xsd:element name="Base" type="xlink:SimpleLink" minOccurs="0">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Catalogue:Component</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Catalogue:Component</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
               <xsd:element name="Interface" type="xlink:SimpleLink" minOccurs="0" maxOccurs="unbounded">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Catalogue:Interface</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Catalogue:Interface</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
               <xsd:element name="EntryPoint" type="Catalogue:EntryPoint" minOccurs="0"
                            maxOccurs="unbounded"/>
               <xsd:element name="EventSink" type="Catalogue:EventSink" minOccurs="0" maxOccurs="unbounded"/>
               <xsd:element name="EventSource" type="Catalogue:EventSource" minOccurs="0"
                            maxOccurs="unbounded"/>
               <xsd:element name="Field" type="Types:Field" minOccurs="0" maxOccurs="unbounded"/>
               <xsd:element name="Association" type="Types:Association" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'Interface' -->
   <xsd:complexType name="Interface">
      <xsd:annotation>
         <xsd:documentation>An Interface is a reference type that serves as a contract in a loosely coupled architecture. It has
the ability to contain constants, nested types, properties and operations (from ReferenceType). An Interface may inherit from
other interfaces (interface inheritance), which is represented via the Base links.</xsd:documentation>
```

```
        <xsd:documentation>&lt;i&gt;&lt;b&gt;Remark&lt;/b&gt;&lt;/i&gt;: It is strongly recommended to only use value types,
references and other interfaces in the properties and operations of an interface (i.e. not to use models). Otherwise, a
dependency between a model implementing the interface, and other models referenced by this interface is introduced, which is
against the idea of interface-based or component-based design.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
        <xsd:extension base="Catalogue:ReferenceType">
          <xsd:sequence>
            <xsd:element name="Base" type="xlink:SimpleLink" minOccurs="0" maxOccurs="unbounded">
              <xsd:annotation>
                <xsd:documentation>Link destination type: Catalogue:Interface</xsd:documentation>
                <xsd:appinfo source="http://www.w3.org/1999/xlink">Catalogue:Interface</xsd:appinfo>
              </xsd:annotation>
            </xsd:element>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- UML Class 'Model' -->
    <xsd:complexType name="Model">
      <xsd:annotation>
        <xsd:documentation>The Model metaclass is a component and hence inherits all component mechanisms.
These mechanisms allow using various different modelling approaches.
For a class-based design, a Model may provide a collection of Field elements to define its internal state. For scheduling and
global events, a Model may provide a collection of EntryPoint elements that can be registered with the Scheduler or EventManager
services of a Simulation Environment.
For an interface-based design, a Model may provide (i.e. implement) an arbitrary number of interfaces, which is represented via
the Interface links.
For a component-based design, a Model may provide Container elements to contain other models (composition), and Reference
elements to reference other components (aggregation). These components can either be models or services.
For an event-based design, a Model may support inter-model events via the EventSink and EventSource elements.
For a dataflow-based design, the fields of a Model can be tagged as Input or Output fields.
In addition, a Model may have Association elements to express associations to other models or fields of other models, and it may
define its own value types, which is represented by the NestedType elements.</xsd:documentation>
        <xsd:documentation>&lt;b&gt;&lt;i&gt;Remark&lt;/i&gt;&lt;/b&gt;: The use of nested types is not recommended, as they may
not map to all platform specific models.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
        <xsd:extension base="Catalogue:Component">
          <xsd:sequence>
            <xsd:element name="Container" type="Catalogue:Container" minOccurs="0" maxOccurs="unbounded"/>
            <xsd:element name="Reference" type="Catalogue:Reference" minOccurs="0" maxOccurs="unbounded"/>
          </xsd:sequence>
```

```
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- UML Class 'Service' -->
    <xsd:complexType name="Service">
      <xsd:annotation>
        <xsd:documentation>The Service metaclass is a component and hence inherits all component mechanisms. A Service can
reference one or more interfaces via the Interface links (inherited from Component), where at least one of them must be derived
from Smp::IService (see SMP Component Model), which qualifies it as a service interface.
</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
        <xsd:extension base="Catalogue:Component"/>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- ======================================== -->
    <!-- UML Package 'Catalogue::Modelling Mechanisms' -->
    <!-- ======================================== -->
    <!-- This section describes the different modelling mechanisms. -->

    <!-- ======================================== -->
    <!-- UML Package 'Catalogue::Modelling Mechanisms::Class based Design' -->
    <!-- ======================================== -->
    <!-- This section describes modelling mechanisms for a class based design. -->

    <!-- UML Class 'EntryPoint' -->
    <xsd:complexType name="EntryPoint">
      <xsd:annotation>
        <xsd:documentation>An EntryPoint is a named element of a Component (Model or Service). It corresponds to a void
operation taking no parameters that can be called from an external client (e.g. the Scheduler or Event Manager services). An
Entry Point can reference both Input fields (which must have their Input attribute set to true) and Output fields (which must
have their Output attribute set to true). These links can be used to ensure that all input fields are updated before the entry
point is called, or that all output fields can be used after the entry point has been called.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
        <xsd:extension base="Elements:NamedElement">
          <xsd:sequence>
            <xsd:element name="Input" type="xlink:SimpleLink" minOccurs="0" maxOccurs="unbounded">
              <xsd:annotation>
                <xsd:documentation>Link destination type: Types:Field</xsd:documentation>
                <xsd:appinfo source="http://www.w3.org/1999/xlink">Types:Field</xsd:appinfo>
              </xsd:annotation>
```

```
            </xsd:element>
            <xsd:element name="Output" type="xlink:SimpleLink" minOccurs="0" maxOccurs="unbounded">
               <xsd:annotation>
                  <xsd:documentation>Link destination type: Types:Field</xsd:documentation>
                  <xsd:appinfo source="http://www.w3.org/1999/xlink">Types:Field</xsd:appinfo>
               </xsd:annotation>
            </xsd:element>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>

<!-- ======================================== -->
<!-- UML Package 'Catalogue::Modelling Mechanisms::Component based Design' -->
<!-- ======================================== -->
<!-- This section describes modelling mechanisms for a component based design. -->

<!-- UML Class 'Container' -->
<xsd:complexType name="Container">
   <xsd:annotation>
      <xsd:documentation>A Container defines the rules of composition (containment of children) for a Model.
The type of components that can be contained is specified vi_u32 ?the Type link.
The Lower and Upper attributes specify the multiplicity, i.e. the number of possibly stored components. Therein the upper bound
may be unlimited, which is represented by Upper=-1.
Furthermore, a container may specify a default implementation of the container type via the nefaultModel link.
</xsd:documentation>
      <xsd:documentation>&lt;b&gt;&lt;i&gt;Remark&lt;/i&gt;&lt;/b&gt;: SMDL support tools may use this during instantiation
(i.e. creation of an assembly) to select an initial implementation for newly created contained components.</xsd:documentation>
   </xsd:annotation>
   <xsd:complexContent>
      <xsd:extension base="Elements:NamedElement">
         <xsd:sequence>
            <xsd:element name="Type" type="xlink:SimpleLink">
               <xsd:annotation>
                  <xsd:documentation>Link destination type: Catalogue:ReferenceType</xsd:documentation>
                  <xsd:appinfo source="http://www.w3.org/1999/xlink">Catalogue:ReferenceType</xsd:appinfo>
               </xsd:annotation>
            </xsd:element>
            <xsd:element name="DefaultModel" type="xlink:SimpleLink" minOccurs="0">
               <xsd:annotation>
                  <xsd:documentation>Link destination type: Catalogue:Model</xsd:documentation>
                  <xsd:appinfo source="http://www.w3.org/1999/xlink">Catalogue:Model</xsd:appinfo>
               </xsd:annotation>
            </xsd:element>
```

