EUROPEAN SPACE AGENCY
DIRECTORATE OF OPERATIONS AND INFRASTRUCTURE

SMP 2.0 Metamodel

EGOS-SIM-GEN-TN-0100

Issue 1 Revision 2

28 October 2005

This Page is Intentionally left Blank

ABSTRACT

This technical note contains the specification of the Metamodel for the SMP2 standard. This Metamodel is called the Simulation Model Definition Language (SMDL).

DOCUMENT APPROVAL

Prepared by	Organisation	Signature	Date
Peter Ellsiepen Peter Fritzen	VEGA VEGA		28 October 2005

Verified by	Organisation	Signature	Date
Christine Dingeldey	VEGA		28 October 2005

Approved by	Organisation	Signature	Date
Niklas Lindman	ESOC/OPS-GIC		

DOCUMENT STATUS SHEET

1. Issue	2. Revision	3. Date	4. Reason for Change
0	Beta 1	13 April 2004	Open Issues marked as “Beta 1”.
0	Beta 2	24 May 2004	Open Issues marked as “Beta 2”.
0	RC1	06 August 2004	RIDs raised on Beta 2, and agreements from CCB meeting.
1	0	13 October 2004	Initial Release of SMP 2.0 Standard.
1	1	28 February 2005	First Update of SMP 2.0 Standard.
1	2	28 October 2005	Second Update of SMP 2.0 Standard.

This Page is Intentionally left Blank

DOCUMENT CHANGE RECORD

DOCUMENT CHANGE RECORD			**DCI NO**	N/A
Changes from SMP 2.0 Metamodel Issue 1 Revision 1 to SMP 2.0 Metamodel Issue 1 Revision 2			**DATE**	28 October 2005
			ORIGINATOR	SMP CCB
			APPROVED BY	Niklas Lindman
1. PAGE	**2. PARAGRAPH**	**3. ISSUE**	**4. CHANGES MADE**	
15	1	36	Package and Workspace added to introduction.	
15	1.1	47	UML diagrams have been rendered more precisely as being UML class diagrams.	
17	1.4	46	SMP Handbook and Alpha Specification moved from applicable documents to reference documents. Version of date of SMP2 documents updated.	
19	2.1	36	New document types `Package` and `Workspace` added.	
21	2.3	47	UML diagrams have been rendered more precisely as being UML class diagrams.	
22	2.4	36	New XML schemas `Package` and `Workspace` added. XML namespaces modified from 2005/02 to 2005/10.	
41	4.3	14, 17	`Feature` removed, and base class of `Field` and `Operation` changed to `VisibilityElement`.	
49	5.2.2	14, 17	Base class of `Property` changed to `VisibilityElement`.	
51	5.2.3	14, 17	Base class of `Association` changed to `VisibilityElement`.	
53	5.3.1	19	`ReferenceType` made abstract.	
66	6.2.3	18	`Publisher` renamed to `Provider`.	
69	7	29	Updated to align with update of `IScheduler` interface in SMP2 Component Model.	
70	7.2.1.1	25	`Activity` made abstract base class of new classes `Trigger`, `Transfer`, and `SubTask`.	
72	7.3	29	`TimedEvent` and `CyclicEvent` replaced by `SimulationEvent`, `MissionEvent`, `EpochEvent` and `ZuluEvent`.	
74	7.4	2	Scheduler attribute for `Initialisation` Tasks added.	
75	8		Specification of `Package` added.	
77	9	36	Specification of `Workspace` added.	
83	11	36	XML Schema documents updated, and Package and Workspace Schemas added.	

This Page is Intentionally left Blank

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

This Page is Intentionally left Blank

1. INTRODUCTION

The Platform Independent Model (**PIM**) of the Simulation Model Portability 2 (**SMP2**) standard consists of two parts:

1. SMP2 Component Model [AD-2], which includes SMP2 Simulation Services,
2. SMP2 Metamodel, also called Simulation Model Definition Language (**SMDL**).

This document details the Simulation Model Definition Language of the SMP2 Platform Independent Model. As this Simulation Model Definition Language 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 Catalogues), to integrate models (SMDL Assemblies), to schedule models (SMDL Schedules), to package model implementations (SMDL Packages), and to group SMDL documents (SMDL Workspaces) are defined in this Metamodel.

1.1 Purpose

This document aims at providing an unambiguous definition of the Simulation Model Definition Language. This language consists of five top-level elements, namely a Catalogue that defines SMP2 models, an Assembly that defines how a collection of model instances is assembled, a Schedule that defines how an assembly shall be scheduled, a Package that defines how the implementations of models are packaged, and Workspace that allows linking all these documents together. 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 three 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.
- **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 an appendix (Appendix B: XML Schemas).

1.2 Scope

This document defines in detail all mechanisms available to define SMP2 models, their packaging, their integration, and their execution in an SMP2 compliant run-time environment. It does not target model developers, but rather tool developers who need to understand the file formats down to the lowest level of detail. A high level overview of the concepts provided by the SMDL, and how they can be applied for model development, model integration, and simulation execution is provided in the SMP2 Handbook [AD-1]. Nevertheless, this document may provide useful reference information for advanced model developers.

1.3 Definitions, acronyms and abbreviations

AD	Applicable Document
API	Application Programming Interface
CLS	Common Language Specification
ESOC	European Space Operations Centre
HTML	Hypertext Mark-up Language
J2EE	Java 2 Enterprise Edition
MJD	Modified Julian Date
N/A	Not Applicable
PIM	Platform Independent Model
RD	Reference Document
SMDL	Simulation Model Definition Language
SMP	Simulation Model Portability
SMP1	Simulation Model Portability 1
SMP2	Simulation Model Portability 2
SQL	Structured Query Language
TBC	To be confirmed
TBD	To be defined
UML	Unified Modelling Language
URI	Uniform Resource Identifier
UUID	Universally Unique Identifier
XMI	XML Metadata Interchange

1.4 References

1.4.1 Applicable Documents

Applicable documents are denoted with AD-n where n is the number in the following list:

AD-1 SMP 2.0 Handbook
EGOS-SIM-GEN-TN-0099, Issue 1.2, 28-Oct-2005

AD-2 SMP 2.0 Component Model
EGOS-SIM-GEN-TN-0101, Issue 1.2, 28-Oct-2005

AD-3 SMP 2.0 C++ Mapping
EGOS-SIM-GEN-TN-0102, Issue 1.2, 28-Oct-2005

1.4.2 Reference Documents

Reference documents are denoted with RD-n where n is the number in the following list:

RD-1 Simulation Model Portability Handbook
EWP-2080, Issue 1.1, 31-Oct-2000

RD-2 SMP2 Alpha Specification
SIM-GST-TN-0045-TOS-GIC, Issue 1.0, 30-Dec-2003

RD-3 UML 2.0 Infrastructure Final Adopted Specification,
OMG Document ptc/03-09-15 at http://www.omg.org/uml/

RD-4 UML 2.0 Superstructure Final Adopted Specification,
OMG Document ptc/03-08-02 at http://www.omg.org/uml/

RD-5 UML 2.0 Diagram Interchange Final Adopted Specification,
OMG Document ptc/03-09-01 at http://www.omg.org/uml/

RD-6 UML 2.0 Object Constraint Language (OCL) Final Adopted Specification,
OMG Document ptc/03-10-14 at http://www.omg.org/uml/

RD-7 XASTRO Framework and Model Prototype
DTOS-SST-TN-0691-TOS-GIC, Issue 3.0, 26 June 2003

RD-8 EGOS Modelling Framework (EGOSMF) Specification –
Part 1: Foundations, Architecture Modelling and UML Profiles
EGOS-XAS-XASTRO-TN-00XX-ESOC, Issue 1.1, 28 February 2005

1.5 Overview

The SMP2 Metamodel provides mechanisms to:

1. Define model types (SMDL Catalogues)
2. Integrate model instances (SMDL Assemblies)
3. Schedule assemblies (SMDL Schedules)
4. Define packaging of implementations (SMDL Packages)
5. Define grouping of SMDL documents (SMDL Workspaces)

These mechanisms depend on each other: The definition of model types in a catalogue defines how model instances can later be connected in an assembly. The model instances available in an assembly and their type definition in the corresponding catalogue(s) limits how these models can later be scheduled. Each implementation element in a package references a type in a catalogue, and a workspace includes references to any of the other four document formats.

All document types make use of common elements, which have been separated out and put into a dedicated XML schema (Core Elements) that is included by all other schema documents.

The remainder of this document is split into the following sections:

2 SMP2 Metamodel

This section summarises the SMP2 Metamodel, names the applied technologies, and provides a diagram legend and colouring scheme used throughout the following sections.

3 Core Elements

The section defines basic simple and complex types used throughout the Metamodel.

4 Core Types

This section defines base types, mechanism to create new types from existing ones, and typed elements referencing these types.

5 SMDL Catalogues

This section describes all Metamodel elements that are needed in order to define models in a catalogue.

6 SMDL Assemblies

This section describes all Metamodel elements that are needed in order to define an assembly of model instances.

7 SMDL Schedules

This section describes all Metamodel elements that are needed in order to define how the model instances in an assembly are to be scheduled.

8 SMDL Packages

This section describes all Metamodel elements that are needed in order to define how implementations of types defined in catalogues are packaged.

9 SMDL Workspaces

This section describes all Metamodel elements that are needed in order to define a workspace of other documents that belong together.

10 Appendix A: XML Links and Primitive Types

This section describes XML linking mechanisms as well as primitive types used within schemas.

11 Appendix B: XML Schemas

This section lists the XML schemas that have been generated from the UML representation of the Metamodel. The schemas are *normative* in that they define the exact syntax of SMDL Catalogue, Assembly and Schedule files.

2. SMP2 METAMODEL

This section summarises the SMP2 Metamodel, names the applied technologies, and provides a diagram legend and colouring scheme used throughout the following sections.

2.1 Overview

The Simulation Model Definition Language is the language used to describe SMP2 compliant types (including models), assemblies and schedules. While the SMP2 Component Model defines mechanisms for the interaction of SMP2 Models and other SMP2 Components, this language forms the basis for the collaboration of SMP2 compliant tools.

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 parts, 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. In addition, a package defines grouping of type implementations, and a workspace links corresponding documents together.

Figure 2-1 shows the top-level structure of the SMP2 Metamodel.

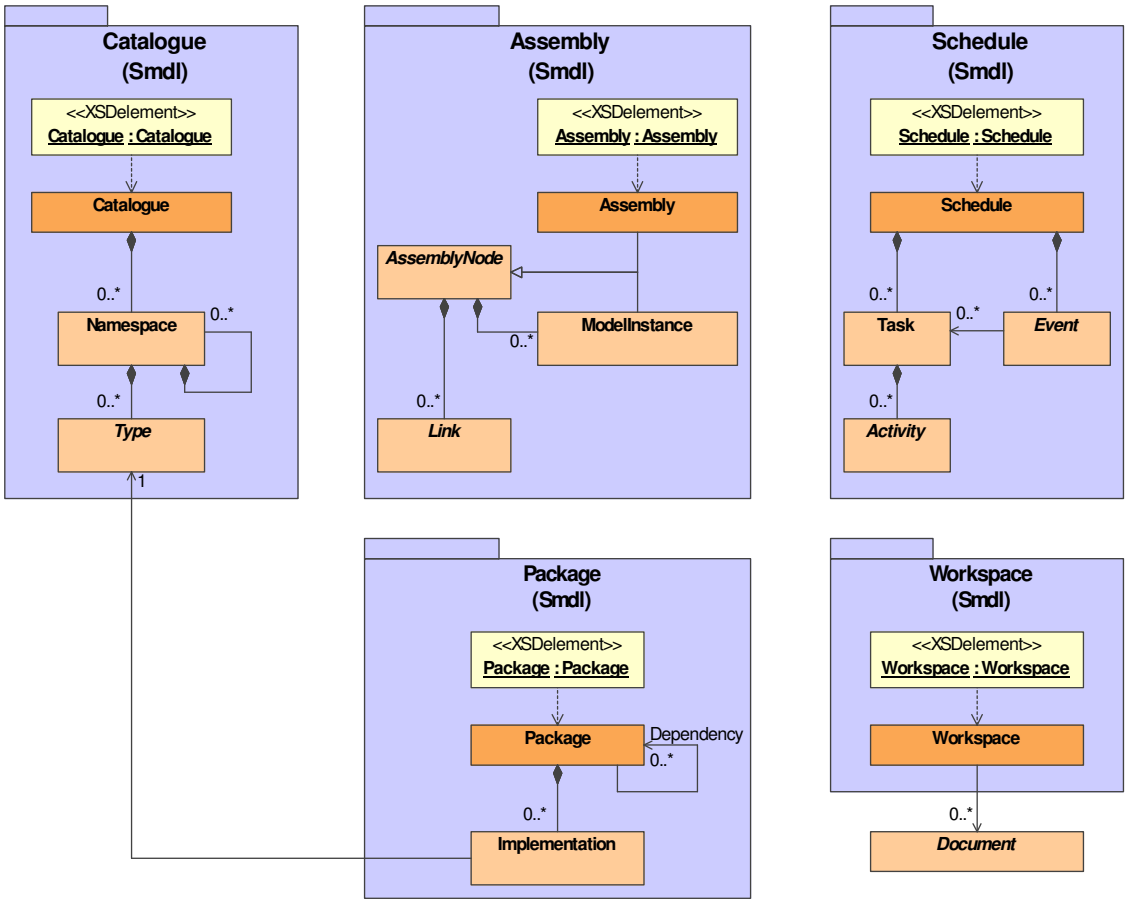

Figure 2-1: Top-Level Structure of the SMP2 Metamodel

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.

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.

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.

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.

A workspace links several other SMDL documents together. It is provided for tool support, to allow linking related documents together and e.g. to load them all into an editor with a single file selection operation.

2.2 Applied Technologies

This section describes relevant technologies applied to specify the Metamodel.

The SMP2 Metamodel is specified in UML as modelling language using the Magic Draw UML 9.5 tool. The output of this tool is a native XML Metadata Interchange (**XMI**) 1.2 file, which is the direct input to the XML Schema Converter based on XSLT 2.0. This converter was developed within the XASTRO project, compare [RD-7], and has been refined within the XASTRO-2 project [RD-8].

In order for the converter to be able to create XML schemas from a UML model, the UML model has to be decorated with stereotypes and tagged values, which are defined in a UML profile. Therefore, a UML Profile for XML Schema has been defined; see [RD-8] for details.

Summarising, the SMP2 Metamodel is specified in UML and represented in XML Schema by an automated mapping process. This allows describing SMDL entities like catalogues and assemblies in XML documents, which can be validated against the auto-generated schemas derived from the Metamodel specified in UML.

2.3 Diagram Legend and Colouring Scheme

In the UML class diagrams used in this document, the following colouring scheme is applied in order to get an immediate impression of the purpose of the element. Note that the colours are not part of the model itself, but they are only applied to the Metaclasses in order to improve readability.

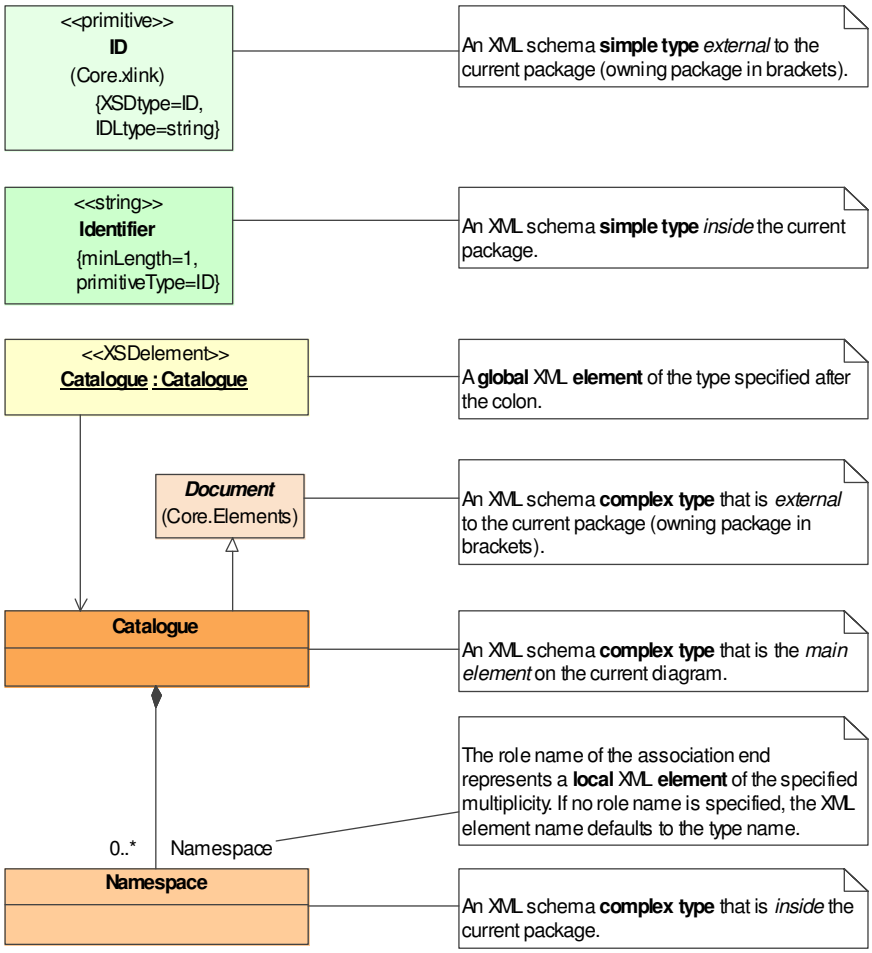

Figure 2-2: Diagram Legend and Colouring Scheme

2.4 Metamodel Schemas

Figure 2-3 below shows the SMP2 Metamodel, which 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.

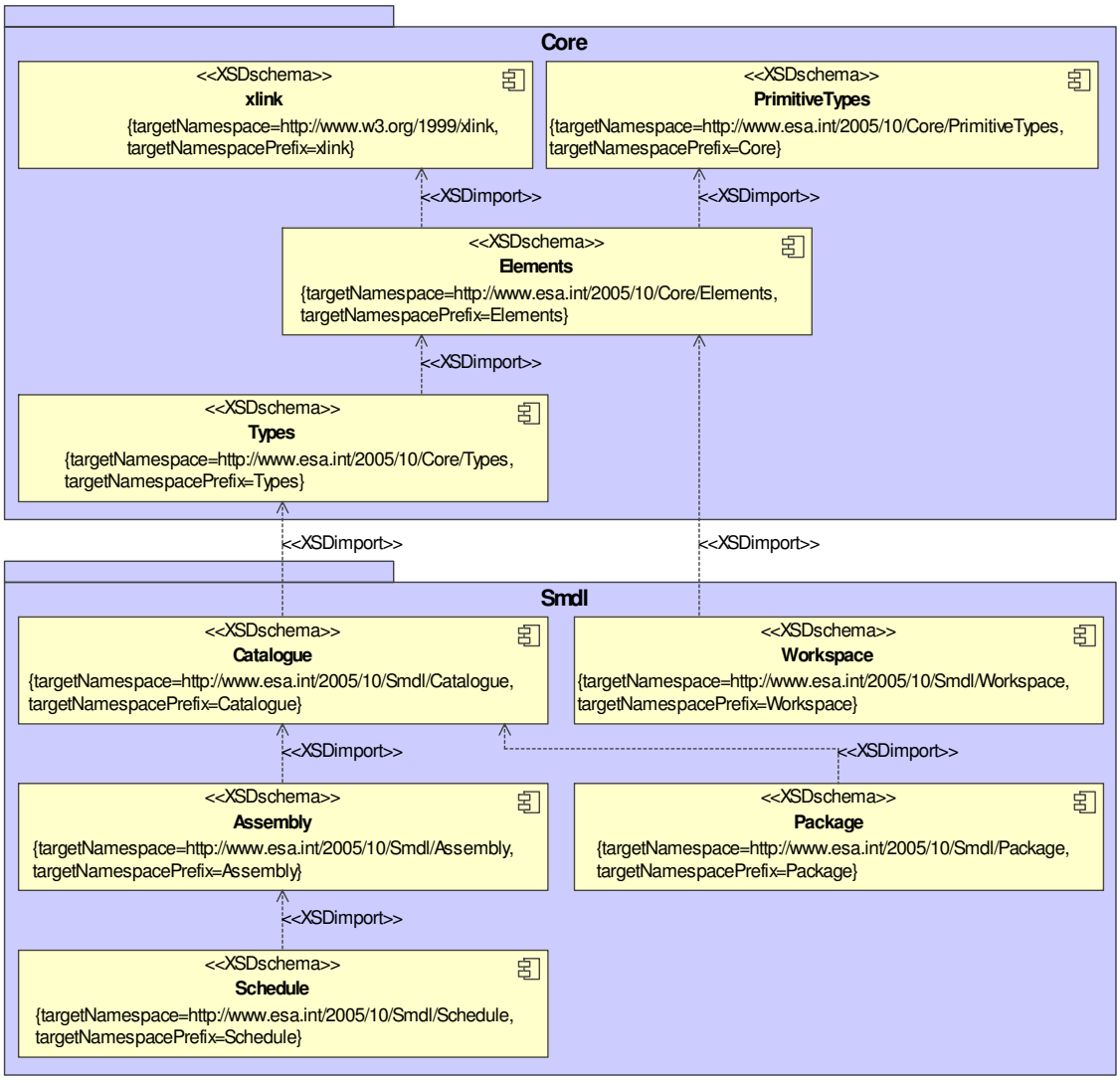

Figure 2-3: SMP2 Metamodel Schemas

A summary of the XML namespaces is given in Table 2.1. UML dependencies between packages decorated with the `<<XSDimport>>` stereotype are mapped to corresponding import statements in XML schema.

Table 2.1: SMP2 Metamodel Schemas and Namespaces

Schema	Default Namespace Prefix	Namespace URI
`Core.PrimitiveTypes`	`Core`	http://www.esa.int/2005/10/Core/PrimitiveTypes
`Core.Elements`	`Elements`	http://www.esa.int/2005/10/Core/Elements
`Core.Types`	`Types`	http://www.esa.int/2005/10/Core/Types

Schema	Default Namespace Prefix	Namespace URI
`Smdl.Catalogue`	`Catalogue`	http://www.esa.int/2005/10/Smdl/Catalogue
`Smdl.Assembly`	`Assembly`	http://www.esa.int/2005/10/Smdl/Assembly
`Smdl.Schedule`	`Schedule`	http://www.esa.int/2005/10/Smdl/Schedule
`Smdl.Package`	`Package`	http://www.esa.int/2005/10/Smdl/Package
`Smdl.Workspace`	`Workspace`	http://www.esa.int/2005/10/Smdl/Workspace
`xlink`	`xlink`	http://www.w3.org/1999/xlink

The following sections present the individual schemas of the SMP2 Metamodel.

2.5 XML Links

SMDL makes use of the XML Linking Language (XLink) for two purposes. First, the `Documentation` Metaclass makes direct use of simple links to reference external resources (see section 3.3.2). Second, simple links are used within the SMDL schemas as referencing mechanism. That is, all directed associations shown in UML diagrams are mapped to the `SimpleLink` type on the XML Schema level. All mechanisms to support XML linking are held in the `xlink` schema. See Appendix A (section 10.1) for details on XML linking.

2.6 Primitive Types

SMDL defines a number of primitive types in order to bootstrap the type system (section 4.2.2). To ensure consistency, the SMDL schemas are based on the same set of primitive types, which have well-defined mappings into XML Schema and other target platforms, including CORBA IDL and ANSI/ISO C++. The primitive types are held in the `Core.PrimitiveTypes` schema.

Note, however, that additional primitive types (such as `String8`, `AnySimple`) are used within the Metamodel specification (i.e. within the SMDL schemas) and the Component Model specification, which are not part of the SMDL primitive types (section 4.2.2). The latter can be used in SMP2 models. See Appendix A (section 10.2) for details on primitive types.

This Page is Intentionally left Blank

3. CORE ELEMENTS

This section defines base Metaclasses and annotation mechanisms used throughout the Metamodel.

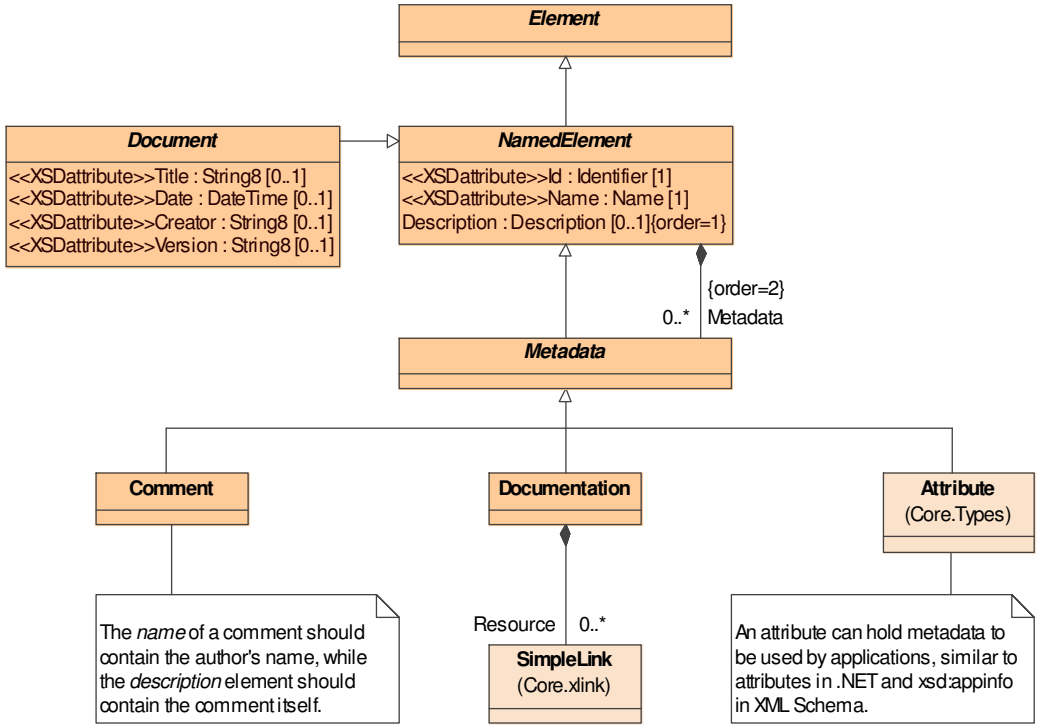

Figure 3-1: The `Core.Elements` Schema

3.1 Simple Types

The `Core.Elements` schema defines some simple types. Types marked with the `<<string>>` stereotype are string types (derived from `String8`, which is based on `xsd:string` in XML Schema, see section 10.2 for details), and may use a `pattern` tag to specify a regular expression that specifies the value space and a `maxLength` tag to specify a maximum length.

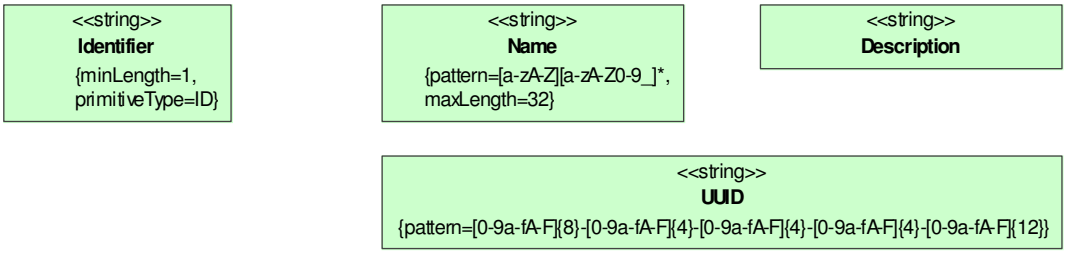

Figure 3-2: Simple Types of the SMP2 Metamodel (`Core.Elements` Schema)

In the following we give a short description for each simple type introduced in Figure 3-2.

3.1.1 Identifier

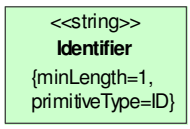

Figure 3-3: Identifier

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 `Element` Metaclass (see below). An identifier must not be empty, which is indicated by the `minLength` tag.

3.1.2 Name

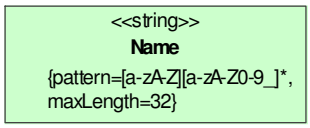

Figure 3-4: Name

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 ("_"). Further, a name is limited to a maximum of 32 characters.

3.1.3 Description

Figure 3-5: Description

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

3.1.4 UUID

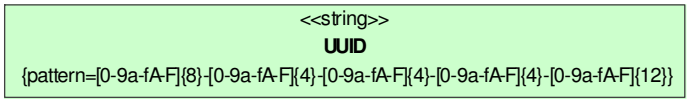

Figure 3-6: Universally Unique Identifier (UUID)

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 (see section 4).

3.2 Elements

The `Core.Elements` schema defines as well some basic element types of the SMP2 Metamodel.

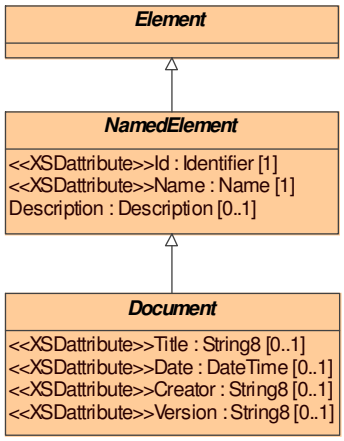

Figure 3-7: Elements of the SMP2 Metamodel (`Core.Elements` Schema)

3.2.1 Element

Figure 3-8: Element

The Metaclass `Element` is the common base for almost all other language elements. It serves as an abstract base class and does not introduce any attributes.

3.2.2 Named Element

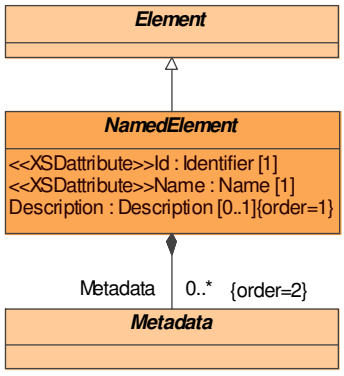

Figure 3-9: Named Element

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.

3.2.3 Document

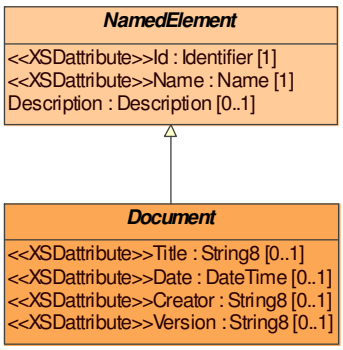

Figure 3-10: Document

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.

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.

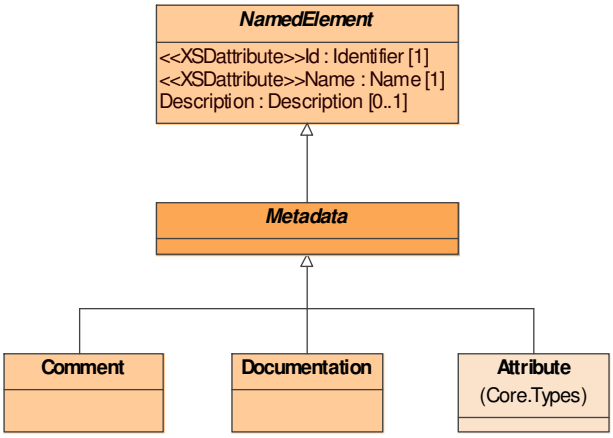

Figure 3-11: Metadata

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.

3.3.1 Comment

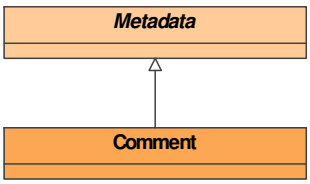

Figure 3-12: Comment

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`.

3.3.2 Documentation

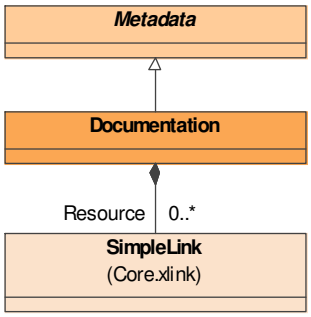

Figure 3-13: Documentation

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

This Page is Intentionally left Blank

4. CORE TYPES

The `Core.Types` Schema defines base types, mechanism to create new types from existing ones, and typed elements referencing these types. As SMDL shares these type definition mechanisms with other projects (namely with the XASTRO-2 project [RD-8]), they have been separated out from the SMDL Catalogue schema.

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 (section 5).

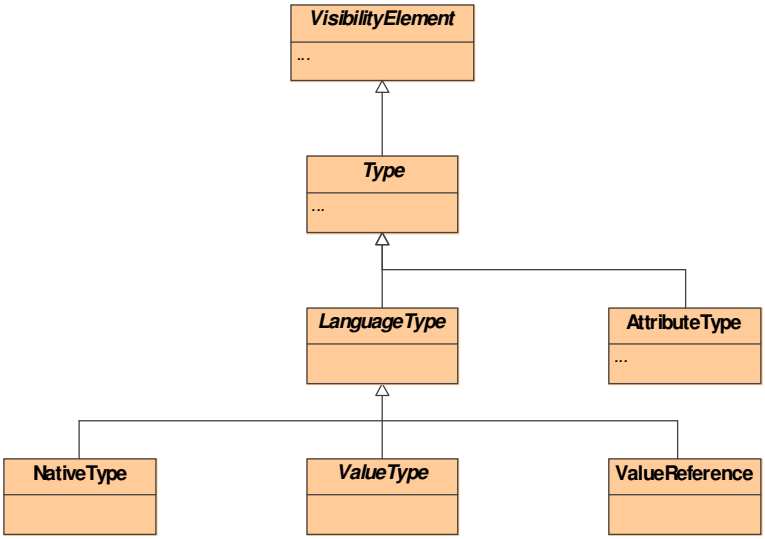

Figure 4-1: Types

4.1.1 Visibility Element

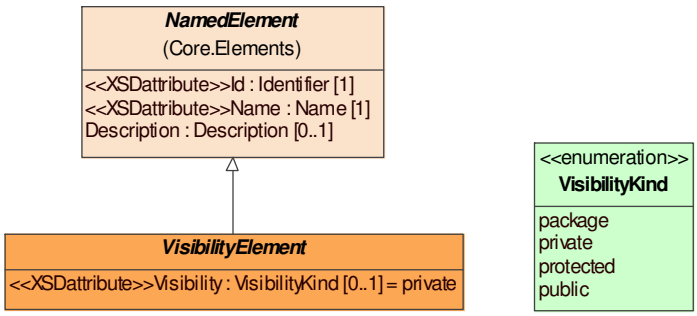

Figure 4-2: Visibility Element

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`).

4.1.2 Type

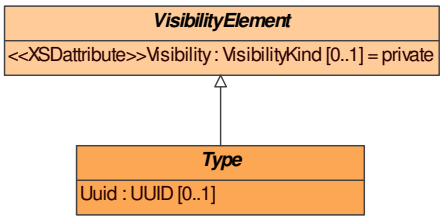

Figure 4-3: Type

A `Type` is the abstract base Metaclass for all type definition constructs specified by SMDL. A type may have a `Uuid` element representing a Universally Unique Identifier (**UUID**) as defined in section 3.1.4. This is needed such that implementations may reference back to their specification without the need to directly reference an XML element in the catalogue.

4.1.3 Language Type

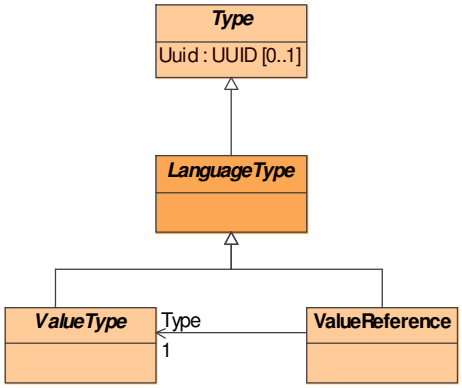

Figure 4-4: Language Type

A `LanguageType` is the abstract base Metaclass for value types (where instances are defined by their value), and references to value types. Please note that SMDL Catalogues define reference types (where instances are defined by their reference, i.e. their location in memory) as being derived from language type as well. As reference types are specific to software systems, they are not introduced in `Core.Types`.

4.1.4 Value Type

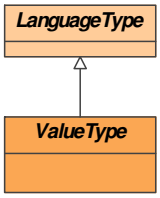

Figure 4-5: Value Type

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. See section 4.2 for details.

4.1.5 Value Reference

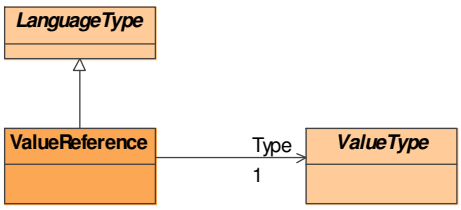

Figure 4-6: Reference

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

4.2 Value Types

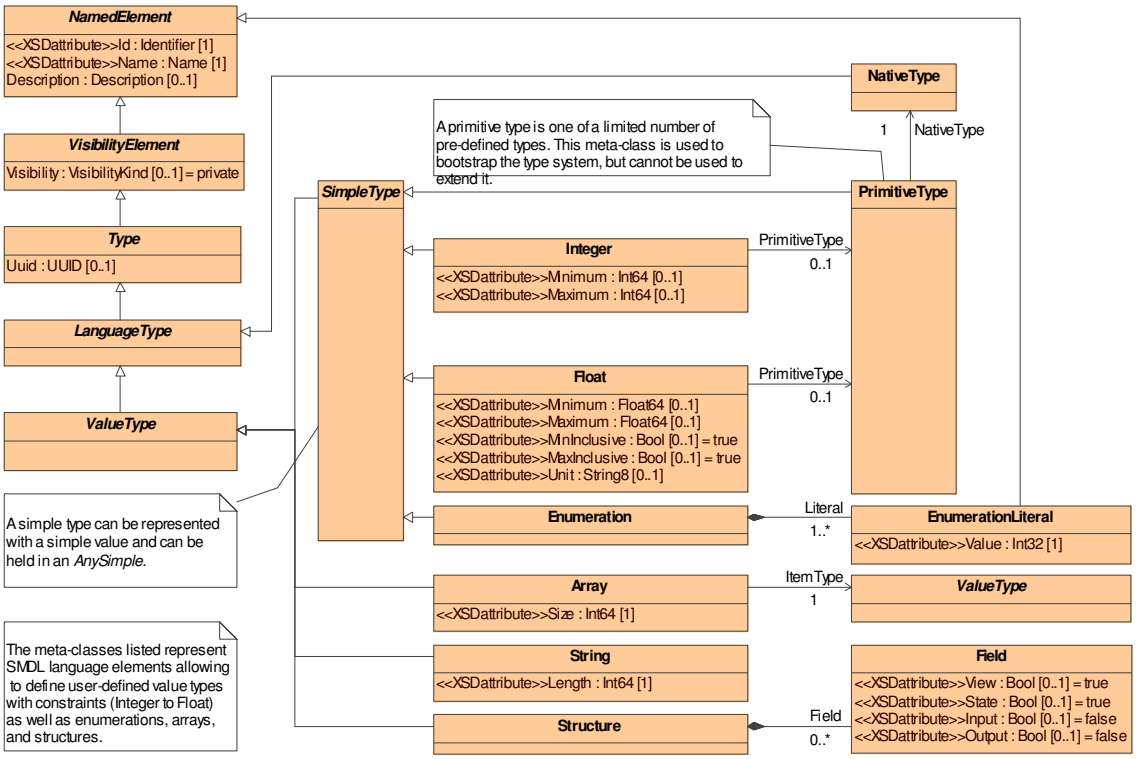

Figure 4-7: Value Types (`Core.Types` Schema)

Figure 4-7 shows the elements available to define value types. Note that 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 Metaclass `PrimitiveType` has been introduced to allow defining the available base types of SMDL, and is only used internally. Further, the `NativeType` is a generic mechanism to specify native platform specific types, which is also used to define the platform mappings of the SMP2 primitive types within the Component Model.

4.2.1 Native Type

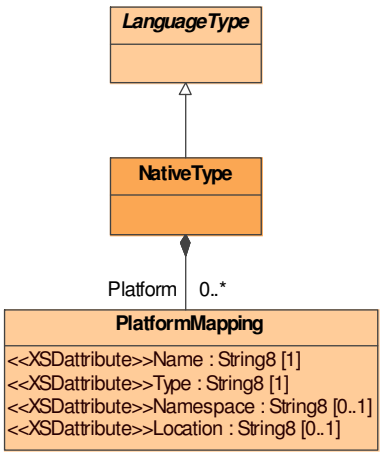

Figure 4-8: Native Type

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 SMP2 specification to define the SMDL primitive types with respect to the SMP2 Metamodel, but it can also be used to define native types within an arbitrary SMDL catalogue for use by SMP2 models. In the latter case, native types are typically used to bind an SMP2 model to some external library or existing Application Programming Interface (**API**).

> Example: An SMP2 model may need to communicate via stream I/O, which would be based on the standard `iostream` library in C++. Using a textual notation (instead of XML), a native type could be defined as follows:
>
> NativeType.Name = “FileStream”
> NativeType.Description = “File-based stream I/O”
> NativeType.PlatformMapping[0] =
> {Name=”cpp”, Type=”fstream”, Namespace=”std”, Location=”fstream”}
>
> This would define a platform mapping for the C++ platform, using the platform specific type “fstream” which is in the “std” namespace, i.e. `std::fstream` in C++, and is located in “fstream”, i.e. requires an `#include <fstream>` statement in C++.