```
          </xsd:sequence>
          <xsd:attribute name="Lower" type="xsd:long" default="1"/>
          <xsd:attribute name="Upper" type="xsd:long" default="1"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- UML Class 'Reference' -->
    <xsd:complexType name="Reference">
      <xsd:annotation>
        <xsd:documentation>A Reference defines the rules of aggregation (links to components) for a Model.
The type of components (models or services) that can be referenced is specified by the Interface link. Thereby, a service
reference is characterized by an interface that is derived from Smp::IService (see SMP Component Model).
The Lower and Upper attributes specify the multiplicity, i.e. the number of possibly held references to components implementing
this interface. Therein the upper bound may be unlimited, which is represented by Upper=-1.
</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
        <xsd:extension base="Elements:NamedElement">
          <xsd:sequence>
            <xsd:element name="Interface" type="xlink:SimpleLink">
              <xsd:annotation>
                <xsd:documentation>Link destination type: Catalogue:Interface</xsd:documentation>
                <xsd:appinfo source="http://www.w3.org/1999/xlink">Catalogue:Interface</xsd:appinfo>
              </xsd:annotation>
            </xsd:element>
          </xsd:sequence>
          <xsd:attribute name="Lower" type="xsd:long" default="1"/>
          <xsd:attribute name="Upper" type="xsd:long" default="1"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- ======================================== -->
    <!-- UML Package 'Catalogue::Modelling Mechanisms::Event based Design' -->
    <!-- ======================================== -->
    <!-- This section describes modelling mechanisms for an event based design. -->

    <!-- UML Class 'EventType' -->
    <xsd:complexType name="EventType">
      <xsd:annotation>
        <xsd:documentation>An EventType is used to specify the type of an event. This can be used not only to give a meaningful
name to an event type, but also to link it to an existing simple type (via the EventArgs attribute) that is passed as an argument
with every invocation of the event.</xsd:documentation>
```

```
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:Type">
            <xsd:sequence>
               <xsd:element name="EventArgs" type="xlink:SimpleLink" minOccurs="0">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Types:SimpleType</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Types:SimpleType</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'EventSource' -->
   <xsd:complexType name="EventSource">
      <xsd:annotation>
         <xsd:documentation>An EventSource is used to specify that a Component publishes a specific event under a given name. The
Multicast attribute can be used to specify whether any number of sinks can connect to the source (the default), or only a single
sink can connect (Multicast=false).</xsd:documentation>
         <xsd:documentation>&lt;b&gt;&lt;i&gt;Remark&lt;/i&gt;&lt;/b&gt;: An assembly editor can use the Multicast attribute to
configure the user interface accordingly to ease the specification of event links.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Elements:NamedElement">
            <xsd:sequence>
               <xsd:element name="Type" type="xlink:SimpleLink">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Catalogue:EventType</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Catalogue:EventType</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
            <xsd:attribute name="Multicast" type="xsd:boolean" default="true"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'EventSink' -->
   <xsd:complexType name="EventSink">
      <xsd:annotation>
```

```
         <xsd:documentation>An EventSink is used to specify that a Component can receive a specific event using a given name. An EventSink can be connected to any number of EventSource instances (e.g. in an Assembly using EventLink instances).</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Elements:NamedElement">
            <xsd:sequence>
               <xsd:element name="Type" type="xlink:SimpleLink">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Catalogue:EventType</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Catalogue:EventType</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- ======================================== -->
   <!-- UML Package 'Catalogue::Catalogue Attributes' -->
   <!-- ======================================== -->
   <!-- This section summarises pre-defined SMDL attributes that can be attached to elements in a Catalogue. -->

   <!-- UML Enumeration 'ViewKind' -->
   <xsd:simpleType name="ViewKind">
      <xsd:annotation>
         <xsd:documentation>This enumeration defines possible options for the View attribute, which can be used to control if and how an element is made visible when published to the Simulation Environment.
The Simulation Environment &lt;i&gt;must&lt;/i&gt; at least support the "None" and the "All" roles (i.e. hidden or always visible).
The Simulation Environment &lt;i&gt;may&lt;/i&gt; support the selection of different user roles ("Debug", "Expert", "End User"), in which case the "Debug" and the "Expert" role must also be supported as described.</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:string">
         <xsd:enumeration value="None">
            <xsd:annotation>
               <xsd:documentation>The element is not made visible to the user (this is the default).</xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
         <xsd:enumeration value="Debug">
            <xsd:annotation>
               <xsd:documentation>The element is made visible for debugging purposes.
&lt;b&gt;Semantics:&lt;/b&gt;
```

```
&lt;ul&gt;
&lt;li&gt;The element is not visible to end users, neither in the simulation tree nor via scripts.&lt;/li&gt;
&lt;li&gt;If the simulation infrastructure supports the selection of different user roles, the element shall be visible by
"Debug" users only.&lt;/li&gt;&lt;/ul&gt;</xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
         <xsd:enumeration value="Expert">
            <xsd:annotation>
               <xsd:documentation>The element is made visible for expert users.
&lt;b&gt;Semantics:&lt;/b&gt;

&lt;ul&gt;
&lt;li&gt;The element is not visible to end users, neither in the simulation tree nor via scripts.&lt;/li&gt;
&lt;li&gt;If the simulation infrastructure supports the selection of different user roles, the element shall be visible by
"Debug" and "Expert" users.&lt;/li&gt;&lt;/ul&gt;</xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
         <xsd:enumeration value="All">
            <xsd:annotation>
               <xsd:documentation>The element is made visible to all users.
&lt;b&gt;Semantics:&lt;/b&gt;

&lt;ul&gt;
&lt;li&gt;The element is visible to end users, both in the simulation tree and via scripts.&lt;/li&gt;
&lt;li&gt;If the simulation infrastructure supports the selection of different user roles, the element shall be visible by all
users.&lt;/li&gt;&lt;/ul&gt;</xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
      </xsd:restriction>
   </xsd:simpleType>
</xsd:schema>
```

A.4 Smdl Assembly

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:Catalogue="http://www.esa.int/2008/02/Smdl/Catalogue"
            xmlns:Assembly="http://www.esa.int/2008/02/Smdl/Assembly"
            xmlns:xlink="http://www.w3.org/1999/xlink"
            xmlns:Types="http://www.esa.int/2008/02/Core/Types"
            xmlns:Elements="http://www.esa.int/2008/02/Core/Elements"
            xmlns="http://www.esa.int/2008/02/Smdl/Assembly"
```

```
            targetNamespace="http://www.esa.int/2008/02/Smdl/Assembly"
            elementFormDefault="unqualified"
            attributeFormDefault="unqualified">
   <xsd:annotation>
      <xsd:documentation>
This file is generated by the EGOS-MF XML Schema Generation Tool, version 1.0.1.

UML Model Information:
    UML model file:  file:/F:/VST/ECSSMP/design/gen/xmi/Smdl.xmi
    UML model name:  Smdl
    UML metamodel:   http://schema.omg.org/spec/UML/2.1.1
    XMI version:     2.1
    XMI exporter:    EGOS-MF XMI Converter (from MagicDraw UML 12.1), version 1.0.1

XSLT Processing Information:
    Processing date: 2008-03-12T09:26:11.689+01:00
    XSLT processor:  SAXON 8.8 from Saxonica
    XSLT version:    2.0
    XSLT stylesheet: xmi-to-xsd.xslt
      </xsd:documentation>
   </xsd:annotation>
   <!-- Import UML Component 'Smdl::Catalogue' -->
   <xsd:import namespace="http://www.esa.int/2008/02/Smdl/Catalogue"
               schemaLocation="../Smdl/Catalogue.xsd"/>
   <!-- Import UML Component 'xlink' -->
   <xsd:import namespace="http://www.w3.org/1999/xlink" schemaLocation="../xlink.xsd"/>
   <!-- Import UML Component 'Core::Types' -->
   <xsd:import namespace="http://www.esa.int/2008/02/Core/Types"
               schemaLocation="../Core/Types.xsd"/>
   <!-- Import UML Component 'Core::Elements' -->
   <xsd:import namespace="http://www.esa.int/2008/02/Core/Elements"
               schemaLocation="../Core/Elements.xsd"/>

   <!-- ======================================== -->
   <!-- UML Package 'Assembly' -->
   <!-- ======================================== -->
   <!-- This package describes all metamodel elements that are needed in order to define an assembly of model instances. This
includes mechanisms for creating links between model instances, namely between references and implemented interfaces, between
event sources and event sinks, and between output and input fields. -->

   <!-- UML InstanceSpecification 'Assembly' -->
   <xsd:element name="Assembly" type="Assembly:Assembly">
      <xsd:annotation>
         <xsd:documentation>The Assembly element (of type Assembly) is the root element of an SMDL assembly.</xsd:documentation>
```

```
      </xsd:annotation>
   </xsd:element>

   <!-- ======================================== -->
   <!-- UML Package 'Assembly::Assembly' -->
   <!-- ======================================== -->
   <!-- An assembly contains model instances, where each instance references a Model within a catalogue. Further, an assembly may
also contain assembly instances to support sub-assemblies, and service proxies to access services. Each model instance may
contain further model instances as children (as defined via the Containers of the corresponding Model), and links to connect
references and interfaces (InterfaceLink), event sources and event sinks (EventLink), or output and input fields (FieldLink). -->

   <!-- UML Class 'AssemblyNode' -->
   <xsd:complexType name="AssemblyNode" abstract="true">
      <xsd:annotation>
         <xsd:documentation>The AssemblyNode metaclass is an abstract base class for all nodes in the assembly tree and serves as
provider type for interface links. This ensures that the same linking mechanism can be used to resolve model references (via an
interface link to an InstanceNode, i.e. Assembly, ModelInstance or AssemblyInstance) and service references (via an interface
link to a ServiceProxy).</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Elements:NamedElement"/>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'InstanceNode' -->
   <xsd:complexType name="InstanceNode" abstract="true">
      <xsd:annotation>
         <xsd:documentation>The InstanceNode metaclass is an abstract base class for both Assembly and ModelInstance that
provides the common features of both metaclasses.
An InstanceNode represents an instance of a model. Therefore, it has to link to a Model. To allow creating a run-time model
instance, the instance node needs to specify as well which Implementation shall be used when the assembly is instantiated
(loaded) in a Simulation Environment. This is done by specifying a Universally Unique Identifier (UUID).
Depending on the containers of the referenced model, the instance node can hold any number of child model instances.
For each field of the referenced model, the instance node can specify a FieldValue to define the initial value of the associated
Field.
An InstanceNode can link its references, event sinks and input fields, which are specified by the referenced model, to other
model instances via the Link elements.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Assembly:AssemblyNode">
            <xsd:sequence>
               <xsd:element name="Model" type="xlink:SimpleLink">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Catalogue:Model</xsd:documentation>
```

```
                <xsd:appinfo source="http://www.w3.org/1999/xlink">Catalogue:Model</xsd:appinfo>
              </xsd:annotation>
            </xsd:element>
            <xsd:element name="ModelInstance" type="Assembly:ModelInstance" minOccurs="0"
                         maxOccurs="unbounded"/>
            <xsd:element name="FieldValue" type="Types:Value" minOccurs="0" maxOccurs="unbounded"/>
            <xsd:element name="Link" type="Assembly:Link" minOccurs="0" maxOccurs="unbounded"/>
          </xsd:sequence>
          <xsd:attribute name="Implementation" type="Elements:UUID" use="required"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- UML Class 'ServiceProxy' -->
    <xsd:complexType name="ServiceProxy">
      <xsd:annotation>
        <xsd:documentation>A ServiceProxy represents a service which is specified via the Interface link (service type) and the
Name attribute (service name). In the Assembly, a service proxy serves as end point for interface links that resolve service
references of a model. When the SMDL Assembly is instantiated (loaded), the Simulation Environment must resolve the service proxy
(via ISimulator::GetService()) and provide the resolved service interface to using model instances (i.e. model instances that
reference the service proxy via an interface link).</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
        <xsd:extension base="Assembly:AssemblyNode">
          <xsd:sequence>
            <xsd:element name="Interface" type="xlink:SimpleLink">
              <xsd:annotation>
                <xsd:documentation>Link destination type: Catalogue:Interface</xsd:documentation>
                <xsd:appinfo source="http://www.w3.org/1999/xlink">Catalogue:Interface</xsd:appinfo>
              </xsd:annotation>
            </xsd:element>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- UML Class 'Assembly' -->
    <xsd:complexType name="Assembly">
      <xsd:annotation>
        <xsd:documentation>An Assembly is an assembly node that acts as the root element (top-level node) in an SMDL Assembly
document.
Being an instance node, an Assembly is typed by a Model of a catalogue and can contain model instances, links and field values.
Further it can contain service proxies, for use as end points of service links (interface links to services) within the Assembly.
Being a top-level node, an Assembly does not point to its container (as opposed to a model instance).</xsd:documentation>
```

```
        <xsd:documentation>&lt;b&gt;&lt;i&gt;Remark&lt;/i&gt;&lt;/b&gt;: As multiple inheritance cannot be used in the
metamodel, the Document features are replicated in the Assembly metaclass.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
        <xsd:extension base="Assembly:InstanceNode">
          <xsd:sequence>
            <xsd:element name="ServiceProxy" type="Assembly:ServiceProxy" minOccurs="0"
                         maxOccurs="unbounded"/>
          </xsd:sequence>
          <xsd:attribute name="Title" type="xsd:string" use="optional"/>
          <xsd:attribute name="Date" type="xsd:dateTime" use="optional"/>
          <xsd:attribute name="Creator" type="xsd:string" use="optional"/>
          <xsd:attribute name="Version" type="xsd:string" use="optional"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- UML Class 'ModelInstance' -->
    <xsd:complexType name="ModelInstance">
      <xsd:annotation>
        <xsd:documentation>A ModelInstance represents an instance of a model. It is derived from the abstract base metaclass
InstanceNode, which provides most of the functionality. As every model instance is either contained in an assembly, or in another
model instance, it has to specify as well in which Container of the parent it is stored.