4.2.1.1 Platform Mapping

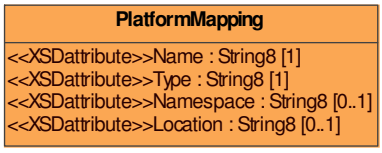

Figure 4-9: Platform Mapping

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 `<language>_<environment>`, where the environment is optional and may be split into `<os>_<compiler>`. Some examples are:

- `cpp` Standard ANSI/ISO C++ (for all environments)
- `cpp_windows_vc71` C++ using Microsoft VC++ 7.1 under Windows
- `cpp_linux_gcc33` C++ using GNU Gcc 3.3 under Linux
- `idl` CORBA IDL
- `xsd` XML Schema
- `java14` Java 1.4.x
- `java15` Java 1.5.x

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.

4.2.2 Simple Type

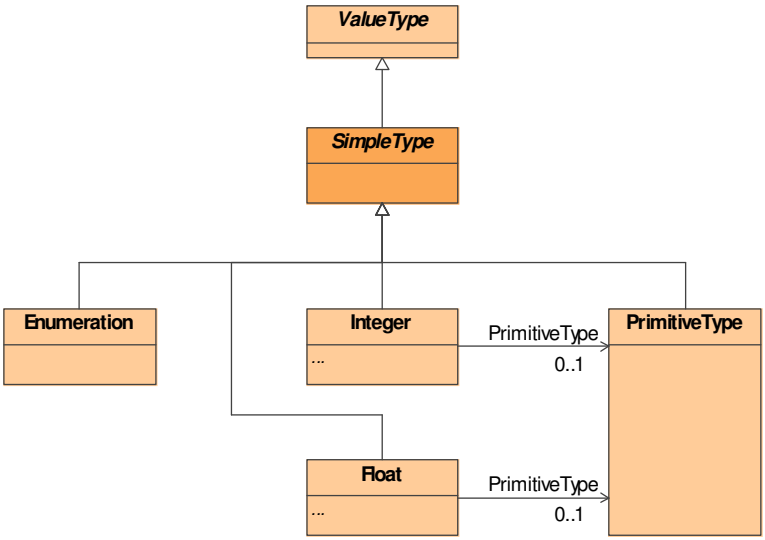

Figure 4-10: Simple Type

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`, and `Float` types.

4.2.3 Primitive Type

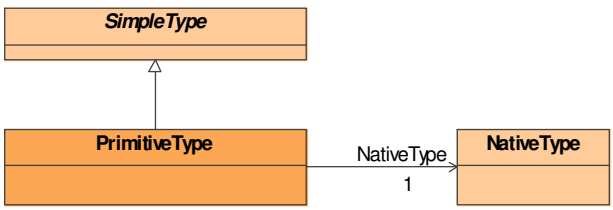

Figure 4-11: Primitive Type

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`. A primitive type references a `NativeType`, which specifies the platform mapping.

This mechanism is only used in order to bootstrap the type system and may *not* be used to define new types for SMP2 modelling. This is an important restriction, as all values of primitive types may be held in a `SimpleValue` (section 4.4.2). This definition, however, needs to be pre-defined and cannot be extended.

4.2.4 Primitive Types

Figure 4-12 shows the 14 standard SMDL primitive types that may be used for SMP2 modelling.

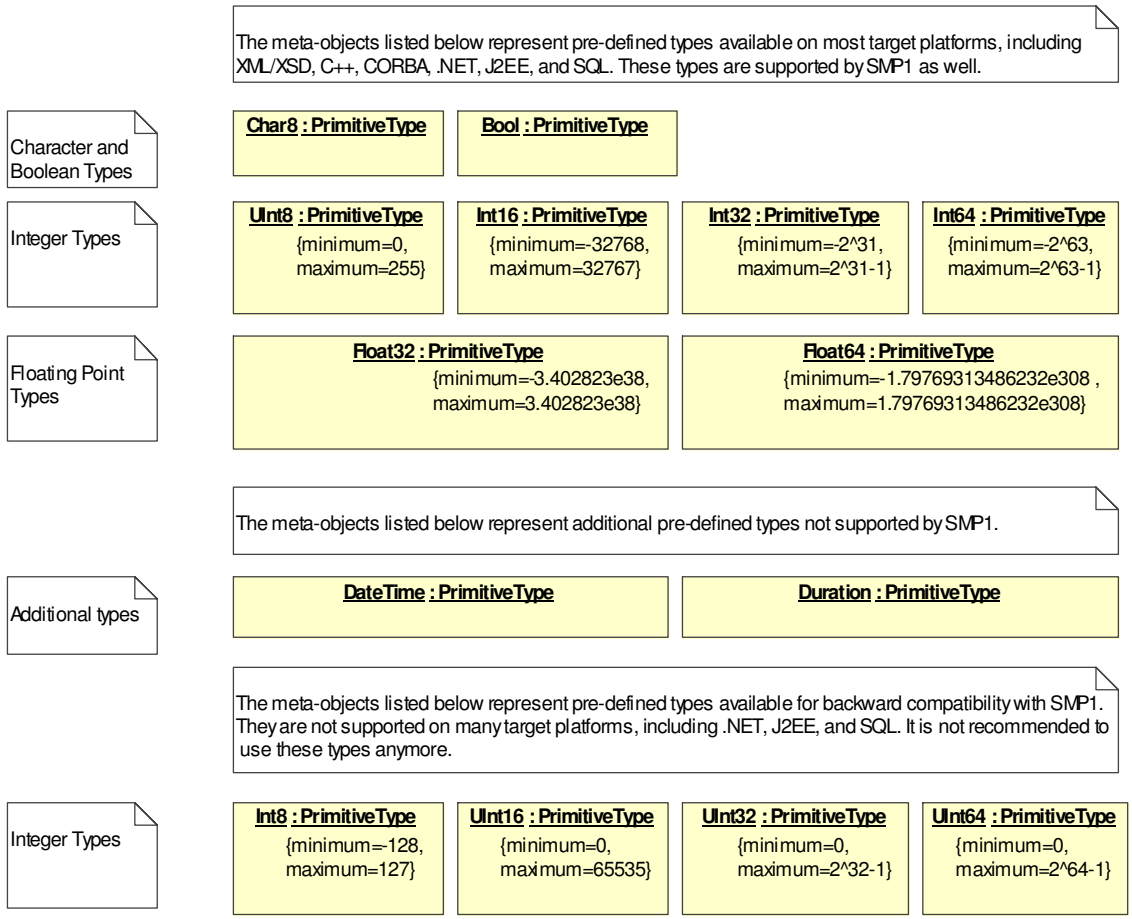

Figure 4-12: Primitive Types

Within type definitions in an SMDL Catalogue (compare section 5), the SMDL primitive types shown in Figure 4-12 shall be referenced using the (virtual) catalogue URI http://www.esa.int/2005/10/Smp. That is, a reference to a primitive type within an SMDL catalogue document is specified by an XML link, where the `xlink:href` attribute is set to

"http://www.esa.int/2005/10/Smp#<Type>",

for example "http://www.esa.int/2005/10/Smp#Int16" for the `Int16` primitive type.

Please note that it is not recommended using `Int8`, `UInt16`, `UInt32`, and `UInt64` anymore, as these types are not supported by some platforms, including the Structured Query Language (**SQL**), the Java 2 Enterprise Edition (**J2EE**), or the Microsoft .NET Common Language Specification (**CLS**).

Table 4.1 lists the primitive types supported by SMDL, their mapping to SMP1/SMI types, and their mapping to XML Schema types. All basic types from SMP1/SMI are supported for compatibility reasons.

Table 4.1: Primitive Types

Id	Description	SMI Type	XSD Type
`Char8`	8 bit unsigned character type	`Character_t`	`string`
`Bool`	Boolean type (values true or false)	`Boolean_t`	`boolean`
`Int8`	8 bit signed integer type	`Integer8_t`	`byte`
`Int16`	16 bit signed integer type	`Integer16_t`	`short`
`Int32`	32 bit signed integer type	`Integer32_t`	`int`
`Int64`	64 bit signed integer type	`Integer64_t`	`long`
`UInt8`	8 bit unsigned integer type	`Unsigned8_t`	`unsignedByte`
`UInt16`	16 bit unsigned integer type	`Unsigned16_t`	`unsignedShort`
`UInt32`	32 bit unsigned integer type	`Unsigned32_t`	`unsignedInt`
`UInt64`	64 bit unsigned integer type	`Unsigned64_t`	`unsignedLong`
`Float32`	IEEE 754 single prec. float type	`SingleFloat_t`	`float`
`Float64`	IEEE 754 double prec. float type	`DoubleFloat_t`	`double`
`DateTime`	Date and time type (resolution 1 nanosecond)	–	`dateTime`
`Duration`	Duration type (in nanoseconds)	–	`duration`

See the C++ Mapping [AD-3] for their mapping to the C++ programming language.

4.2.4.1 DateTime

This primitive type is used to store date and time information. When mapped to a non-XML platform (CORBA IDL, C++), it is stored as a signed 64-bit integer value (`Int64`), with the following interpretation:

- The number stored corresponds to the number of ticks relative to a reference date.
- The reference date defaults to 01.01.2000, 12:00:00 (MJD 2000+0.5).
- A tick is a nanosecond (10^{-9} s), which is the lowest level of granularity available.
- A positive number corresponds to a date after the reference date, while negative numbers store dates before the reference date.

In XML Schema, a value is serialized into an `xsd:dateTime`.

With this definition, `DateTime` is compatible with `Duration` as defined below.

4.2.4.2 Duration

This simple type is used to store duration information. When mapped to a non-XML platform (CORBA IDL, C++), it is stored as a signed 64-bit integer value (`Int64`), with the following interpretation:

- The number stored corresponds to the number of ticks relative to zero.
- A tick is a nanosecond (10^{-9} s), which is the lowest level of granularity available.
- A positive number corresponds to a positive duration, while negative numbers store negative durations.

In XML Schema, a value is serialized into an `xsd:duration`.

With this definition, `Duration` is compatible with `DateTime` as defined above.

4.2.5 Enumeration

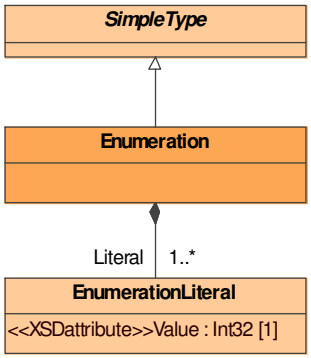

Figure 4-13: Enumeration

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.

4.2.5.1 Enumeration Literal

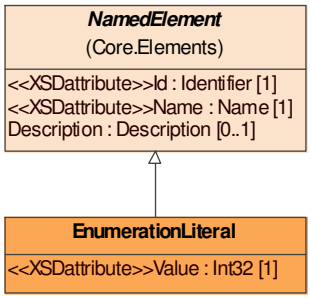

Figure 4-14: Enumeration Literal

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

4.2.6 Integer

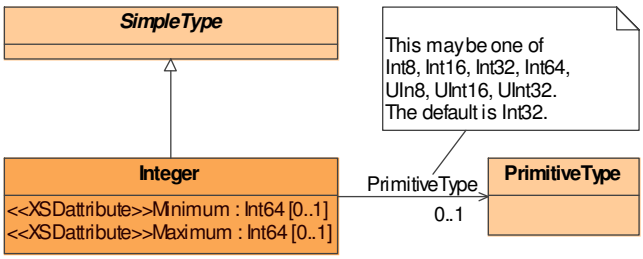

Figure 4-15: Integer

An `Integer` type represents integer values with a given range of valid values (via the `Minimum` and `Maximum` attributes). Optionally, the `PrimitveType` used to encode the integer value may be specified (one of `Int8`, `Int16`, `Int32`, `Int64`, `UIn8`, `UInt16`, `UInt32`, where the default is `Int32`).

4.2.7 Float

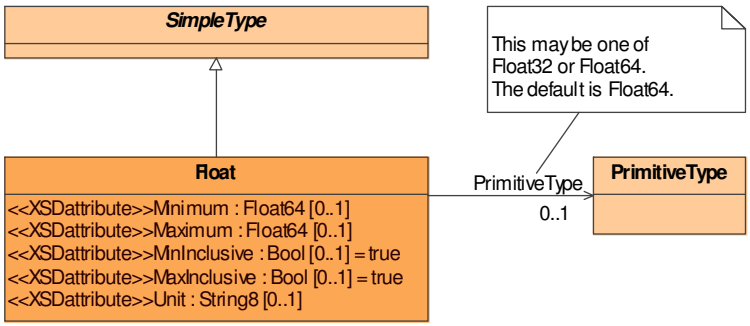

Figure 4-16: Float

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. Furthermore the `Unit` element can hold a physical unit that can be used by applications to ensure physical unit integrity across models. Optionally, the `PrimitveType` used to encode the floating-point value may be specified (one of `Float32` or `Float64`, where the default is `Float64`).

4.2.8 Array

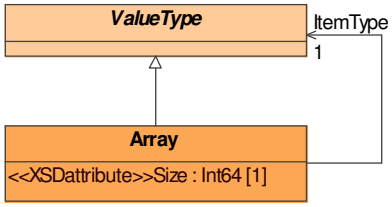

Figure 4-17: Array

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.

Dynamic arrays are not supported by SMDL, as they are not supported by some potential target platforms, and introduce various difficulties in memory management. Nevertheless, specific mechanisms are available to allow dynamic collections of components, either for containment (composition) or references (aggregation). See 5.3.3 for the available concepts.

4.2.9 String

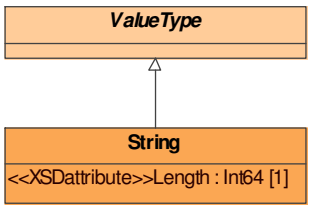

Figure 4-18: String

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). Nevertheless, strings are used internally for names, which are supposed to be constant during a simulation.

4.2.10 Structure

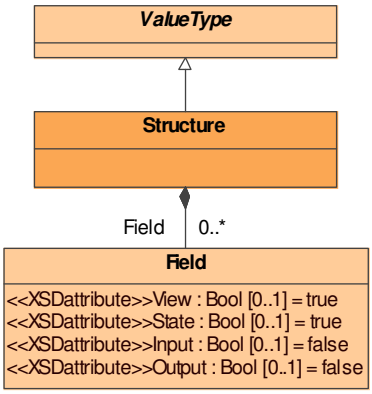

Figure 4-19: Structure

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.

4.3 Typed Elements

Figure 4-20 presents a number of typed elements that are used in the sequel. The most common typed elements are field and operation.

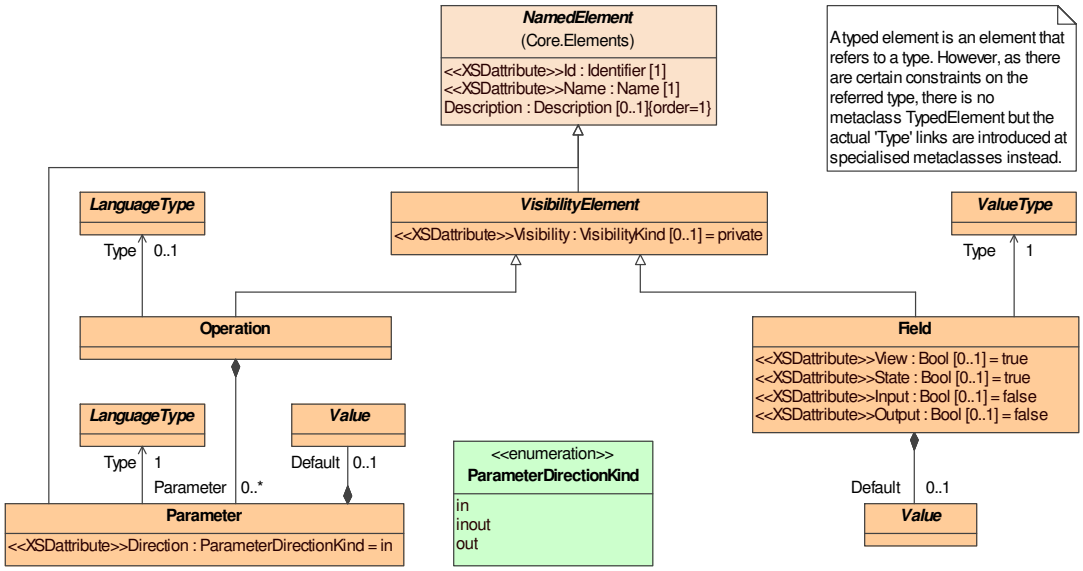

Figure 4-20: Typed Elements (`Core.Types` Schema)

A typed element is an element that refers to a type. However, as there are certain constraints on the referred type, there is no Metaclass *TypedElement* but the actual `Type` links are introduced at specialised Metaclasses instead.

4.3.1 Field

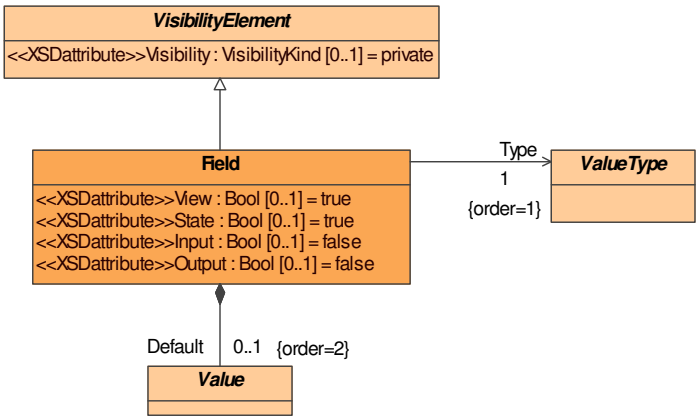

Figure 4-21: Field

A `Field` is a feature that is typed by a value type, and that may have a `Default` value. See section 4.4 for details about values. The `View` and `State` attributes define how the field is published to the simulation environment. Only fields with a `View` value of `true` are visible in the Run-Time Environment. Only fields with a `State` of `true` are stored using external persistence. If both flags are set to `false`, then the field may not be published at all. By default, both attributes are set to `true`.

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.

4.3.2 Operation

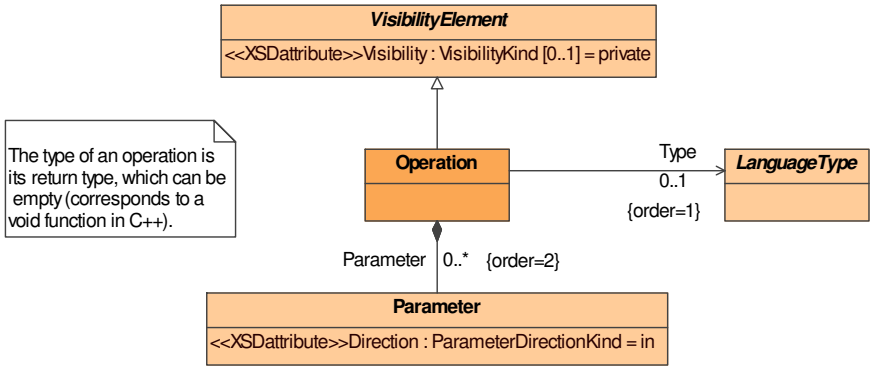

Figure 4-22: Operation

An `Operation` may have an arbitrary number of parameters, and the `Type` is its return type. If the type is absent, the operation is a void function (procedure) without return value.

4.3.2.1 Parameter

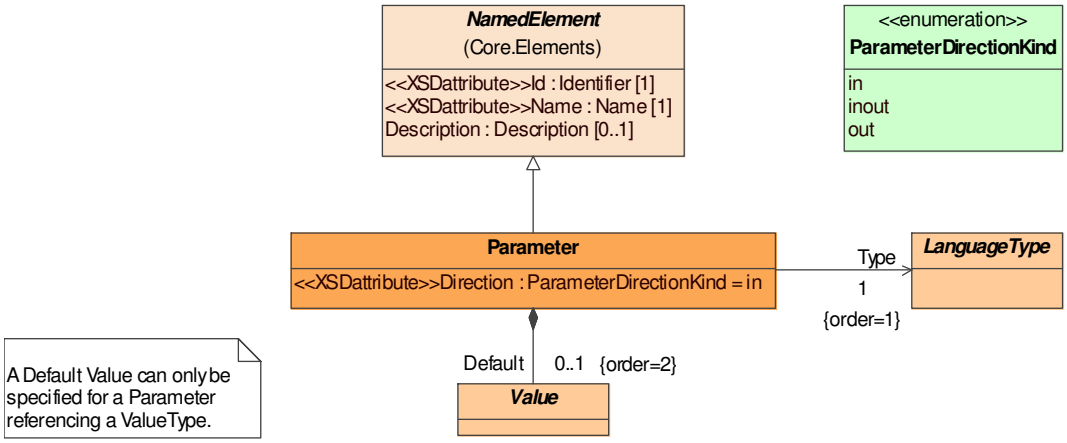

Figure 4-23: Parameter

A `Parameter` has a type and a `Direction`. The parameter direction may be

- `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, or
- `out`: the parameter is write-only to the operation, i.e. its value is unspecified on call, and must be set by the operation; or
- `inout`: the parameter must be specified on call, and may be changed by the operation.