In order to allow an assembly to be interpreted without the associated catalogue, the xlink:title attribute of the Container link
shall contain the container name.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
        <xsd:extension base="Assembly:InstanceNode">
          <xsd:sequence>
            <xsd:element name="Container" type="xlink:SimpleLink">
              <xsd:annotation>
                <xsd:documentation>Link destination type: Catalogue:Container</xsd:documentation>
                <xsd:appinfo source="http://www.w3.org/1999/xlink">Catalogue:Container</xsd:appinfo>
              </xsd:annotation>
            </xsd:element>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- UML Class 'AssemblyInstance' -->
    <xsd:complexType name="AssemblyInstance">
      <xsd:annotation>
```

```
         <xsd:documentation>An AssemblyInstance represents a model instance which is specified by an Assembly in &lt;i&gt;another&lt;/i&gt; SMDL Assembly document (a sub-assembly).
Semantics and constraints:
&lt;ul&gt;
&lt;li&gt;The Assembly link must reference an Assembly node in &lt;i&gt;another&lt;/i&gt; SMDL Assembly document.&lt;/li&gt;
&lt;li&gt;The FieldValue elements owned by the AssemblyInstance node override any FieldValue elements specified in the referenced Assembly node.&lt;/li&gt;
&lt;li&gt;The Model of both the AssemblyInstance and the referenced Assembly must be identical.&lt;/li&gt;
&lt;li&gt;The Implementation element of an AssemblyInstance node must be identical with the Implementation specified by the referenced Assembly.&lt;/li&gt;
&lt;li&gt;Model instances in the sub-assembly defined by the AssemblyInstance can be referenced from assembly nodes (e.g. model instances) in the using assembly via Links, where the AssemblyInstance link points to this.&lt;/li&gt;
&lt;/ul&gt;
</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Assembly:ModelInstance">
            <xsd:sequence>
               <xsd:element name="Assembly" type="xlink:SimpleLink">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Assembly:Assembly</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Assembly:Assembly</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
               <xsd:element name="Configuration" type="xlink:SimpleLink" minOccurs="0">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Configuration:Configuration</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Configuration:Configuration</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- ======================================== -->
   <!-- UML Package 'Assembly::Links' -->
   <!-- ======================================== -->
   <!-- Links allow connecting all model instances and the assembly itself together. Four types of links are supported:
<ul>
<li>An Interface Link connects a reference to a provided interface of proper type. This is typically used in interface and component-based design.</li>
<li>An Event Link connects an event sink to an event source of the same type. This is typically used in event-based design.</li>
```

```
<li>A Field Link connects an input field to an output field of the same type. This is typically used in dataflow-based
design.</li>
</ul>
All link metaclasses are derived from a common Link base class. -->

   <!-- UML Class 'Link' -->
   <xsd:complexType name="Link" abstract="true">
      <xsd:annotation>
         <xsd:documentation>The Link metaclass serves as a base metaclass for all links.
The AssemblyInstance link may reference an assembly instance, to support the case where the Link references into a sub-assembly.
If the AssemblyInstance is empty (default), the Link must reference into the containing Assembly, which implicitly defines its
assembly instance.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Elements:NamedElement">
            <xsd:sequence>
               <xsd:element name="AssemblyInstance" type="xlink:SimpleLink" minOccurs="0">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Assembly:AssemblyInstance</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Assembly:AssemblyInstance</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'InterfaceLink' -->
   <xsd:complexType name="InterfaceLink">
      <xsd:annotation>
         <xsd:documentation>An InterfaceLink resolves a reference of an instance node in an assembly.
Therefore, it links to the Reference of the corresponding Model to uniquely identify the link source - together with the
knowledge of its parent model instance, which defines the Model.
In order to uniquely identify the link target, the InterfaceLink links to the Provider assembly node, which provides a matching
Interface, either via its Model (in case of an InstanceNode) or via its associated service implementation (in case of a
ServiceProxy). In the latter case, the Simulation Environment must resolve the service proxy to a service interface when the
assembly is instantiated (loaded).
In order to allow an assembly to be interpreted without the associated catalogue, the xlink:title attribute of the Reference link
shall contain the name of the associated Reference element in the catalogue.
To be semantically correct, the Interface specified by the Provider must coincide with the Interface that the associated
Reference points to, or that is derived from it via interface inheritance.
If the Provider is an instance node, an Interface of the associated Model must match the Reference's Interface.
If the Provider is a service proxy, the Interface of the service proxy must match the Reference's Interface. Note that obviously
the associated service implementation must implement such a matching interface, which can only be verified when the Simulation
```

```
Environment instantiates the assembly (i.e. during runtime) and the service proxy is resolved via its Name using the standard SMP
service acquisition mechanism.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Assembly:Link">
            <xsd:sequence>
               <xsd:element name="Reference" type="xlink:SimpleLink">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Catalogue:Reference</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Catalogue:Reference</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
               <xsd:element name="Provider" type="xlink:SimpleLink">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Assembly:AssemblyNode</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Assembly:AssemblyNode</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'EventLink' -->
   <xsd:complexType name="EventLink">
      <xsd:annotation>
         <xsd:documentation>An EventLink resolves an event sink of an instance node in an assembly. Therefore, it links to the
EventSink of the corresponding Model to uniquely identify the link target - together with the knowledge of its parent model
instance, which defines the Model.
In order to uniquely identify the link source, the EventLink links to an EventSource of an event Provider model instance.
In order to allow an assembly to be interpreted without the associated catalogue, the xlink:title attribute of the EventSink and
EventSource links shall contain the name of the associated EventSink and EventSource elements in the catalogue, respectively.
To be semantically correct, the EventSource of the Provider and the referenced EventSink need to reference the same
EventType.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Assembly:Link">
            <xsd:sequence>
               <xsd:element name="EventSink" type="xlink:SimpleLink">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Catalogue:EventSink</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Catalogue:EventSink</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
```

```
            <xsd:element name="Provider" type="xlink:SimpleLink">
               <xsd:annotation>
                  <xsd:documentation>Link destination type: Assembly:InstanceNode</xsd:documentation>
                  <xsd:appinfo source="http://www.w3.org/1999/xlink">Assembly:InstanceNode</xsd:appinfo>
               </xsd:annotation>
            </xsd:element>
            <xsd:element name="EventSource" type="xlink:SimpleLink">
               <xsd:annotation>
                  <xsd:documentation>Link destination type: Catalogue:EventSource</xsd:documentation>
                  <xsd:appinfo source="http://www.w3.org/1999/xlink">Catalogue:EventSource</xsd:appinfo>
               </xsd:annotation>
            </xsd:element>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>

<!-- UML Class 'FieldLink' -->
<xsd:complexType name="FieldLink">
   <xsd:annotation>
      <xsd:documentation>A FieldLink resolves an input field of an instance node in an assembly.
Therefore, it links to the Input of the corresponding Model to uniquely identify the link target - together with the knowledge of
its parent model instance, which defines the Model.
In order to uniquely identify the link source, the FieldLink links to an Output field of a Source model instance.
The Factor and Offset attributes specify an affine transformation, which is performed when the field link is "executed" via a
Transfer in a Schedule:
&lt;ul&gt;
&lt;li&gt;InputValue = OutputValue * Factor + Offset&lt;/li&gt;
&lt;/ul&gt;
The default transformation is the identity (Factor=1.0, Offset=0.0).
In order to allow an assembly to be interpreted without the associated catalogue, the xlink:title attribute of the Input and