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

4.4 Values

A `Value` represents the state of a `ValueType`. For each Metaclass derived from `ValueType`, a corresponding Metaclass derived from `Value` has been 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`. Figure 4-24 shows the metamodel values.

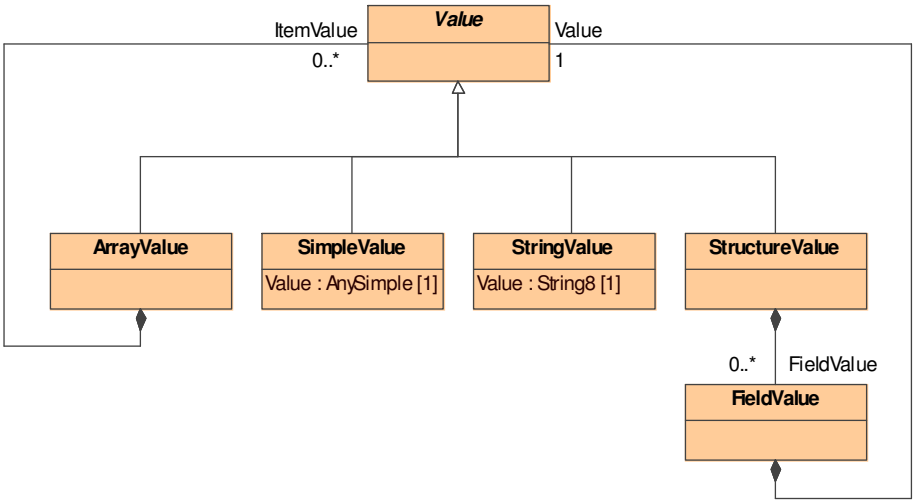

Figure 4-24: Values (`Core.Types` Schema)

4.4.1 Value

The `Value` Metaclass is an abstract base class for specialised values.

4.4.2 Simple Value

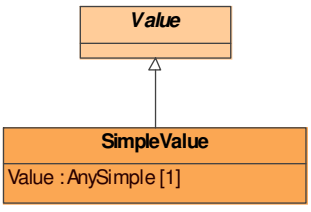

Figure 4-25: Simple Value

The `SimpleValue` Metaclass is used for values of simple types, i.e. primitive types as well as user-defined `Integer`, `Float`, and `Enumeration` types. As each simple type corresponds to a different XML type, the `Value` element is typed by an `AnySimple` type. In XML, the concrete type has to be specified via a standard `xsi:type` attribute, which may be one of `xsd:string`, `xsd:boolean` `xsd:byte`, `xsd:short`, `xsd:int`, `xsd:long`, `xsd:unsignedByte`, `xsd:unsignedShort` `xsd:unsignedInt`, `xsd:unsignedLong`, `xsd:float`, `xsd:double`, `xsd:dateTime`, or `xsd:duration`. This list is the union of all XML Schema mappings of SMDL primitive types (compare sections 4.2.4 and 10.2).

4.4.3 Array Value

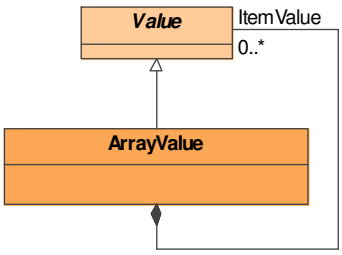

Figure 4-26: Array Value

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

4.4.4 String Value

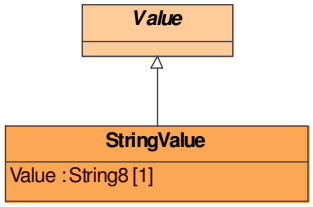

Figure 4-27: String Value

A `StringValue` holds a value for a string, represented by the `Value` element. As opposed to the array value, the string value uses a single string instead of an array of values.

4.4.5 Structure Value

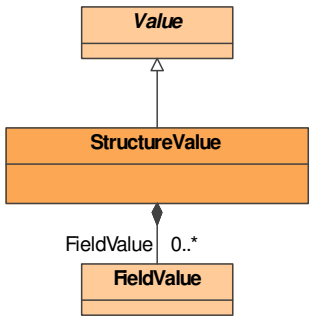

Figure 4-28: Structure Value

A `StructureValue` holds field values for all fields of the corresponding structure type.

4.4.5.1 Field Value

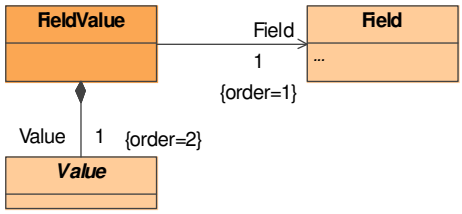

Figure 4-29: Field Value

A `FieldValue` links to the defining `Field` and contains a value holding the actual `Value`.

4.5 Attributes

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 (see section 4.3.1) in SMDL.

4.5.1 Attribute Type

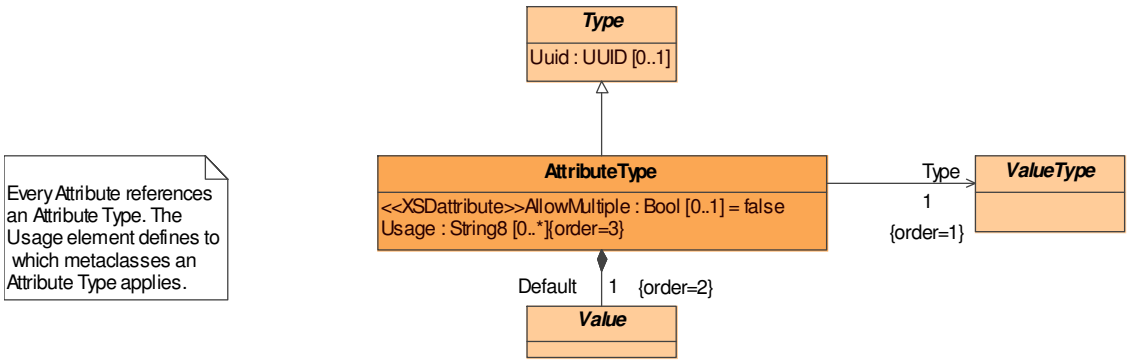

Figure 4-30: Attribute Type

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.

4.5.2 Attribute

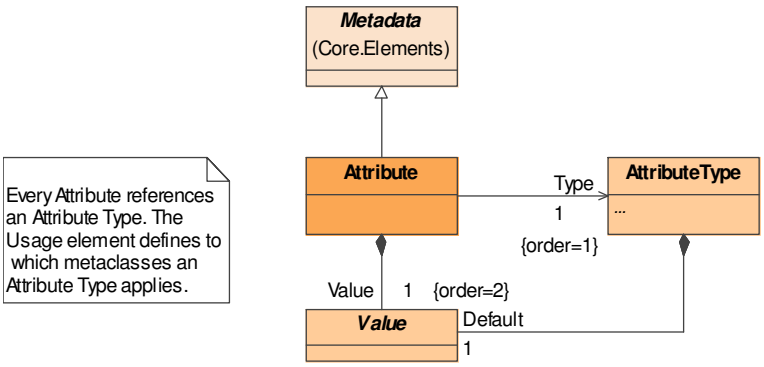

Figure 4-31: Attribute

An `Attribute` element holds name-value pairs allowing to attach user-defined metadata to any language element. This provides a similar mechanism as tagged values in UML, `xsd:appinfo` in XML Schema, or attributes in the .NET framework. A possible application of using attributes could be to decorate an SMDL model with information needed to guide a code generator, for example to taylor the mapping to C++.

5. SMDL CATALOGUES

This section 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, but they add interfaces, classes, and models, as well as a grouping mechanism (namespaces).

5.1 A Catalogue Document

Figure 5-1 shows the top-level structure of an SMDL catalogue document.

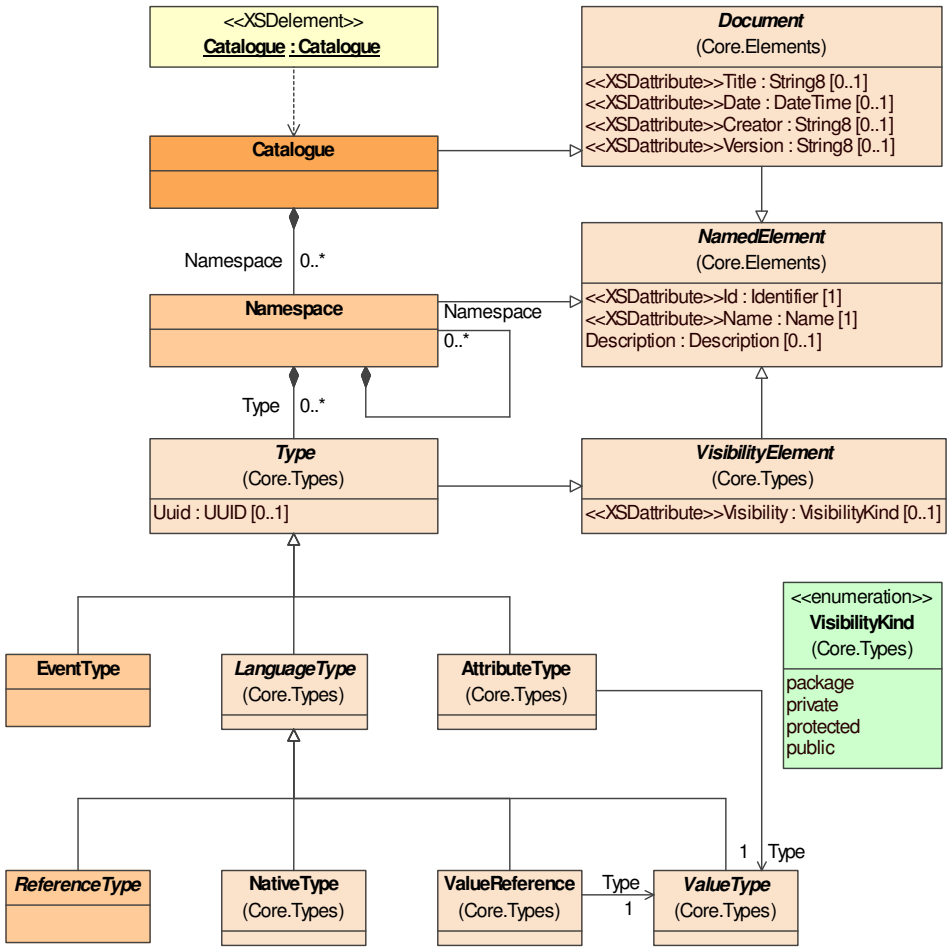

Figure 5-1: Catalogue Document (`Smdl.Catalogue` Schema)

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, classes, and models).

The Metaclasses introduced in Figure 5-1 are detailed below.

5.1.1 Catalogue

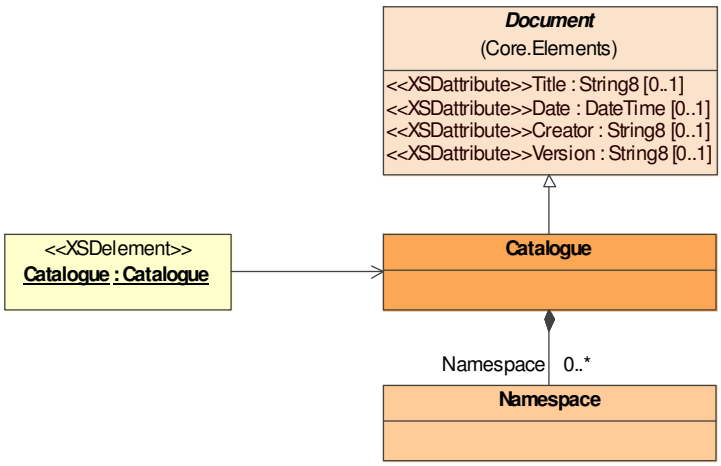

Figure 5-2: Catalogue

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.

5.1.2 Namespace

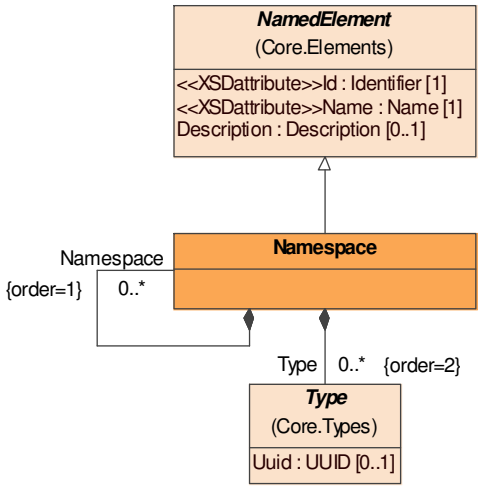

Figure 5-3: Namespace

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.

5.2 Classes

On top of the value types defined by the Core Types Schema, the SMDL Catalogue adds the `Class` Metaclass. This extends the `Structure` by the software specific features of inheritance and operations.

5.2.1 Class

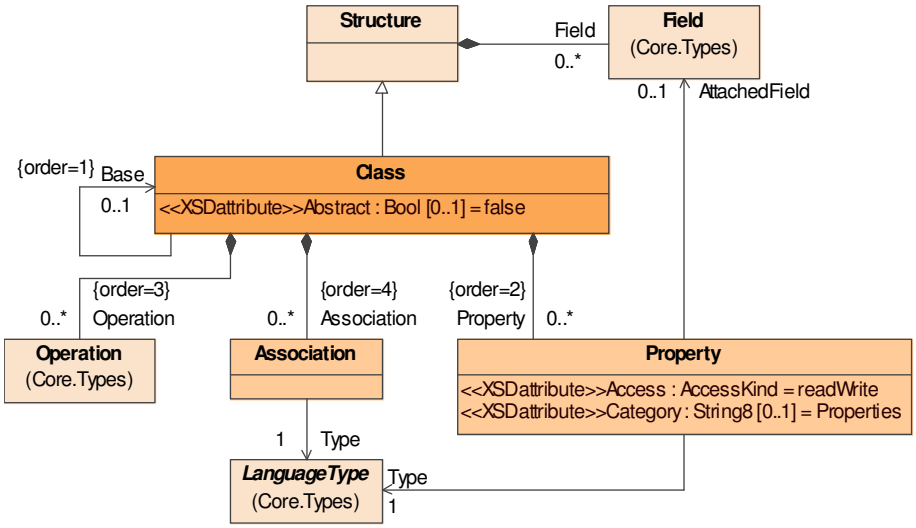

Figure 5-4: Class

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; see the discussion in [AD-1, Section 2.3.7].

In addition to the fields (`Field` elements, see 4.3.1) a class can contain because it is derived from structure, it can have arbitrary numbers of operations (`Operation` elements, see 4.3.2), properties (`Property` elements, see 5.2.2), and associations (`Association` elements, see 5.2.3).

> For example, a *ComplexNumber* could be modelled as a class, where the fields are the Real and the Imaginary part, and the operations are *Add, Subtract, Multiply, Divide*.

5.2.2 Property

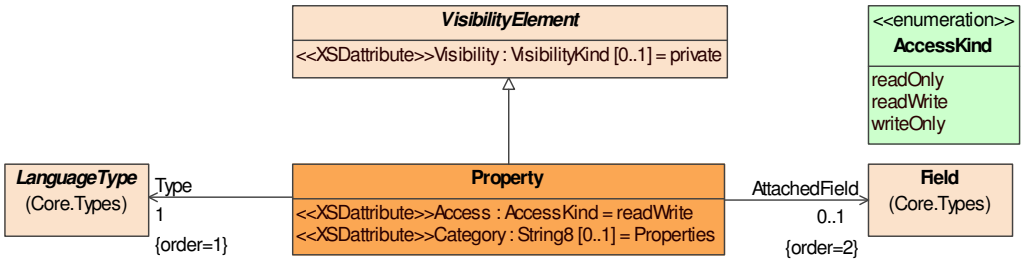

Figure 5-5: Property

A `Property` has a similar syntax as a `Field`: It is a `Feature` that references a `LanguageType`. However, the semantics is very different: A property can be assigned an `Access` attribute limiting access

to one of `readWrite`, `readOnly`, and `writeOnly`. Typically, this is mapped to one or two operations, called the "setter" and "getter" of the property. Note that the term *property* used here closely corresponds to the same term in the JavaBeans specification and in the Microsoft .NET framework, formally representing a getter or a setter function or both.

Furthermore, a `Property` can be assigned a `Category` attribute to be used as ordering or filtering criterion in applications, e.g. in a property grid.

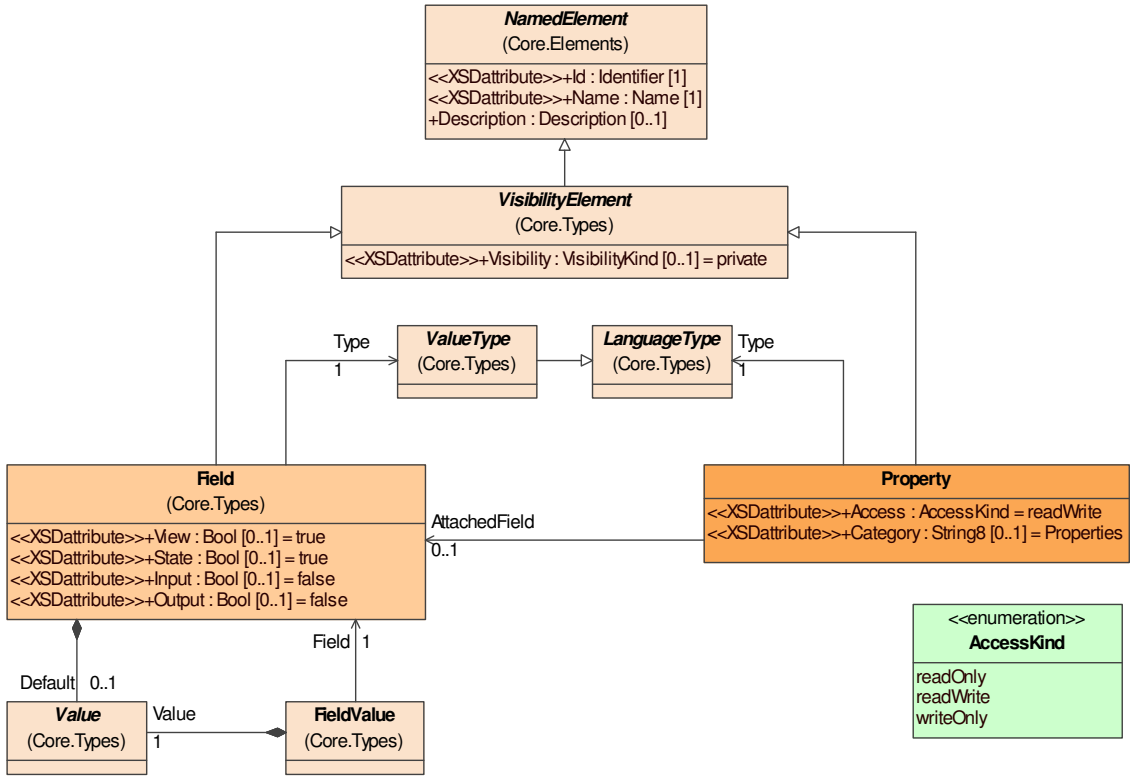

Figure 5-6: Field and Property

Note that 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.

For example, in C++, a property Mass (ignoring units) of a class Satellite may be represented as follows:

Table 5.1: Example Property in C++ (or similarly in Java)

Definition	Usage

`class Satellite`
`{`
`  public:`
`  double get_Mass();`
`  void set_Mass(double value);`
`  ...`
`};`
	`...`
	`Satellite sat = new Satellite();`
	`...`
	`sat.set_Mass(10.0);`
	`double mass = sat.get_Mass();`

In C#, the same property is represented as:

Table 5.2: Example Property in C#

Definition	Usage

`class Satellite`
`{`
`  public double Mass`
`  {`
`  get { … }`
`  set { … }`
`  }`
`  ...`
`}`
	`...`
	`Satellite sat = new Satellite();`
	`...`
	`sat.Mass = 10.0;`
	`float mass = sat.Mass;`

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.

However, 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. It is envisaged, for example, that 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.

5.2.3 Association

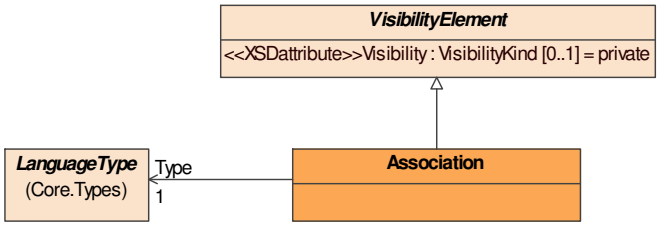

Figure 5-7: Association

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`).

5.3 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 be not equal, if they occupy different (memory) locations. Reference types include interfaces and models.

Figure 5-8 shows the metamodel elements for reference types. The two major Metaclasses are `Interface` and `Model`, which are described in detail in the following sections.

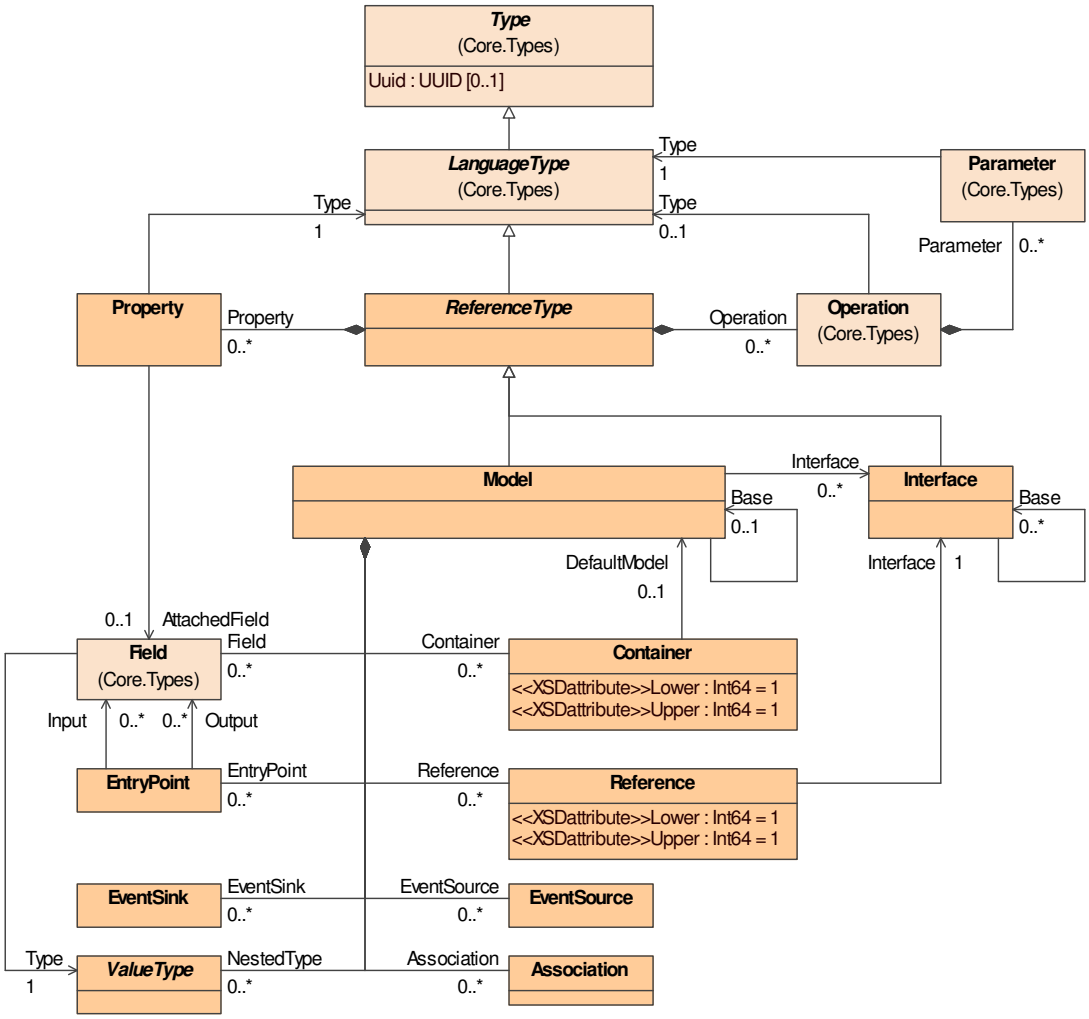

Figure 5-8: Reference Types (`Smdl.Catalogue` Schema)

Every reference type supports properties and operations. A model 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.

5.3.1 Reference Type

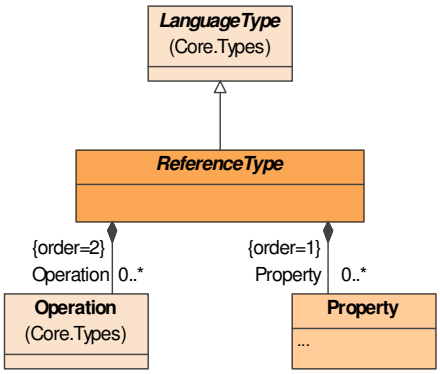

Figure 5-9: Reference Type

A `ReferenceType` may hold any number of properties (see 5.2.2) and operations (see 4.3.1). It serves as an abstract base class for `Interface` and `Model`.

5.3.2 Interface

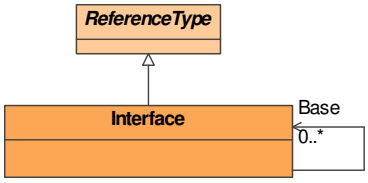

Figure 5-10: Interface

An `Interface` is a reference type that serves as a contract in a loosely coupled architecture. It has the ability to contain properties and operations (inherited from `ReferenceType`). An interface may inherit from other interfaces (interface inheritance), which is represented via the `Base` links; see the discussion in [AD-1, Section 2.3.7].

It is strongly recommended to only use value types, references and other interfaces in the properties and operations of an interface (i.e. do not 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.

5.3.3 Model

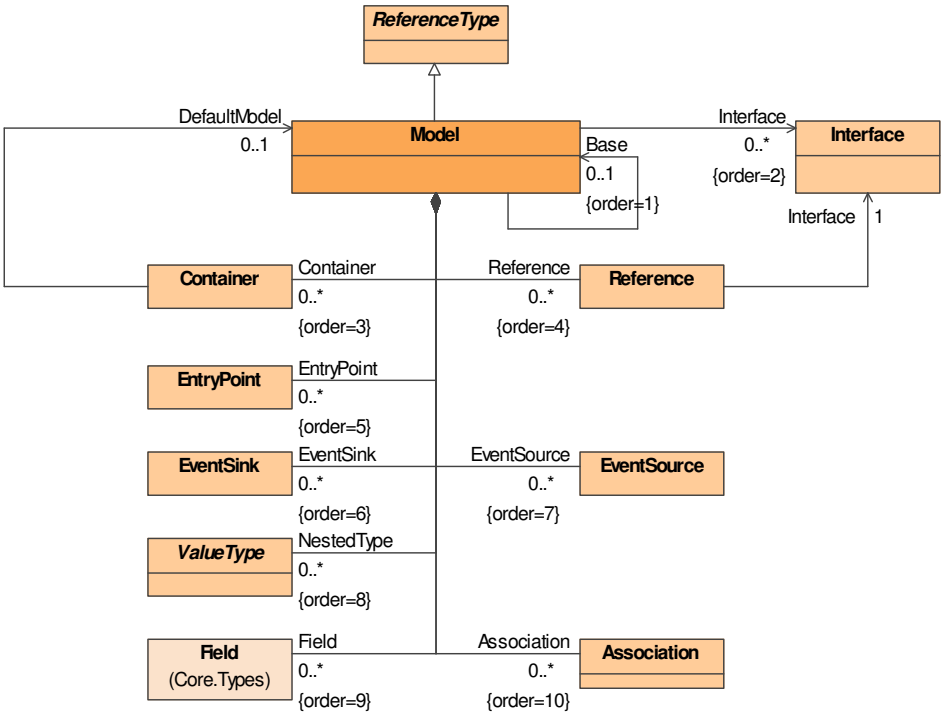

Figure 5-11: Model

The `Model` Metaclass is a reference type and hence inherits properties and operations from its base class. In addition, it provides various optional elements in order to allow various different modelling approaches. As a `Model` semantically forms a deployable unit (a component), it may use the available component mechanisms as specified in the SMP 2.0 Component Model [AD-2].

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` events 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 models (Aggregation).

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

For a Dataflow-based design, the fields of a `Model` can be tagged as `Input` or `Output` fields (see 4.3.1).

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.

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

5.3.3.1 Entry Point

Entry Points are needed in the models of a Catalogue to allow connecting Scheduling information later. See section 7 for details on scheduling.

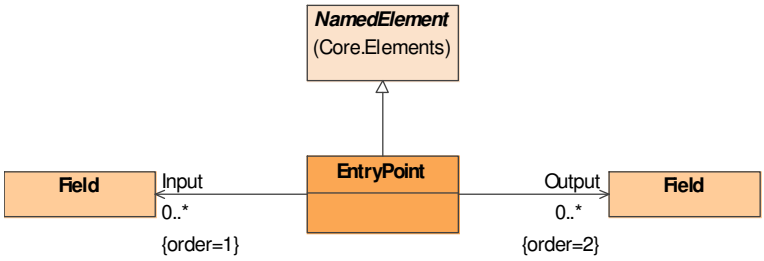

Figure 5-12: Entry Point

An `EntryPoint` is a named element of a `Model`. 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 should have their `Input` attribute set to `true`) and `Output` fields (which should 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.

5.3.3.2 Container

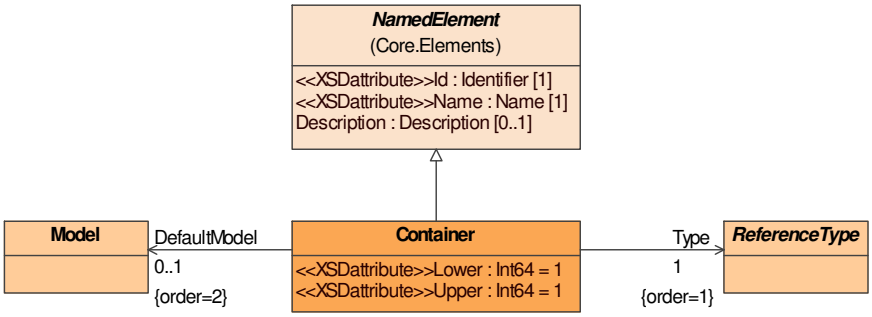

Figure 5-13: Container

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

The type of elements 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`[1].

Furthermore, a component collection allows specifying a default model (`DefaultModel`). If the Type link points to an Interface, then the default model needs to implement this interface. When the Type link

[1] *This is consistent to the XMI format used for UML, where an unlimited upper bound (typically represented as "*") is stored as -1. This allows using a number rather than a string for the upper bound.*

points to a Model, the default model either needs to be this model as well, or another model which inherits from this model. SMDL support tools may use this during instantiation (i.e. creation of an assembly) to select an initial implementation for newly created contained elements.

5.3.3.3 Reference

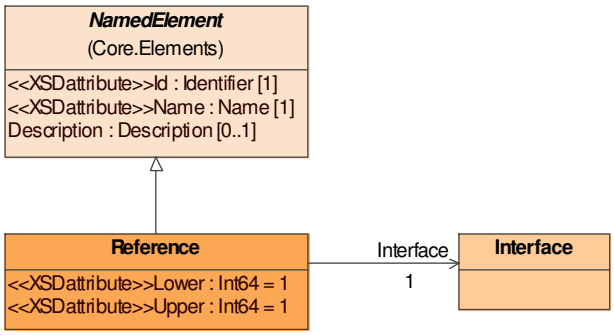

Figure 5-14: Reference

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

The type of models that can be referenced is specified via the `Interface` link.

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

5.4 Events

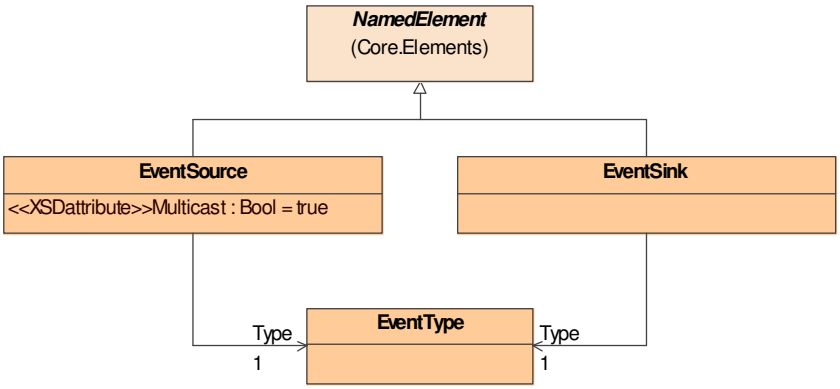

Figure 5-15: Events

Models developed in SMDL can use the concept of events to communicate with each other. This is supported by the `EventSource` and `EventSink` classes, which both are typed by an `EventType`.

Events are used to allow event-driven model interactions. They are realised using `EventType` (to classify an event), `EventSource` (to provide an event) and `EventSink` (to consume an event). Events are later connected (in SMDL Assemblies) using the `EventLink` Metaclass. Events are supported by `Model`.

5.4.1 Event Type

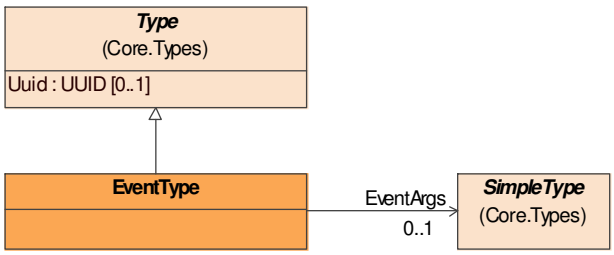

Figure 5-16: Event Type

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.

5.4.2 Event Source

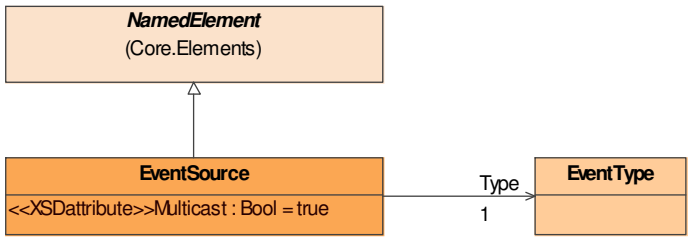

Figure 5-17: Event Source

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

5.4.3 Event Sink

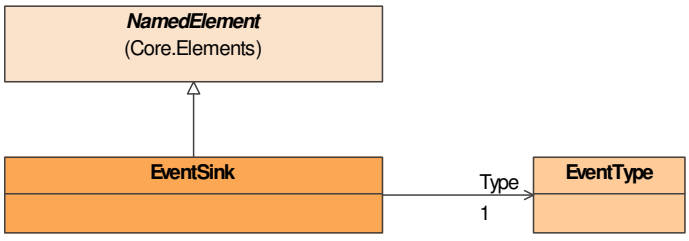

Figure 5-18: Event Sink

An `EventSink` is used to specify that a `Model` can consume 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).

5.5 Catalogue Attributes

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

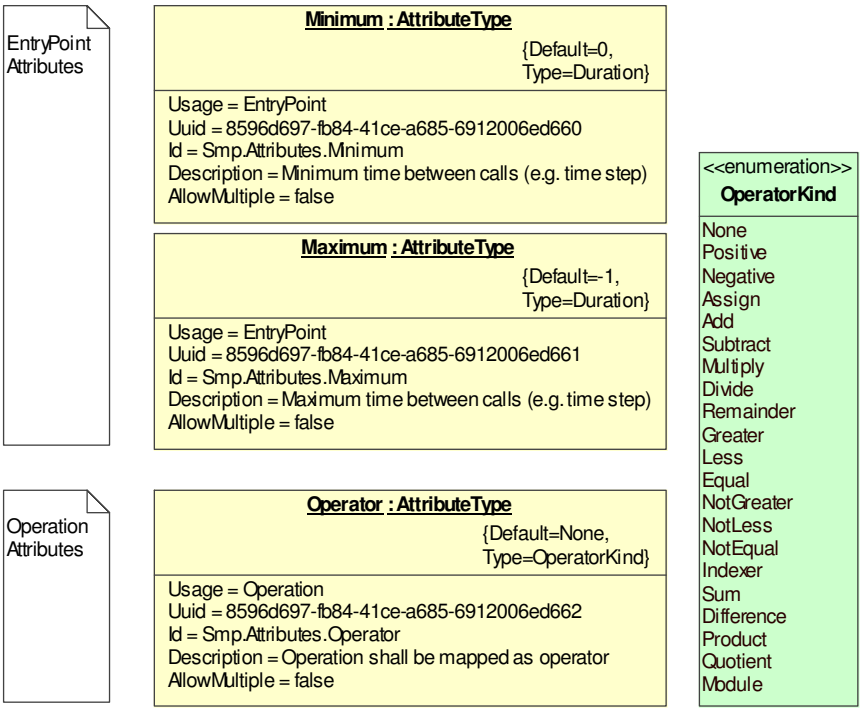

Figure 5-19: Predefined Catalogue Attributes

The attributes are summarised in the table below. .

Table 5.3: Predefined Catalogue Attributes

Name	Description	Uuid	Usage	Type	Default
Minimum	Defines minimum duration between two calls.	`8596d697-fb84-41ce-a685-6912006ed660`	`EntryPoint`	`Duration`	`0`
Maximum	Defines maximum duration between two calls.	`8596d697-fb84-41ce-a685-6912006ed661`	`EntryPoint`	`Duration`	`-1`
Operator	Defines operator that shall be used for operation.	`8596d697-fb84-41ce-a685-6912006ed662`	`Operator`	`OperatorKind`	`None`

5.5.1 Minimum Attribute

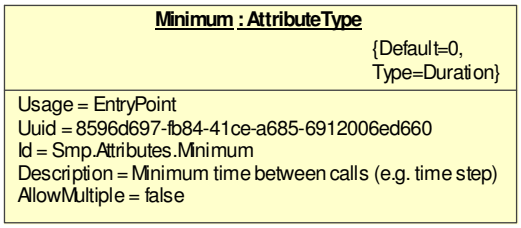

Figure 5-20: Minimum Attribute Type

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.

5.5.2 Maximum Attribute

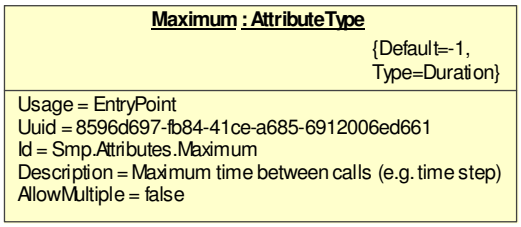

Figure 5-21: Maximum Attribute Type

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

5.5.3 Operator Attribute

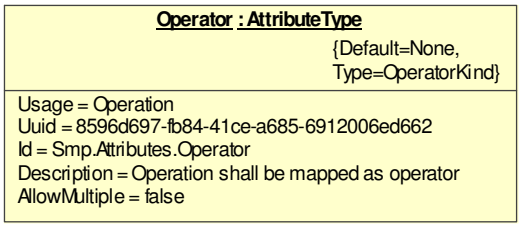

Figure 5-22: Operator Attribute Type

The `Operator` attribute type defines an operator kind for an operation. When supported by the target platform, it can be evaluated to map the operation to an operator. The default value for this attribute is `None`, which corresponds to not mapping to an operator.

Typically, the mapping of an `Operator` to a target language (e.g. to C++) differs from the mapping of an `Operation` (operator overloading), although it is possible to map operators to operations as well (e.g. if operator overloading is not supported by the target platform). Note that a mapping to an operator will not make use of the operation name, but of the operator. Nevertheless, operations with an `Operator` attribute

have to have unique names within their context, to ensure that a mapping to a platform without operator overloading is possible.

5.5.3.1 Operator Kind

The different operator kinds supported by SMDL are defined via the `OperatorKind` enumeration type (Uuid `8596d697-fb84-41ce-a685-6912006ed663`, Namespace `Smp::Attributes`; compare Table 5.3 and Figure 5-19 above). Table 5.4 below shows all operator kinds together with the mapping of each operator to the C++ Programming Language.