Output links shall contain the name of the associated input and output Field elements in the catalogue, respectively.
To be semantically correct, the Input field needs to have its Input attribute set to true, the Output field of the Source needs
to have its Output attribute set to true, and both fields need to reference the same value type.
</xsd:documentation>
   </xsd:annotation>
   <xsd:complexContent>
      <xsd:extension base="Assembly:Link">
         <xsd:sequence>
            <xsd:element name="Input" type="xlink:SimpleLink">
               <xsd:annotation>
                  <xsd:documentation>Link destination type: Types:Field</xsd:documentation>
                  <xsd:appinfo source="http://www.w3.org/1999/xlink">Types:Field</xsd:appinfo>
               </xsd:annotation>
```

```
          </xsd:element>
          <xsd:element name="Source" type="xlink:SimpleLink">
            <xsd:annotation>
              <xsd:documentation>Link destination type: Assembly:InstanceNode</xsd:documentation>
              <xsd:appinfo source="http://www.w3.org/1999/xlink">Assembly:InstanceNode</xsd:appinfo>
            </xsd:annotation>
          </xsd:element>
          <xsd:element name="Output" type="xlink:SimpleLink">
            <xsd:annotation>
              <xsd:documentation>Link destination type: Types:Field</xsd:documentation>
              <xsd:appinfo source="http://www.w3.org/1999/xlink">Types:Field</xsd:appinfo>
            </xsd:annotation>
          </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="Factor" type="xsd:double" use="optional" default="1.0"/>
        <xsd:attribute name="Offset" type="xsd:double" use="optional" default="0.0"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```

A.5 Smdl Schedule

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:Catalogue="http://www.esa.int/2008/02/Smdl/Catalogue"
            xmlns:Assembly="http://www.esa.int/2008/02/Smdl/Assembly"
            xmlns:Schedule="http://www.esa.int/2008/02/Smdl/Schedule"
            xmlns:xlink="http://www.w3.org/1999/xlink"
            xmlns:Types="http://www.esa.int/2008/02/Core/Types"
            xmlns:Elements="http://www.esa.int/2008/02/Core/Elements"
            xmlns="http://www.esa.int/2008/02/Smdl/Schedule"
            targetNamespace="http://www.esa.int/2008/02/Smdl/Schedule"
            elementFormDefault="unqualified"
            attributeFormDefault="unqualified">
  <xsd:annotation>
    <xsd:documentation>
This file is generated by the EGOS-MF XML Schema Generation Tool, version 1.0.1.

UML Model Information:
    UML model file:  file:/F:/VST/ECSSMP/design/gen/xmi/Smdl.xmi
    UML model name:  Smdl
```

```
    UML metamodel:   http://schema.omg.org/spec/UML/2.1.1
    XMI version:     2.1
    XMI exporter:    EGOS-MF XMI Converter (from MagicDraw UML 12.1), version 1.0.1

XSLT Processing Information:
    Processing date: 2008-03-12T09:26:11.689+01:00
    XSLT processor:  SAXON 8.8 from Saxonica
    XSLT version:    2.0
    XSLT stylesheet: xmi-to-xsd.xslt
      </xsd:documentation>
   </xsd:annotation>
   <!-- Import UML Component 'Smdl::Catalogue' -->
   <xsd:import namespace="http://www.esa.int/2008/02/Smdl/Catalogue"
               schemaLocation="../Smdl/Catalogue.xsd"/>
   <!-- Import UML Component 'Smdl::Assembly' -->
   <xsd:import namespace="http://www.esa.int/2008/02/Smdl/Assembly"
               schemaLocation="../Smdl/Assembly.xsd"/>
   <!-- Import UML Component 'xlink' -->
   <xsd:import namespace="http://www.w3.org/1999/xlink" schemaLocation="../xlink.xsd"/>
   <!-- Import UML Component 'Core::Types' -->
   <xsd:import namespace="http://www.esa.int/2008/02/Core/Types"
               schemaLocation="../Core/Types.xsd"/>
   <!-- Import UML Component 'Core::Elements' -->
   <xsd:import namespace="http://www.esa.int/2008/02/Core/Elements"
               schemaLocation="../Core/Elements.xsd"/>

   <!-- ======================================== -->
   <!-- UML Package 'Schedule' -->
   <!-- ======================================== -->
   <!-- This package describes all metamodel elements that are needed in order to define how the model instances in an assembly
are to be scheduled. This includes mechanisms for tasks and events. -->

   <!-- UML InstanceSpecification 'Schedule' -->
   <xsd:element name="Schedule" type="Schedule:Schedule">
      <xsd:annotation>
         <xsd:documentation>The Schedule element (of type Schedule) is the root element of an SMDL schedule.</xsd:documentation>
      </xsd:annotation>
   </xsd:element>

   <!-- ======================================== -->
   <!-- UML Package 'Schedule::Schedule' -->
   <!-- ======================================== -->
   <!-- The following figure shows the top-level structure of an SMDL Schedule document. -->
```

```
   <!-- UML Class 'Schedule' -->
   <xsd:complexType name="Schedule">
      <xsd:annotation>
         <xsd:documentation>A Schedule is a Document that holds an arbitrary number of tasks (Task elements) and events (Event
elements) triggering these tasks. Additionally, it may specify the origins of epoch time and mission time via its EpochTime and
MissionStart elements, respectively. These values are used to initialise epoch time and mission time via the SetEpochTime() and
SetMissionStart() methods of the TimeKeeper service.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Elements:Document">
            <xsd:sequence>
               <xsd:element name="EpochTime" type="xsd:dateTime" minOccurs="0">
                  <xsd:annotation>
                     <xsd:documentation>The origin of the schedule's EpochTime. This is typically used to initialise epoch time
in the TimeKeeper via the SetEpochTime() operation.</xsd:documentation>
                  </xsd:annotation>
               </xsd:element>
               <xsd:element name="MissionStart" type="xsd:dateTime" minOccurs="0">
                  <xsd:annotation>
                     <xsd:documentation>The origin of the schedule's MissionTime. This is typically used to initialise mission
time in the TimeKeeper via the SetMissionStart() operation.</xsd:documentation>
                  </xsd:annotation>
               </xsd:element>
               <xsd:element name="Task" type="Schedule:Task" minOccurs="0" maxOccurs="unbounded"/>
               <xsd:element name="Event" type="Schedule:Event" minOccurs="0" maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- ======================================== -->
   <!-- UML Package 'Schedule::Tasks' -->
   <!-- ======================================== -->
   <!-- Tasks are elements that can be triggered by events. The most simple task is one calling a single entry point only, but
more complex tasks can call a series of entry points of different providers. In addition, tasks can trigger data transfers of
field links in an assembly. -->

   <!-- UML Class 'Task' -->
   <xsd:complexType name="Task">
      <xsd:annotation>
         <xsd:documentation>A Task is a container of activities. The order of activities defines in which order the entry points
referenced by the Activity elements are called.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
```

```
        <xsd:extension base="Elements:NamedElement">
          <xsd:sequence>
            <xsd:element name="Activity" type="Schedule:Activity" minOccurs="0" maxOccurs="unbounded"/>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- UML Class 'Activity' -->
    <xsd:complexType name="Activity" abstract="true">
      <xsd:annotation>
        <xsd:documentation>An Activity is the abstract base class for the three different activities that can be contained in a
Task.
&lt;ul&gt;
&lt;li&gt;A Trigger allows to execute an EntryPoint.&lt;/li&gt;
&lt;li&gt;A Transfer allows to initiate a data transfer as defined in a FieldLink.&lt;/li&gt;
&lt;li&gt;A SubTask allows to execute all activities defined in another Task.&lt;/li&gt;
&lt;/ul&gt;</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
        <xsd:extension base="Elements:NamedElement"/>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- UML Class 'Trigger' -->
    <xsd:complexType name="Trigger">
      <xsd:annotation>
        <xsd:documentation>A Trigger selects an EntryPoint defined in the corresponding Model of a Catalogue from a Provider
instance defined in an Assembly.
In order to allow a schedule to be interpreted without the associated catalogue, the xlink:title attribute of the EntryPoint link
shall contain the name of the entry point.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
        <xsd:extension base="Schedule:Activity">
          <xsd:sequence>
            <xsd:element name="Provider" type="xlink:SimpleLink">
              <xsd:annotation>
                <xsd:documentation>Link destination type: Assembly:InstanceNode</xsd:documentation>
                <xsd:appinfo source="http://www.w3.org/1999/xlink">Assembly:InstanceNode</xsd:appinfo>
              </xsd:annotation>
            </xsd:element>
            <xsd:element name="EntryPoint" type="xlink:SimpleLink">
              <xsd:annotation>
                <xsd:documentation>Link destination type: Catalogue:EntryPoint</xsd:documentation>
```

```
                <xsd:appinfo source="http://www.w3.org/1999/xlink">Catalogue:EntryPoint</xsd:appinfo>
              </xsd:annotation>
            </xsd:element>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- UML Class 'Transfer' -->
    <xsd:complexType name="Transfer">
      <xsd:annotation>
        <xsd:documentation>A Transfer selects a FieldLink defined in an Assembly, to initiate its execution to transfer data
from the source to the target.
The transformation specified by the Field Link is performed during the transfer of the associated field
value.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
        <xsd:extension base="Schedule:Activity">
          <xsd:sequence>
            <xsd:element name="FieldLink" type="xlink:SimpleLink">
              <xsd:annotation>
                <xsd:documentation>Link destination type: Assembly:FieldLink</xsd:documentation>
                <xsd:appinfo source="http://www.w3.org/1999/xlink">Assembly:FieldLink</xsd:appinfo>
              </xsd:annotation>
            </xsd:element>
          </xsd:sequence>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>

    <!-- UML Class 'SubTask' -->
    <xsd:complexType name="SubTask">
      <xsd:annotation>
        <xsd:documentation>A SubTask references another Task in a Schedule, to initiate all activities defined in it.
Circular references between tasks are not allowed.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
        <xsd:extension base="Schedule:Activity">
          <xsd:sequence>
            <xsd:element name="Task" type="xlink:SimpleLink">
              <xsd:annotation>
                <xsd:documentation>Link destination type: Schedule:Task</xsd:documentation>
                <xsd:appinfo source="http://www.w3.org/1999/xlink">Schedule:Task</xsd:appinfo>
              </xsd:annotation>
```

```
            </xsd:element>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>

<!-- ======================================== -->
<!-- UML Package 'Schedule::Events' -->
<!-- ======================================== -->
<!-- Events are elements in a schedule that can trigger tasks, which themselves reference entry points or field links in
assembly nodes. Triggers can either be single events, i.e. they trigger tasks only once, or cyclic events, i.e. they trigger
tasks several times. The time when to trigger an event for the first time can be specified in each of the four time kinds
supported by SMP. Therefore, four different types of events exist which all derive from a common base class. -->

<!-- UML Enumeration 'TimeKind' -->
<xsd:simpleType name="TimeKind">
   <xsd:annotation>
      <xsd:documentation>This enumeration defines the four SMP time formats.</xsd:documentation>
   </xsd:annotation>
   <xsd:restriction base="xsd:string">
      <xsd:enumeration value="SimulationTime">
         <xsd:annotation>
            <xsd:documentation>Simulation Time starts at 0 when the simulation is kicked off. It progresses while the
simulation is in Executing state.</xsd:documentation>
         </xsd:annotation>
      </xsd:enumeration>
      <xsd:enumeration value="EpochTime">
         <xsd:annotation>
            <xsd:documentation>Epoch Time is an absolute time and typically progresses with simulation time. The offset
between epoch time and simulation time may get changed during a simulation.</xsd:documentation>
         </xsd:annotation>
      </xsd:enumeration>
      <xsd:enumeration value="ZuluTime">
         <xsd:annotation>
            <xsd:documentation>Zulu Time is the computer clock time, also called wall clock time. It progresses whether the
simulation is in Executing or Standby state, and is not necessarily related to simulation, epoch or mission
time.</xsd:documentation>
         </xsd:annotation>
      </xsd:enumeration>
      <xsd:enumeration value="MissionTime">
         <xsd:annotation>
            <xsd:documentation>Mission Time is a relative time (duration from some start time) and progresses with epoch
time.</xsd:documentation>
         </xsd:annotation>
```

```
        </xsd:enumeration>
      </xsd:restriction>
   </xsd:simpleType>

   <!-- UML Class 'Event' -->
   <xsd:complexType name="Event" abstract="true">
      <xsd:annotation>
         <xsd:documentation>An Event defines a point in time when to execute a collection of tasks. Such a scheduled event can
either be called only once, or it may be repeated using a given cycle time. While the first type of event is called a single
event or timed event, repeated events are called cyclic.
The Repeat attribute defines how often an event shall be repeated.
&lt;ul&gt;
&lt;li&gt;An event with Repeat=0 will not be repeated, so it will be only called once. This is a single event.&lt;/li&gt;
&lt;li&gt;An event with Repeat&gt;0 will be called Repeat+1 times, so it is cyclic.&lt;/li&gt;
&lt;li&gt;Finally, a value of Repeat=-1 indicates that an event shall be called forever (unlimited number of repeats). For a
cyclic event, the CycleTime needs to specify a positive Duration between two consecutive executions of the event.&lt;/li&gt;
&lt;/ul&gt;
Four different time formats are supported to define the time of the (first) execution of the event (see TimeKind). For each of
these four time kinds, a dedicated metaclass derived from Event is defined. These are required because relative times (simulation
and mission times) are specified using a Duration, while absolute times (epoch and Zulu times) are specified using a
DateTime.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Elements:NamedElement">
            <xsd:sequence>
               <xsd:element name="Task" type="xlink:SimpleLink" minOccurs="0" maxOccurs="unbounded">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Schedule:Task</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Schedule:Task</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
            <xsd:attribute name="CycleTime" type="xsd:duration"/>
            <xsd:attribute name="Repeat" type="xsd:long" use="optional" default="-1"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'SimulationEvent' -->
   <xsd:complexType name="SimulationEvent">
      <xsd:annotation>
         <xsd:documentation>A SimulationEvent is derived from Event and adds a SimulationTime attribute.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
```

```
      <xsd:extension base="Schedule:Event">
        <xsd:attribute name="SimulationTime" type="xsd:duration"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- UML Class 'EpochEvent' -->
  <xsd:complexType name="EpochEvent">
    <xsd:annotation>
      <xsd:documentation>An EpochEvent is derived from Event and adds an EpochTime attribute.</xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="Schedule:Event">
        <xsd:attribute name="EpochTime" type="xsd:dateTime"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- UML Class 'MissionEvent' -->
  <xsd:complexType name="MissionEvent">
    <xsd:annotation>
      <xsd:documentation>A MissionEvent is derived from Event and adds a MissionTime attribute.</xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="Schedule:Event">
        <xsd:attribute name="MissionTime" type="xsd:duration"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- UML Class 'ZuluEvent' -->
  <xsd:complexType name="ZuluEvent">
    <xsd:annotation>
      <xsd:documentation>A ZuluEvent is derived from Event and adds a ZuluTime attribute.</xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
      <xsd:extension base="Schedule:Event">
        <xsd:attribute name="ZuluTime" type="xsd:dateTime"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- ======================================== -->
  <!-- UML Package 'Schedule::Schedule Attributes' -->
```

```
   <!-- ======================================== -->
   <!-- This section summarises pre-defined SMDL attributes that can be attached to elements in a Schedule. -->

   <!-- UML Enumeration 'FieldUpdateKind' -->
   <xsd:simpleType name="FieldUpdateKind">
      <xsd:annotation>
         <xsd:documentation>This enumeration allows to specify the behaviour when a Trigger is updated.</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:string">
         <xsd:enumeration value="None">
            <xsd:annotation>
               <xsd:documentation>Field values are not updated automatically when the entry point is executed. In this case, all
field updates must be explicitly scheduled via Transfer elements.</xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
         <xsd:enumeration value="Pull">
            <xsd:annotation>
               <xsd:documentation>All input fields associated with the entry point are updated from the linked outputs
&lt;i&gt;before&lt;/i&gt; the entry point is called.</xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
         <xsd:enumeration value="Push">
            <xsd:annotation>
               <xsd:documentation>The values of all output fields associated with the entry point are automatically transferred
to their linked input fields (in other models) &lt;i&gt;after&lt;/i&gt; the entry point has been called.</xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
         <xsd:enumeration value="PullPush">
            <xsd:annotation>
               <xsd:documentation>Combination of Pull and Push. That is, all input fields associated with the entry point are
updated from the linked outputs &lt;i&gt;before&lt;/i&gt; the entry point is called and the values of all output fields
associated with the entry point are automatically transferred to their linked input fields (in other models)
&lt;i&gt;after&lt;/i&gt; the entry point has been called.</xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
      </xsd:restriction>
   </xsd:simpleType>
</xsd:schema>
```