Table 5.4: Operator Kinds

Kind	Enum. Value	Description	Ordinality	C++ Example
`None`	0	Undefined.	`-`	`-`
`Positive`	1	Positive value of instance.	`0`	`+x`
`Negative`	2	Negative value of instance.	`0`	`-x`
`Assign`	3	Assigns new value to instance.	`1`	`x = a`
`Add`	4	Adds value to instance.	`1`	`x += a`
`Subtract`	5	Subtracts value from instance.	`1`	`x -= a`
`Multiply`	6	Multiplies instance with value.	`1`	`x *= a`
`Divide`	7	Divides instance by value.	`1`	`x /= a`
`Remainder`	8	Remainder of instance for value.	`1`	`x %= a`
`Greater`	9	Compares whether instance is greater than value.	`1`	`x > a`
`Less`	10	Compares whether instance is less than value.	`1`	`x < a`
`Equal`	11	Compares whether instance is equal to value.	`1`	`x == a`
`NotGreater`	12	Compares whether instance is not gerate than value.	`1`	`x <= a`
`NotLess`	13	Compares whether instance is notless than value.	`1`	`x >= a`
`NotEqual`	14	Compares whether instance is not equal to value.	`1`	`x != a`
`Indexer`	15	Returns indexed value of instance.	`1`	`x[a]`
`Sum`	16	Returns sum of two values.	`2`	`a + b`
`Difference`	17	Returns difference between two values.	`2`	`a - b`
`Product`	18	Returns product of two values.	`2`	`a * b`
`Quotient`	19	Returns quotient of two values.	`2`	`a / b`
`Module`	20	Returns remainder of two values.	`2`	`a % b`

6. SMDL ASSEMBLIES

This section describes all metamodel elements that are needed in order to define an assembly of model instances. This does include 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 An Assembly Document

Figure 6-1 shows the structure of an SMDL assembly document.

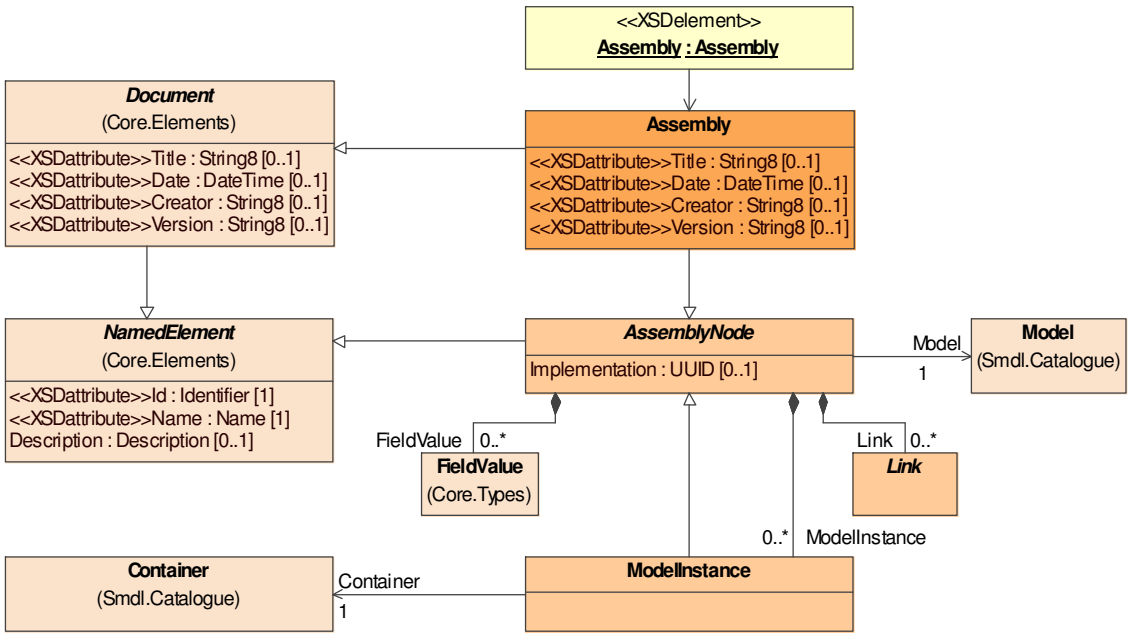

Figure 6-1: Assembly Document (`Smdl.Assembly` Schema)

An assembly contains model instances, where each instance references a `Model` within a catalogue. 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 source and event sink (`EventLink`), or output and input fields (`FieldLink`).

The classes introduced in Figure 6-1 are detailed below.

6.1.1 Assembly Node

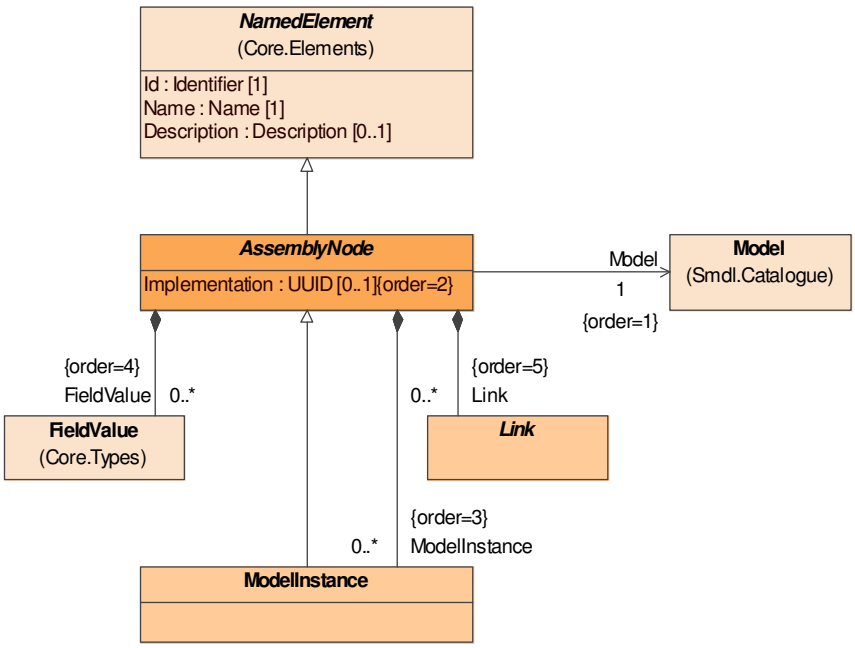

Figure 6-2: Assembly Node

The `AssemblyNode` Metaclass is an abstract base class for both `Assembly` and `ModelInstance` (see below) that provides the common features of both Metaclasses.

An `AssemblyNode` represents an instance of a model. Therefore, it has to link to a `Model`. To allow creating a run-time model instance, the assembly node needs to specify as well which `Implementation` shall be used when loading the assembly into a simulation environment. This is done by specifying a Universally Unique Identifier (**UUID**).

Depending on the containers of the referenced model, the `AssemblyNode` can hold a number of child model instances.

For each field of the referenced model, the `AssemblyNode` can specify a `FieldValue` (see 4.4.5.1 above).

An `AssemblyNode` can link its references, event sinks and input fields, which are specified by the referenced model, to other model instances via the `Link` elements (see 6.2.1 below).

6.1.2 Assembly

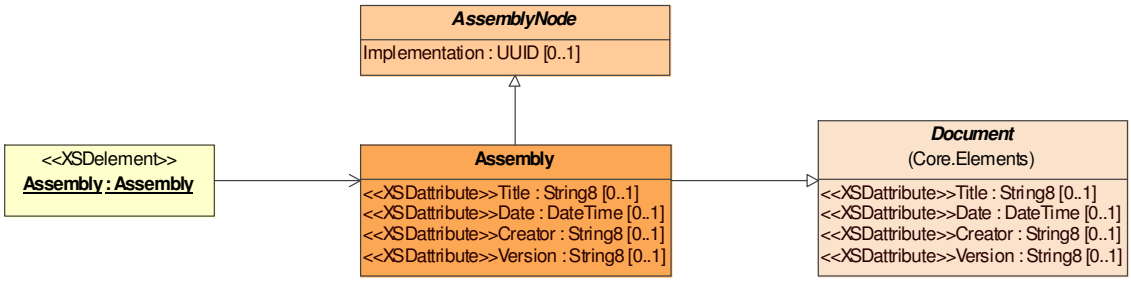

Figure 6-3: Assembly

An `Assembly` is an assembly node that acts as the root element (top-level node) in an SMDL Assembly document. As we cannot use multiple inheritance in the metamodel, the `Document` features are replicated in the `Assembly` Metaclass. Apart from that, `Assembly` inherits from `AssemblyNode`, such that it is also typed by a `Model` of a catalogue. It is nearly identical to a model instance (see below), except that it does not point to its container, being a top-level node. This similarity, which manifests in the `AssemblyNode` metaclass, will allow replacing child model instances by sub-assemblies later if needed.

6.1.3 Model Instance

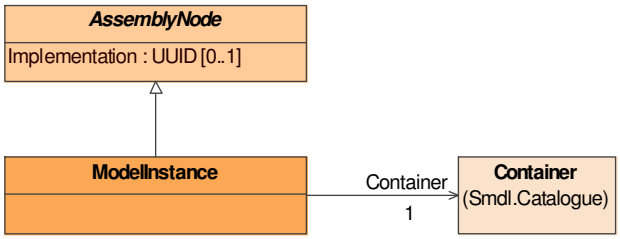

Figure 6-4: Model Instance

A `ModelInstance` represents an instance of a model. It is derived from the abstract base Metaclass `AssemblyNode` (see section 6.1.1), 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.

6.2 Links

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

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

All link Metaclasses are derived from a common `Link` base class.

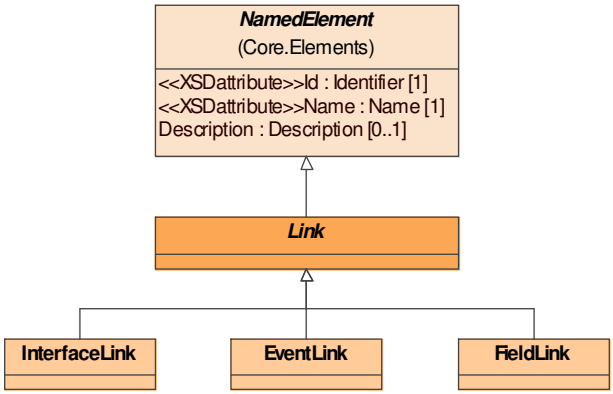

Figure 6-5: Links

6.2.1 Link

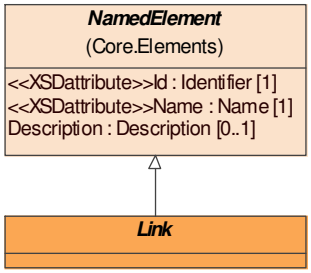

Figure 6-6: Link

The `Link` Metaclass does not introduce new attributes, but serves as a base class for derived Metaclasses.

6.2.2 Interface Link

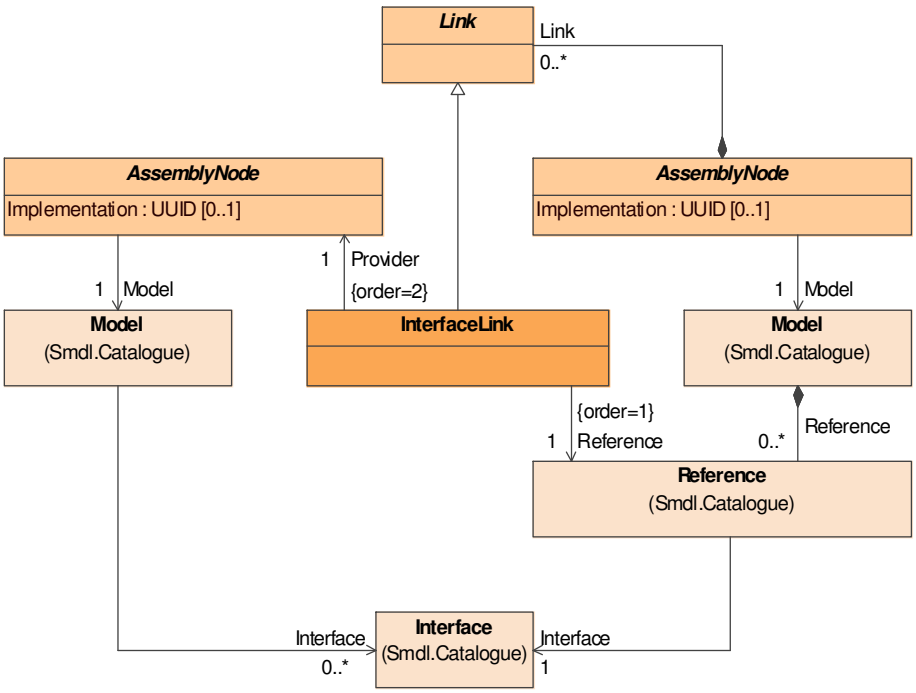

Figure 6-7: Interface Link

An `InterfaceLink` resolves a reference of an assembly node (assembly or model instance) 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 allow an assembly to be interpreted without the associated catalogue, the `xlink:title` attribute of the `Reference` link *shall* contain the reference name (i.e. the name of the `Reference` element in the catalogue).

In order to uniquely identify the link target, the `InterfaceLink` links to the `Provider` assembly node (model instance or assembly), which provides (via its `Model`) a matching `Interface`.

To be semantically correct, the `Model` of the `Provider` needs to provide an `Interface` that coincides with the `Interface` the `Reference` points to, or needs to be derived from it via interface inheritance.

6.2.3 Event Link

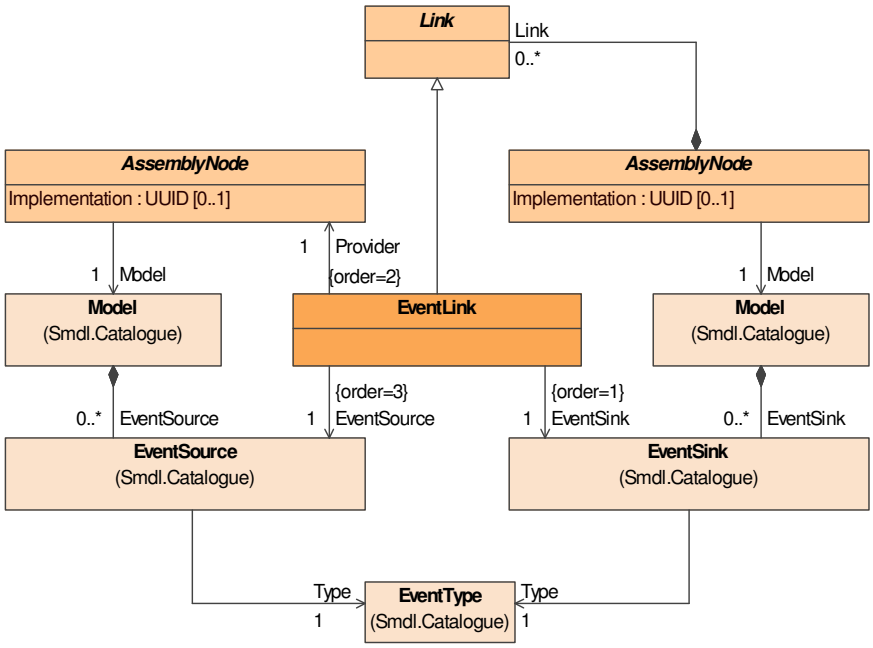

Figure 6-8: Event Link

An `EventLink` resolves an event sink of an assembly node (assembly or model instance) 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 event sink and event source, respectively.

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

6.2.4 Field Link

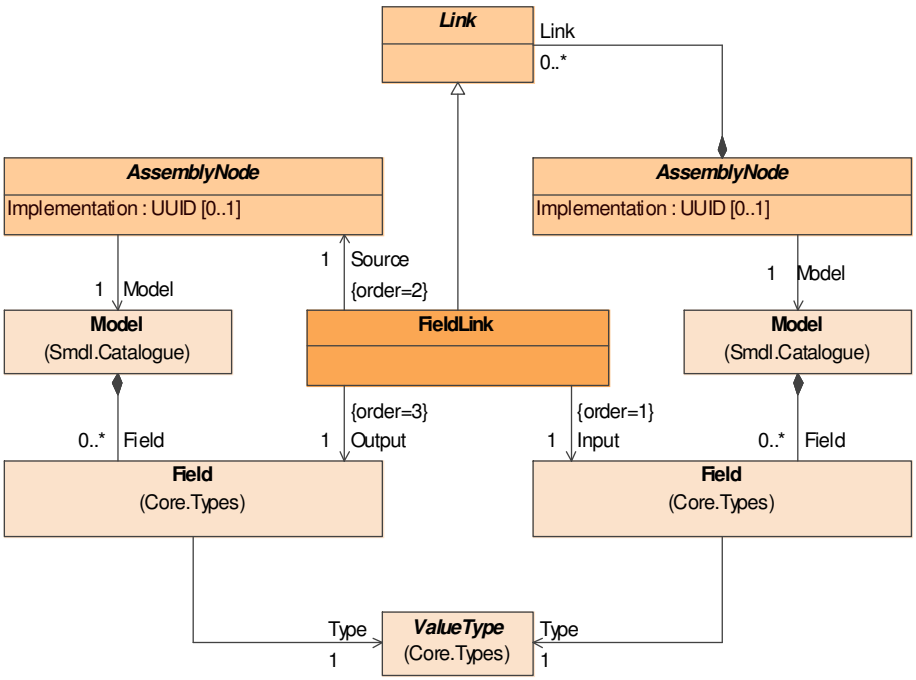

Figure 6-9: Field Link

A `FieldLink` resolves an input field of an assembly node (assembly or model instance) 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.

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 fields, 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.

This Page is Intentionally left Blank

7. SMDL SCHEDULES

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

7.1 A Schedule Document

Figure 7-1 shows the top-level structure of an SMDL schedule document.

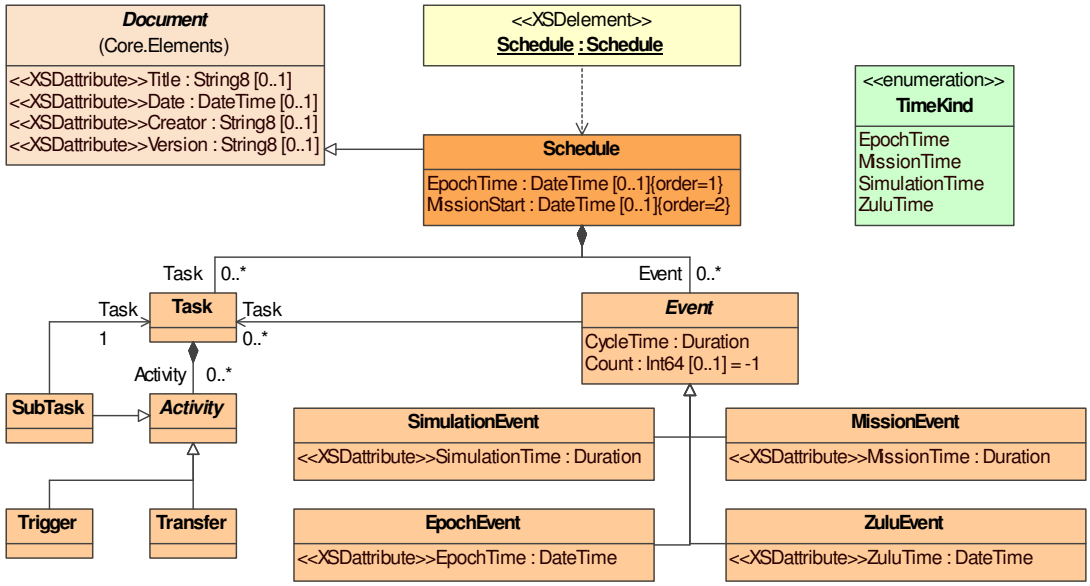

Figure 7-1: Schedule Document (`Smdl.Schedule` Schema)

The classes introduced in Figure 7-1 are detailed below.

7.1.1 Schedule

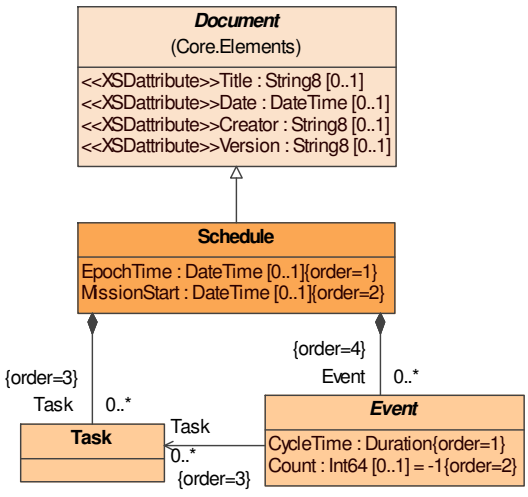

Figure 7-2: Schedule

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 typically used to initialise epoch time and mission time via the `SetEpochTime()` and `SetMissionStart()` methods of the *TimeKeeper* service [AD-2].

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

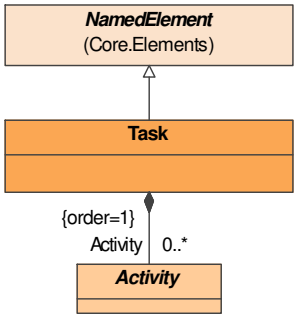

Figure 7-3: Task

A `Task` is a container of activities. The order of activities defines in which order the `Activity` elements are called.