A.6 Smdl Package

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:Catalogue="http://www.esa.int/2008/02/Smdl/Catalogue"
            xmlns:Package="http://www.esa.int/2008/02/Smdl/Package"
            xmlns:xlink="http://www.w3.org/1999/xlink"
            xmlns:Types="http://www.esa.int/2008/02/Core/Types"
            xmlns:Elements="http://www.esa.int/2008/02/Core/Elements"
            xmlns="http://www.esa.int/2008/02/Smdl/Package"
            targetNamespace="http://www.esa.int/2008/02/Smdl/Package"
            elementFormDefault="unqualified"
            attributeFormDefault="unqualified">
   <xsd:annotation>
      <xsd:documentation>
This file is generated by the EGOS-MF XML Schema Generation Tool, version 1.0.1.

UML Model Information:
    UML model file:  file:/F:/VST/ECSSMP/design/gen/xmi/Smdl.xmi
    UML model name:  Smdl
    UML metamodel:   http://schema.omg.org/spec/UML/2.1.1
    XMI version:     2.1
    XMI exporter:    EGOS-MF XMI Converter (from MagicDraw UML 12.1), version 1.0.1

XSLT Processing Information:
    Processing date: 2008-03-12T09:26:11.689+01:00
    XSLT processor:  SAXON 8.8 from Saxonica
    XSLT version:    2.0
    XSLT stylesheet: xmi-to-xsd.xslt
      </xsd:documentation>
   </xsd:annotation>
   <!-- Import UML Component 'Smdl::Catalogue' -->
   <xsd:import namespace="http://www.esa.int/2008/02/Smdl/Catalogue"
               schemaLocation="../Smdl/Catalogue.xsd"/>
   <!-- Import UML Component 'xlink' -->
   <xsd:import namespace="http://www.w3.org/1999/xlink" schemaLocation="../xlink.xsd"/>
   <!-- Import UML Component 'Core::Types' -->
   <xsd:import namespace="http://www.esa.int/2008/02/Core/Types"
               schemaLocation="../Core/Types.xsd"/>
   <!-- Import UML Component 'Core::Elements' -->
   <xsd:import namespace="http://www.esa.int/2008/02/Core/Elements"
               schemaLocation="../Core/Elements.xsd"/>

   <!-- ======================================== -->
   <!-- UML Package 'Package' -->
   <!-- ======================================== -->
```

```
   <!-- This package describes all metamodel elements that are needed in order to define how implementations of types defined in
catalogues are packaged. This includes not only models, which may have different implementations in different packages, but as
well all other user-defined types. -->

   <!-- UML InstanceSpecification 'Package' -->
   <xsd:element name="Package" type="Package:Package">
      <xsd:annotation>
         <xsd:documentation>The Package element (of type Package) is the root element of an SMDL package.</xsd:documentation>
      </xsd:annotation>
   </xsd:element>

   <!-- ======================================== -->
   <!-- UML Package 'Package::Package' -->
   <!-- ======================================== -->
   <!-- The following figure shows the top-level structure of an SMDL Package document. -->