7.2.1.1 Activity

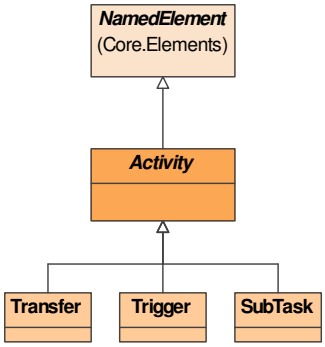

Figure 7-4: Activity

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

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

7.2.1.2 Trigger

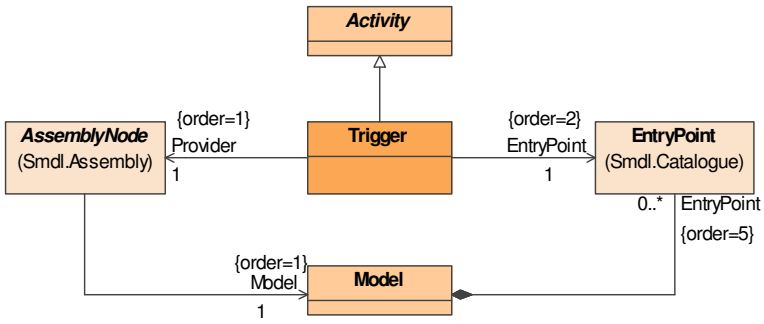

Figure 7-5: Trigger

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

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.

7.2.1.3 Transfer

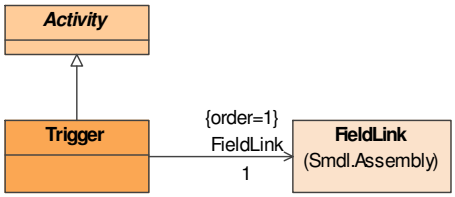

Figure 7-6: Transfer

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

7.2.1.4 Sub-Task

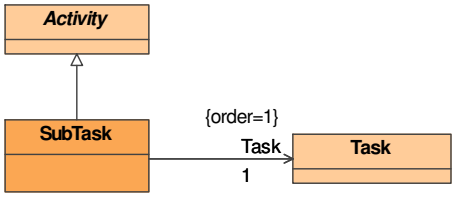

Figure 7-7: Sub-Task

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

Circular references between tasks are not allowed.

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 SMP2. Therefore, four different types of events exist which all derive from a common base class.

7.3.1 Event

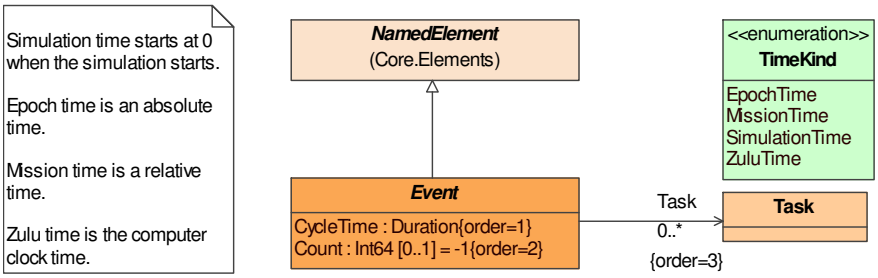

Figure 7-8: Timed Event

An `Event` defines a point in time when to execute a collection of tasks. Such a scheduler 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 `Count` element defines how often an event shall be repeated. An event with `Count=0` will not be repeated, so it will be only called once. This is a single event. An event with `Count>0` will be called `Count+1` times, so it is cyclic. Finally, a value of `Count=-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:

- **Simulation Time** starts at 0 when the simulation is kicked off. It progresses while the simulation is in execution mode.
- **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.
- **Mission Time** is a relative time (duration from some start time) and typically progresses with epoch time.
- **Zulu Time** is the computer clock time, also called wall clock time. It progresses whether the simulation is in execution or standby mode, and is not necessarily related to simulation, epoch or mission time.

For each of these four time kinds, a dedicated metaclass derived from `Event` exists. 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`.

7.3.2 Simulation Event

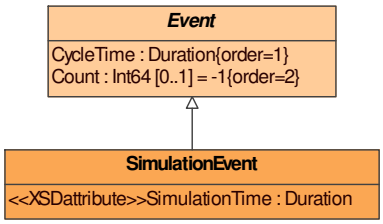

Figure 7-9: Simulation Event

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

7.3.3 Epoch Event

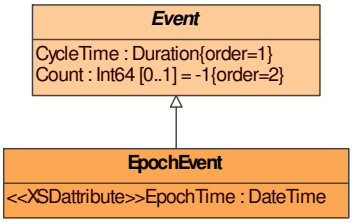

Figure 7-10: Epoch Event

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

7.3.4 Mission Event

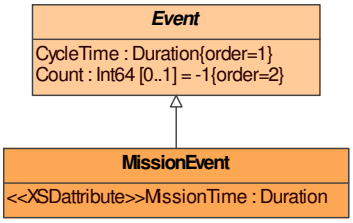

Figure 7-11: Mission Event

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

7.3.5 Zulu Event

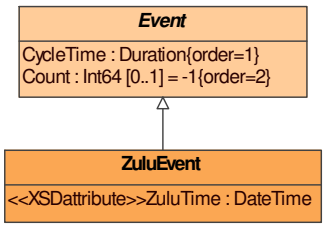

Figure 7-12: Zulu Event

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

7.4 Schedule Attributes

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

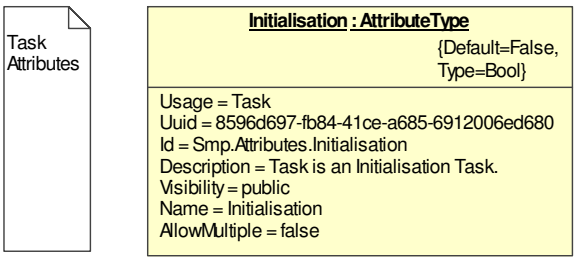

Figure 7-13: Predefined Schedule Attributes

The attributes are summarised in the table below. .

Table 7.1: Predefined Schedule Attributes

Name	Description	Uuid	Usage	Type	Default
Initialisation	Task is an Initialisation Task,	`8596d697-fb84-41ce-a685-6912006ed680`	`Task`	`Bool`	`False`

8. SMDL PACKAGES

This section 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 A Package Document

Figure 8-1 shows the top-level structure of an SMDL package document.

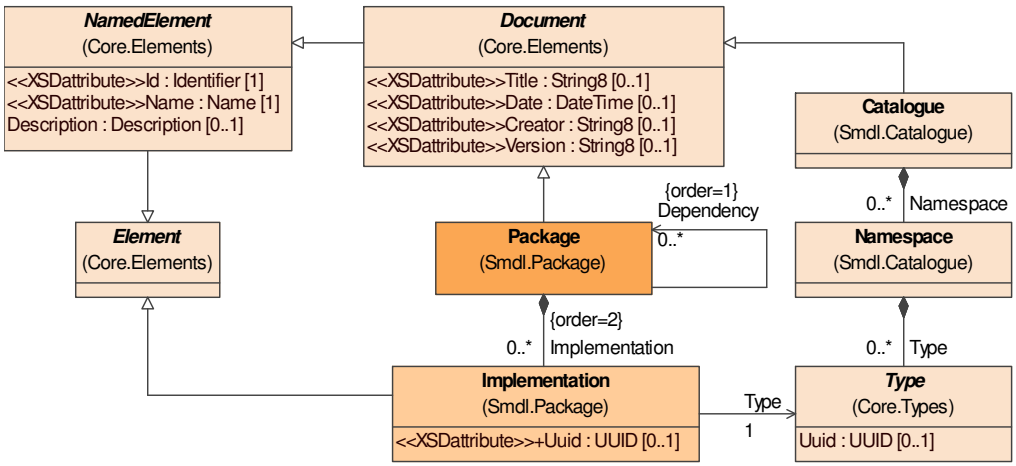

Figure 8-1: Package Document (`Smdl.Package` Schema)

The classes introduced in Figure 8-1 are detailed below.

8.1.1 Package

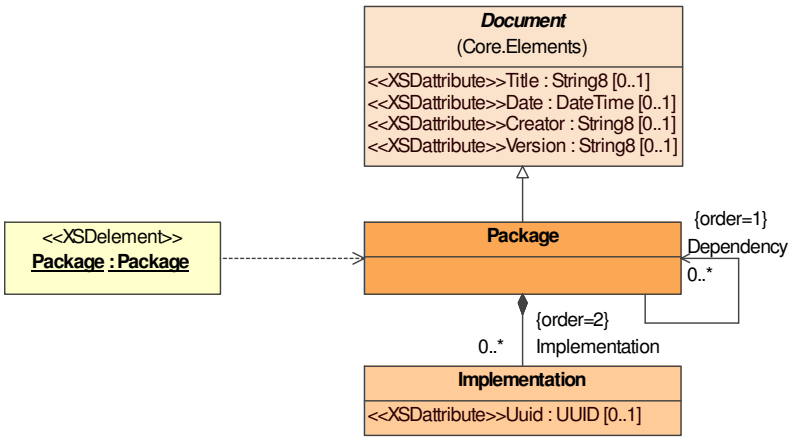

Figure 8-2: Package

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.

8.1.2 Implementation

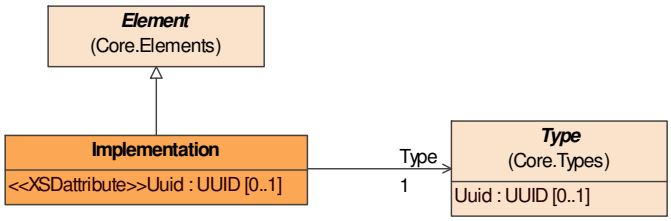

Figure 8-3: Implementation

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.

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.

9. SMDL WORKSPACES

This section describes all Metamodel elements that are needed in order to define a workspace of other documents that belong together. Such a workspace has no dedicated meaning for model design, integration or execution, but is an optional convenience mechanism that may be used by SMDL tools to group related SMDL documents together.

9.1 A Workspace Document

Figure 9-1 shows the top-level structure of an SMDL workspace document.

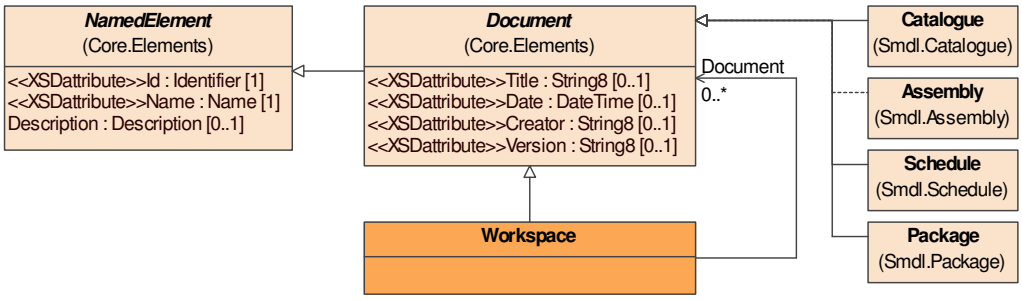

Figure 9-1: Workspace Document (`Smdl.Workspace` Schema)

The classes introduced in Figure 9-1 are detailed below.

9.1.1 Workspace

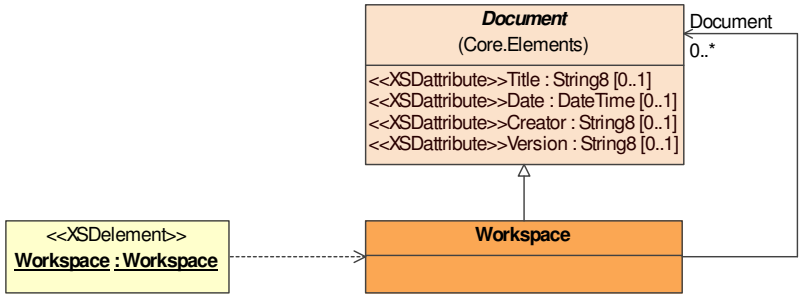

Figure 9-2: Workspace

A `Workspace` is a `Document` that references an arbitrary number of other SMDL documents via its `Document` links.

This Page is Intentionally left Blank

10. APPENDIX A: XML LINKS AND PRIMITIVE TYPES

10.1 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.

10.1.1 Simple Link

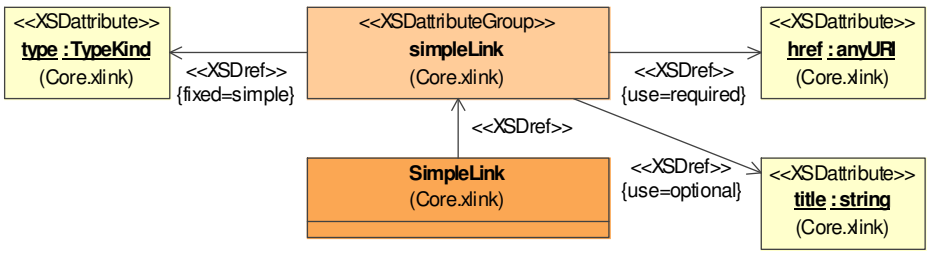

Figure 10-1: Simple Link (`xlink` Schema)

A `SimpleLink` is an XML Link of a fixed `type` ("`simple`") which references any Uniform Resource Identifier (**URI**) using its `href` attribute. Further, it may additionally identify the link target by name using its `title` attribute.

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

10.2 Primitive Types

All SMDL schemas are 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++.

The XML Schema representations of the primitive types are held in the `Core.PrimitiveTypes` schema. Note that some of the types are only used for the definition of the Metamodel and the Component Model (`String8`, `AnySimple`), while the other types may also be used by SMP2 models.

10.2.1 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.

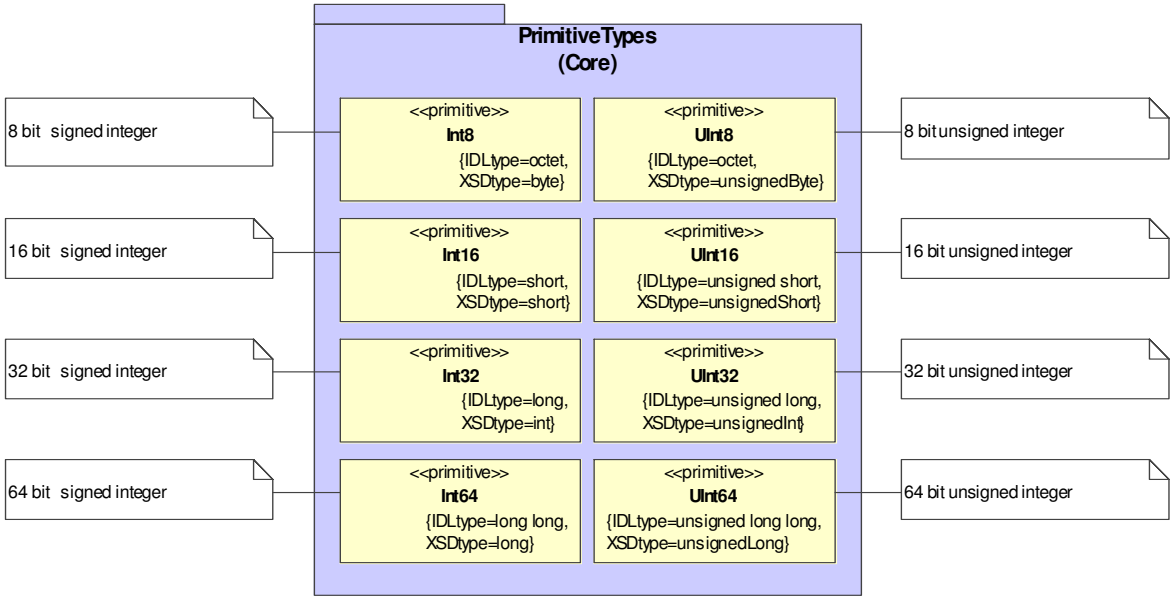

Figure 10-2: Integer types

10.2.2 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.

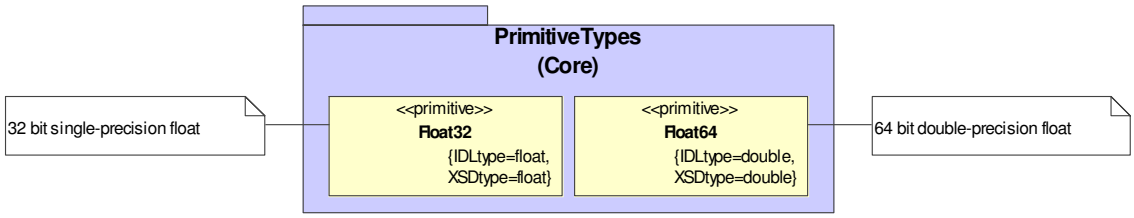

Figure 10-3: Floating-point types

10.2.3 Character and String Types

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

Furthermore, for use in the Metamodel and in the Component Model, SMP2 also supports 8-bit character strings based on UTF-8 encoding, which is commonly used in XML.

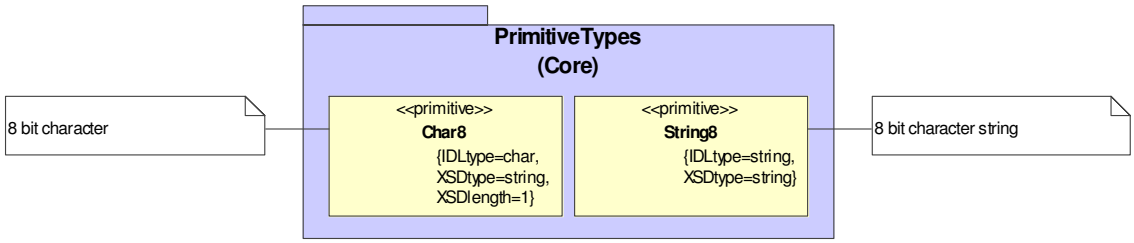

Figure 10-4: Character and String types

10.2.4 Other Primitive Types

Apart from the above integer, floating-point, and character and string types, SMDL additionally supports other relevant 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`.

Furthermore, a container type is introduced for use in the Metamodel and Component Model.

- `AnySimple` is a container for any simple type.

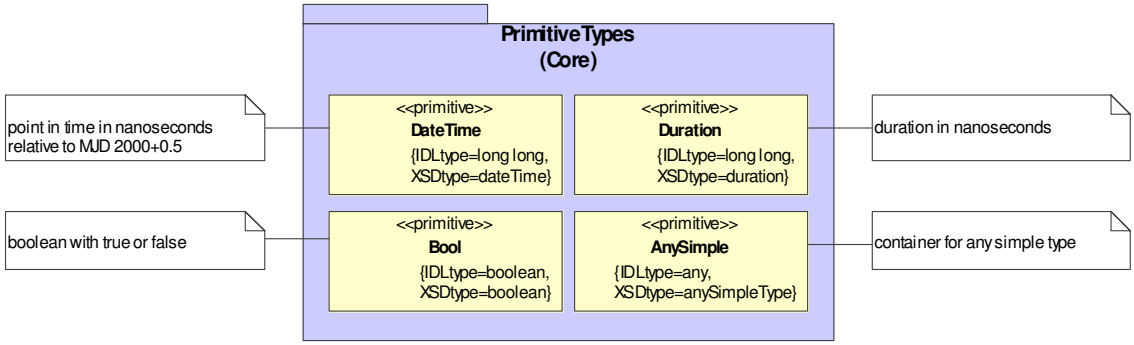

Figure 10-5: Other primitive types

This Page is Intentionally left Blank

11. APPENDIX B: XML SCHEMAS

This section lists the XML schemas that have been generated from the UML representation of the Metamodel presented above. The schemas are *normative* in that they define the exact syntax of SMDL Catalogue, Assembly and Schedule files.

11.1 The XML Linking (xlink) Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xlink="http://www.w3.org/1999/xlink" targetNamespace="http://www.w3.org/1999/xlink"
elementFormDefault="unqualified" attributeFormDefault="unqualified">
   <xsd:annotation>
      <xsd:documentation>
UML Model Information:
    Filename:        file:/c:/PEllsiepen/cvswork/Metamodel/uml/gen/flat-Smdl.xml
    Last modified:   Mon Feb 28 12:04:14 CET 2005
    Model name:      Data
    Model comments:
Author:Peter Ellsiepen, Peter Fritzen.
Created:.
Title:SMDL Specification (SMP 2.0, Issue 1.1).
Comment:This is the specification of the Simulation Model Definition Language (SMDL).

    XMI version:     1.2
    XMI exporter:    MagicDraw UML 8.0
    XMI metamodel:   UML 1.4

XSLT Processing Information:
    Processing date: 2005-02-28+01:00 12:07:05.292+01:00
    XSLT Processor:  SAXON 8.1.1 from Saxonica
    XSLT Version:    2.0
    XSLT Stylesheet: xmi-to-xsd.xslt
      </xsd:documentation>
   </xsd:annotation>
   <!-- ======================================== -->
   <!-- PACKAGE: xlink -->
   <!-- ======================================== -->
   <xsd:annotation>
      <xsd:documentation>
This package is specific to the XML Schema platform and provides XLink attributes and
attribute groups to allow properly validating other schemas holding XML Links in the
standard xlink namespace prefix.