   <!-- UML Class 'Implementation' -->
   <xsd:complexType name="Implementation">
      <xsd:annotation>
         <xsd:documentation>An Implementation selects a single Type from a catalogue for a package. For the implementation, the
Uuid of the type is used, unless the type is a Model: For a model, a different Uuid for the implementation can be specified, as
for a model, different implementations may exist in different packages.
The purpose of the Implementation metaclass is to make a selected type available (as part of a package) for use in a simulation.
The detailed meaning is specific to each platform, and detailed in the individual platform mappings.</xsd:documentation>
         <xsd:documentation>&lt;b&gt;&lt;i&gt;Remark&lt;/i&gt;&lt;/b&gt;: As an implementation is not a named element, is has
neither a name nor a description. Nevertheless, it can be shown in a tool using name and description of the type it
references.</xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
         <xsd:element name="Type" type="xlink:SimpleLink">
            <xsd:annotation>
               <xsd:documentation>Link destination type: Types:Type</xsd:documentation>
               <xsd:appinfo source="http://www.w3.org/1999/xlink">Types:Type</xsd:appinfo>
            </xsd:annotation>
         </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="Uuid" type="Elements:UUID" use="optional"/>
   </xsd:complexType>

   <!-- UML Class 'Package' -->
   <xsd:complexType name="Package">
      <xsd:annotation>
         <xsd:documentation>A Package is a Document that holds an arbitrary number of Implementation elements. Each of these
implementations references a type in a catalogue that shall be implemented in the package.
```

```
In addition, a package may reference other packages as a Dependency. This indicates that a type referenced from an implementation
in the package requires a type implemented in the referenced package.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Elements:Document">
            <xsd:sequence>
               <xsd:element name="Dependency" type="xlink:SimpleLink" minOccurs="0" maxOccurs="unbounded">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type: Package:Package</xsd:documentation>
                     <xsd:appinfo source="http://www.w3.org/1999/xlink">Package:Package</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
               <xsd:element name="Implementation" type="Package:Implementation" minOccurs="0"
                            maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
</xsd:schema>
```

A.7 Smdl Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:Configuration="http://www.esa.int/2008/02/Smdl/Configuration"
            xmlns:xlink="http://www.w3.org/1999/xlink"
            xmlns:Types="http://www.esa.int/2008/02/Core/Types"
            xmlns:Elements="http://www.esa.int/2008/02/Core/Elements"
            xmlns="http://www.esa.int/2008/02/Smdl/Configuration"
            targetNamespace="http://www.esa.int/2008/02/Smdl/Configuration"
            elementFormDefault="unqualified"
            attributeFormDefault="unqualified">
   <xsd:annotation>
      <xsd:documentation>
This file is generated by the EGOS-MF XML Schema Generation Tool, version 1.0.1.

UML Model Information:
    UML model file:  file:/F:/VST/ECSSMP/design/gen/xmi/Smdl.xmi
    UML model name:  Smdl
    UML metamodel:   http://schema.omg.org/spec/UML/2.1.1
    XMI version:     2.1
    XMI exporter:    EGOS-MF XMI Converter (from MagicDraw UML 12.1), version 1.0.1
```

```
XSLT Processing Information:
    Processing date: 2008-03-12T09:26:11.689+01:00
    XSLT processor:  SAXON 8.8 from Saxonica
    XSLT version:    2.0
    XSLT stylesheet: xmi-to-xsd.xslt
      </xsd:documentation>
   </xsd:annotation>
   <!-- Import UML Component 'xlink' -->
   <xsd:import namespace="http://www.w3.org/1999/xlink" schemaLocation="../xlink.xsd"/>
   <!-- Import UML Component 'Core::Types' -->
   <xsd:import namespace="http://www.esa.int/2008/02/Core/Types"
               schemaLocation="../Core/Types.xsd"/>
   <!-- Import UML Component 'Core::Elements' -->
   <xsd:import namespace="http://www.esa.int/2008/02/Core/Elements"
               schemaLocation="../Core/Elements.xsd"/>

   <!-- ======================================== -->
   <!-- UML Package 'Configuration' -->
   <!-- ======================================== -->
   <!-- This package describes all metamodel elements that are needed in order to define an SMDL configuration document. A
configuration document allows specifying arbitrary field values of component instances in the simulation hierarchy. This can be
used to initialise or reinitialise the simulation. -->

   <!-- UML InstanceSpecification 'Configuration' -->
   <xsd:element name="Configuration" type="Configuration:Configuration">
      <xsd:annotation>
         <xsd:documentation>The Configuration element (of type Configuration) is the root element of an SMDL
configuration.</xsd:documentation>
      </xsd:annotation>
   </xsd:element>

   <!-- UML Class 'ComponentConfiguration' -->
   <xsd:complexType name="ComponentConfiguration">
      <xsd:annotation>
         <xsd:documentation>A ComponentConfiguration defines field values of a component instance (model or service instance).
The component instance is specified via the Path attribute, which can be
&lt;ul&gt;
&lt;li&gt;absolute, i.e. the path starts at the root of the simulation tree (e.g. Path="/Models/Spacecraft/AOCS" or
Path="/Services/CustomService"), or&lt;/li&gt;
&lt;li&gt;relative, i.e. the path is appended to the path of the parent component configuration.&lt;/li&gt;
&lt;/ul&gt;
```

```
Each FieldValue is an instance of one of the available Value metaclasses (section 4.4). A FieldValue has to reference the
corresponding field of the component via its Field attribute which specifies the field's local name/path (e.g. Field="field1" or
Field="struct1.structField2").
In addition to the ability to define a hierarchy of component configurations via the Component element, the Include element
enables to include another Configuration file using a relative or absolute Path for it.</xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
         <xsd:element name="Include" type="Configuration:ConfigurationUsage" minOccurs="0"
                      maxOccurs="unbounded"/>
         <xsd:element name="Component" type="Configuration:ComponentConfiguration" minOccurs="0"
                      maxOccurs="unbounded"/>
         <xsd:element name="FieldValue" type="Types:Value" minOccurs="0" maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="Path" type="xsd:string" use="required"/>
   </xsd:complexType>

   <!-- UML Class 'Configuration' -->
   <xsd:complexType name="Configuration">
      <xsd:annotation>
         <xsd:documentation>A Configuration acts as the root element (top-level node) in an SMDL Configuration document. A
configuration contains a tree of component configurations that define field values of component instances (model or service
instances). Further, a configuration may include other configuration documents via the Include element.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Elements:Document">
            <xsd:sequence>
               <xsd:element name="Include" type="Configuration:ConfigurationUsage" minOccurs="0"
                            maxOccurs="unbounded"/>
               <xsd:element name="Component" type="Configuration:ComponentConfiguration" minOccurs="0"
                            maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>

   <!-- UML Class 'ConfigurationUsage' -->
   <xsd:complexType name="ConfigurationUsage">
      <xsd:annotation>
         <xsd:documentation>A ConfigurationUsage allows to include another SMDL Configuration document. The external
configuration document is referenced via the Configuration link. The Path specifies the (optional) prefix that shall be applied
to all paths in the external configuration document. This allows pre-defined configuration documents to be re-used for the
configuration of components in various places in the model hierarchy.</xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
```

```
        <xsd:element name="Configuration" type="xlink:SimpleLink">
          <xsd:annotation>
            <xsd:documentation>Link destination type: Configuration:Configuration</xsd:documentation>
            <xsd:appinfo source="http://www.w3.org/1999/xlink">Configuration:Configuration</xsd:appinfo>
          </xsd:annotation>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="Path" type="xsd:string" use="optional"/>
    </xsd:complexType>
</xsd:schema>
```