Note that there is no normative schema for the XLink specification.
      </xsd:documentation>
   </xsd:annotation>
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- <<XSDattribute>> Objects                  -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- OBJECT: type -->
   <xsd:attribute name="type" type="xlink:TypeKind">
      <xsd:annotation>
         <xsd:documentation>
ATTRIBUTE: type

The type attribute identifies XLink element types.

The value of the type attribute must be supplied. The value must be one of "simple",
"extended", "locator", "arc", "resource", "title", or "none".

When the value of the type attribute is "none", the element has no XLink-specified
meaning, and any XLink-related content or attributes have no XLink-specified relationship
to the element.      </xsd:documentation>
      </xsd:annotation>
   </xsd:attribute>
   <!-- OBJECT: arcrole -->
   <xsd:attribute name="arcrole" type="xlink:anyURI">
      <xsd:annotation>
         <xsd:documentation>
ATTRIBUTE: arcrole
```

```
The attributes that describe the meaning of resources within the context of a link are
role, arcrole, and title.

The arcrole attribute may be used on arc- and simple-type elements.

The URI reference identifies some resource that describes the intended property. When no
value is supplied, no particular role value is to be inferred.      </xsd:documentation>
      </xsd:annotation>
   </xsd:attribute>
   <!-- OBJECT: href -->
   <xsd:attribute name="href" type="xlink:anyURI">
      <xsd:annotation>
         <xsd:documentation>
ATTRIBUTE: href

The href attribute supplies the data that allows an XLink application to find a remote
resource (or resource fragment). It may be used on simple-type elements, and must be used
on locator-type elements.      </xsd:documentation>
      </xsd:annotation>
   </xsd:attribute>
   <!-- OBJECT: label -->
   <xsd:attribute name="label" type="xlink:NMTOKEN">
      <xsd:annotation>
         <xsd:documentation>
ATTRIBUTE: label

The traversal attributes are label, from, and to.

The label attribute may be used on the resource- and locator-type elements.
</xsd:documentation>
      </xsd:annotation>
   </xsd:attribute>
   <!-- OBJECT: title -->
   <xsd:attribute name="title" type="xlink:string">
      <xsd:annotation>
         <xsd:documentation>
ATTRIBUTE: title

The attributes that describe the meaning of resources within the context of a link are
role, arcrole, and title.

The title attribute may be used on extended-, simple-, locator-, resource-, and arc-type
elements.

The title attribute is used to describe the meaning of a link or resource in a human-
readable fashion, along the same lines as the role or arcrole attribute. A value is
optional; if a value is supplied, it should contain a string that describes the resource.
The use of this information is highly dependent on the type of processing being done. It
may be used, for example, to make titles available to applications used by visually
impaired users, or to create a table of links, or to present help text that appears when
a user lets a mouse pointer hover over a starting resource.
      </xsd:documentation>
      </xsd:annotation>
   </xsd:attribute>
   <!-- OBJECT: show -->
   <xsd:attribute name="show" type="xlink:ShowKind">
      <xsd:annotation>
         <xsd:documentation>
ATTRIBUTE: show

The show attribute is used to communicate the desired presentation of the ending resource
on traversal from the starting resource.

Conforming XLink applications should apply the specified treatment for the show values
(see ShowKind).

Background:
The behavior attributes are show and actuate. They may be used on the simple- and arc-
type elements. When used on a simple-type element, they signal behavior intentions for
traversal to that link's single remote ending resource. When they are used on an arc-type
element, they signal behavior intentions for traversal to whatever ending resources
(local or remote) are specified by that arc.
```

```
The show and actuate attributes are not required. When they are used, conforming XLink
applications should give them the treatment specified in this section. There is no hard
requirement ("must") for this treatment because what makes sense for an interactive
application, such as a browser, is unlikely to make sense for a noninteractive
application, such as a robot. However, all applications should take into account the full
implications of ignoring the specified behavior before choosing a different course.
</xsd:documentation>
      </xsd:annotation>
   </xsd:attribute>
   <!-- OBJECT: to -->
   <xsd:attribute name="to" type="xlink:NMTOKEN">
      <xsd:annotation>
         <xsd:documentation>
ATTRIBUTE: from

The traversal attributes are label, from, and to.

The from and to attributes may be used on the arc-type element.

Constraint: If a value is supplied for a from or to attribute, it must correspond to the
same value for some label attribute on a locator- or resource-type element that appears
as a direct child inside the same extended-type element as does the arc-type element.
      </xsd:documentation>
      </xsd:annotation>
   </xsd:attribute>
   <!-- OBJECT: actuate -->
   <xsd:attribute name="actuate" type="xlink:ActuateKind">
      <xsd:annotation>
         <xsd:documentation>
ATTRIBUTE: actuate

The actuate attribute is used to communicate the desired timing of traversal from the
starting resource to the ending resource.

Conforming XLink applications should apply the specified treatment for the actuate values
(see ActuateKind).

Background:
The behavior attributes are show and actuate. They may be used on the simple- and arc-
type elements. When used on a simple-type element, they signal behavior intentions for
traversal to that link's single remote ending resource. When they are used on an arc-type
element, they signal behavior intentions for traversal to whatever ending resources
(local or remote) are specified by that arc.

The show and actuate attributes are not required. When they are used, conforming XLink
applications should give them the treatment specified in this section. There is no hard
requirement ("must") for this treatment because what makes sense for an interactive
application, such as a browser, is unlikely to make sense for a noninteractive
application, such as a robot. However, all applications should take into account the full
implications of ignoring the specified behavior before choosing a different course.
</xsd:documentation>
      </xsd:annotation>
   </xsd:attribute>
   <!-- OBJECT: role -->
   <xsd:attribute name="role" type="xlink:anyURI">
      <xsd:annotation>
         <xsd:documentation>
ATTRIBUTE: role

The attributes that describe the meaning of resources within the context of a link are
role, arcrole, and title.

The role attribute may be used on extended-, simple-, locator-, and resource-type
elements.

The URI reference identifies some resource that describes the intended property. When no
value is supplied, no particular role value is to be inferred.      </xsd:documentation>
      </xsd:annotation>
   </xsd:attribute>
   <!-- OBJECT: from -->
   <xsd:attribute name="from" type="xlink:NMTOKEN">
      <xsd:annotation>
         <xsd:documentation>
ATTRIBUTE: from
```

```
The traversal attributes are label, from, and to.

The from and to attributes may be used on the arc-type element.

Constraint: If a value is supplied for a from or to attribute, it must correspond to the
same value for some label attribute on a locator- or resource-type element that appears
as a direct child inside the same extended-type element as does the arc-type element.
      </xsd:documentation>
      </xsd:annotation>
   </xsd:attribute>
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- <<XSDattributeGroup>> Classes                 -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- CLASS: arcLink -->
   <xsd:attributeGroup name="arcLink">
      <xsd:attribute ref="xlink:type" fixed="arc"/>
      <xsd:attribute ref="xlink:title"/>
      <xsd:attribute ref="xlink:arcrole"/>
      <xsd:attribute ref="xlink:from"/>
      <xsd:attribute ref="xlink:to"/>
      <xsd:attribute ref="xlink:actuate"/>
      <xsd:attribute ref="xlink:show"/>
   </xsd:attributeGroup>
   <!-- CLASS: extendedLink -->
   <xsd:attributeGroup name="extendedLink">
      <xsd:attribute ref="xlink:type" fixed="extended"/>
      <xsd:attribute ref="xlink:title"/>
      <xsd:attribute ref="xlink:role"/>
   </xsd:attributeGroup>
   <!-- CLASS: locatorLink -->
   <xsd:attributeGroup name="locatorLink">
      <xsd:attribute ref="xlink:type" fixed="locator"/>
      <xsd:attribute ref="xlink:href" use="required"/>
      <xsd:attribute ref="xlink:label"/>
      <xsd:attribute ref="xlink:title"/>
      <xsd:attribute ref="xlink:role"/>
   </xsd:attributeGroup>
   <!-- CLASS: resourceLink -->
   <xsd:attributeGroup name="resourceLink">
      <xsd:attribute ref="xlink:type" fixed="resource"/>
      <xsd:attribute ref="xlink:label"/>
      <xsd:attribute ref="xlink:title"/>
      <xsd:attribute ref="xlink:role"/>
   </xsd:attributeGroup>
   <!-- CLASS: simpleLink -->
   <xsd:attributeGroup name="simpleLink">
      <xsd:attribute ref="xlink:type" fixed="simple"/>
      <xsd:attribute ref="xlink:href" use="required"/>
      <xsd:attribute ref="xlink:title" use="optional"/>
      <xsd:attribute ref="xlink:actuate"/>
      <xsd:attribute ref="xlink:show"/>
      <xsd:attribute ref="xlink:role"/>
      <xsd:attribute ref="xlink:arcrole"/>
   </xsd:attributeGroup>
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- <<primitive>> DataTypes                       -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- DATATYPE: anyURI -->
   <xsd:simpleType name="anyURI">
      <xsd:annotation>
         <xsd:documentation>
Universal Resource Identifier      </xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:anyURI"/>
   </xsd:simpleType>
   <!-- DATATYPE: ID -->
   <xsd:simpleType name="ID">
      <xsd:restriction base="xsd:ID"/>
   </xsd:simpleType>
   <!-- DATATYPE: NMTOKEN -->
   <xsd:simpleType name="NMTOKEN">
      <xsd:restriction base="xsd:NMTOKEN"/>
   </xsd:simpleType>
```

```
   <!-- DATATYPE: string -->
   <xsd:simpleType name="string">
      <xsd:annotation>
         <xsd:documentation>
8 bit character string      </xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:string"/>
   </xsd:simpleType>
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- <<enumeration>> Classes                -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- CLASS: ActuateKind -->
   <xsd:simpleType name="ActuateKind">
      <xsd:restriction base="xsd:string">
         <xsd:enumeration value="onLoad">
            <xsd:annotation>
               <xsd:documentation>
An application should traverse to the ending resource immediately on loading the starting
resource.      </xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
         <xsd:enumeration value="onRequest">
            <xsd:annotation>
               <xsd:documentation>
An application should traverse from the starting resource to the ending resource only on
a post-loading event triggered for the purpose of traversal.      </xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
         <xsd:enumeration value="other">
            <xsd:annotation>
               <xsd:documentation>
The behavior of an application traversing to the ending resource is unconstrained by this
specification. The application should look for other markup present in the link to
determine the appropriate behavior.
      </xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
         <xsd:enumeration value="none">
            <xsd:annotation>
               <xsd:documentation>
The behavior of an application traversing to the ending resource is unconstrained by this
specification. No other markup is present to help the application determine the
appropriate behavior.
      </xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
      </xsd:restriction>
   </xsd:simpleType>
   <!-- CLASS: ShowKind -->
   <xsd:simpleType name="ShowKind">
      <xsd:restriction base="xsd:string">
         <xsd:enumeration value="new">
            <xsd:annotation>
               <xsd:documentation>
An application traversing to the ending resource should load it in a new window, frame,
pane, or other relevant presentation context.      </xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
         <xsd:enumeration value="replace">
            <xsd:annotation>
               <xsd:documentation>
An application traversing to the ending resource should load the resource in the same
window, frame, pane, or other relevant presentation context in which the starting
resource was loaded.      </xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
         <xsd:enumeration value="embed">
            <xsd:annotation>
               <xsd:documentation>
An application traversing to the ending resource should load its presentation in place of
the presentation of the starting resource. The presentation of embedded resources is
application dependent.      </xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
```

```
         <xsd:enumeration value="other">
            <xsd:annotation>
               <xsd:documentation>
The behavior of an application traversing to the ending resource is unconstrained by this
specification. The application should look for other markup present in the link to
determine the appropriate behavior.      </xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
         <xsd:enumeration value="none">
            <xsd:annotation>
               <xsd:documentation>
The behavior of an application traversing to the ending resource is unconstrained by this
specification. No other markup is present to help the application determine the
appropriate behavior.      </xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
      </xsd:restriction>
   </xsd:simpleType>
   <!-- CLASS: TypeKind -->
   <xsd:simpleType name="TypeKind">
      <xsd:restriction base="xsd:string">
         <xsd:enumeration value="simple"/>
         <xsd:enumeration value="extended"/>
         <xsd:enumeration value="title"/>
         <xsd:enumeration value="arc"/>
         <xsd:enumeration value="resource"/>
         <xsd:enumeration value="locator"/>
         <xsd:enumeration value="none"/>
      </xsd:restriction>
   </xsd:simpleType>
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- <<structure>> Classes                        -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- CLASS: SimpleLink -->
   <xsd:complexType name="SimpleLink">
      <xsd:annotation>
         <xsd:documentation>
SimpleLink represents a simple XML link with a required xlink:href attribute that links
to an arbitrary URI, and an optional xlink:title attribute that may hold the name of the
link target.
Furthermore, other standard xlink attributes may be specified.      </xsd:documentation>
      </xsd:annotation>
      <xsd:attributeGroup ref="xlink:simpleLink"/>
   </xsd:complexType>
</xsd:schema>
```

11.2 The Core.PrimitiveTypes Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:Core="http://www.esa.int/2005/10/Core/PrimitiveTypes"
targetNamespace="http://www.esa.int/2005/10/Core/PrimitiveTypes"
elementFormDefault="unqualified" attributeFormDefault="unqualified">
   <xsd:annotation>
      <xsd:documentation>
UML Model Information:
    Filename:        file:/c:/PEllsiepen/cvswork/Metamodel/uml/flat-Smdl.xml
    Last modified:   Fri Oct 28 01:20:00 CEST 2005
    Model name:      Data
    Model comments:
Author:Peter Ellsiepen, Peter Fritzen.
Created:.
Title:SMDL Specification (SMP 2.0, Issue 1.2).
Comment:This is the specification of the Simulation Model Definition Language (SMDL).

    XMI version:     1.2
    XMI exporter:    MagicDraw UML 9.5
    XMI metamodel:   UML 1.4

XSLT Processing Information:
    Processing date: 2005-10-28+02:00 01:20:00+02:00
    XSLT Processor:  SAXON 8.1.1 from Saxonica
    XSLT Version:    2.0
    XSLT Stylesheet: xmi-to-xsd.xslt
      </xsd:documentation>
   </xsd:annotation>
   <!-- ======================================== -->
   <!-- PACKAGE: PrimitiveTypes -->
   <!-- ======================================== -->
   <xsd:annotation>
      <xsd:documentation>This package holds the primitive types, both for use in UML
modelling and as source for generating XML Schema and other target platforms.
</xsd:documentation>
   </xsd:annotation>
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- <<primitive>> DataTypes                  -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- DATATYPE: AnySimple -->
   <xsd:simpleType name="AnySimple">
      <xsd:annotation>
         <xsd:documentation>container for any simple type</xsd:documentation>
      </xsd:annotation>
      <xsd:union memberTypes="xsd:string xsd:boolean xsd:byte xsd:short xsd:int xsd:long
xsd:unsignedByte xsd:unsignedShort xsd:unsignedInt xsd:unsignedLong xsd:float xsd:double
xsd:dateTime xsd:duration"/>
   </xsd:simpleType>
   <!-- DATATYPE: Bool -->
   <xsd:simpleType name="Bool">
      <xsd:annotation>
         <xsd:documentation>boolean with true or false</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:boolean"/>
   </xsd:simpleType>
   <!-- DATATYPE: Char8 -->
   <xsd:simpleType name="Char8">
      <xsd:annotation>
         <xsd:documentation>8 bit character</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:string">
         <xsd:length value="1"/>
      </xsd:restriction>
   </xsd:simpleType>
   <!-- DATATYPE: DateTime -->
   <xsd:simpleType name="DateTime">
      <xsd:annotation>
         <xsd:documentation>point in time in nanoseconds
relative to MJD 2000+0.5</xsd:documentation>
      </xsd:annotation>
```

```
      <xsd:restriction base="xsd:dateTime"/>
   </xsd:simpleType>
   <!-- DATATYPE: Duration -->
   <xsd:simpleType name="Duration">
      <xsd:annotation>
         <xsd:documentation>duration in nanoseconds</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:duration"/>
   </xsd:simpleType>
   <!-- DATATYPE: Float32 -->
   <xsd:simpleType name="Float32">
      <xsd:annotation>
         <xsd:documentation>32 bit single-precision float</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:float"/>
   </xsd:simpleType>
   <!-- DATATYPE: Float64 -->
   <xsd:simpleType name="Float64">
      <xsd:annotation>
         <xsd:documentation>64 bit double-precision float</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:double"/>
   </xsd:simpleType>
   <!-- DATATYPE: Int16 -->
   <xsd:simpleType name="Int16">
      <xsd:annotation>
         <xsd:documentation>16 bit   signed integer</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:short"/>
   </xsd:simpleType>
   <!-- DATATYPE: Int32 -->
   <xsd:simpleType name="Int32">
      <xsd:annotation>
         <xsd:documentation>32 bit   signed integer</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:int"/>
   </xsd:simpleType>
   <!-- DATATYPE: Int64 -->
   <xsd:simpleType name="Int64">
      <xsd:annotation>
         <xsd:documentation>64 bit   signed integer</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:long"/>
   </xsd:simpleType>
   <!-- DATATYPE: Int8 -->
   <xsd:simpleType name="Int8">
      <xsd:annotation>
         <xsd:documentation>8 bit   signed integer</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:byte"/>
   </xsd:simpleType>
   <!-- DATATYPE: String8 -->
   <xsd:simpleType name="String8">
      <xsd:annotation>
         <xsd:documentation>8 bit character string</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:string"/>
   </xsd:simpleType>
   <!-- DATATYPE: UInt16 -->
   <xsd:simpleType name="UInt16">
      <xsd:annotation>
         <xsd:documentation>16 bit unsigned integer</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:unsignedShort"/>
   </xsd:simpleType>
   <!-- DATATYPE: UInt32 -->
   <xsd:simpleType name="UInt32">
      <xsd:annotation>
         <xsd:documentation>32 bit unsigned integer</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:unsignedInt"/>
   </xsd:simpleType>
   <!-- DATATYPE: UInt64 -->
   <xsd:simpleType name="UInt64">
```

```
      <xsd:annotation>
         <xsd:documentation>64 bit unsigned integer</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:unsignedLong"/>
   </xsd:simpleType>
   <!-- DATATYPE: UInt8 -->
   <xsd:simpleType name="UInt8">
      <xsd:annotation>
         <xsd:documentation>8 bit unsigned integer</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="xsd:unsignedByte"/>
   </xsd:simpleType>
</xsd:schema>
```

11.3 The Core.Elements Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:Elements="http://www.esa.int/2005/10/Core/Elements"
xmlns:Core="http://www.esa.int/2005/10/Core/PrimitiveTypes"
xmlns:xlink="http://www.w3.org/1999/xlink"
targetNamespace="http://www.esa.int/2005/10/Core/Elements"
elementFormDefault="unqualified" attributeFormDefault="unqualified">
   <xsd:annotation>
      <xsd:documentation>
UML Model Information:
    Filename:        file:/c:/PEllsiepen/cvswork/Metamodel/uml/flat-Smdl.xml
    Last modified:   Fri Oct 28 01:20:00 CEST 2005
    Model name:      Data
    Model comments:
Author:Peter Ellsiepen, Peter Fritzen.
Created:.
Title:SMDL Specification (SMP 2.0, Issue 1.2).
Comment:This is the specification of the Simulation Model Definition Language (SMDL).

    XMI version:     1.2
    XMI exporter:    MagicDraw UML 9.5
    XMI metamodel:   UML 1.4

XSLT Processing Information:
    Processing date: 2005-10-28+02:00 01:20:00+02:00
    XSLT Processor:  SAXON 8.1.1 from Saxonica
    XSLT Version:    2.0
    XSLT Stylesheet: xmi-to-xsd.xslt
      </xsd:documentation>
   </xsd:annotation>
   <!-- ======================================== -->
   <!-- IMPORT COMPONENT: Core.PrimitiveTypes -->
   <!-- ======================================== -->
   <xsd:import namespace="http://www.esa.int/2005/10/Core/PrimitiveTypes"
schemaLocation="../Core/PrimitiveTypes.xsd"/>
   <!-- ======================================== -->
   <!-- IMPORT COMPONENT: Core.xlink -->
   <!-- ======================================== -->
   <xsd:import namespace="http://www.w3.org/1999/xlink"
schemaLocation="../Core/xlink.xsd"/>
   <!-- ======================================== -->
   <!-- PACKAGE: Elements -->
   <!-- ======================================== -->
   <xsd:annotation>
      <xsd:documentation>This package provides basic modelling elements and common
mechanisms.</xsd:documentation>
   </xsd:annotation>
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- <<string>> Classes                       -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- CLASS: Description -->
   <xsd:simpleType name="Description">
      <xsd:annotation>
         <xsd:documentation>A Description is a human-readable string describing an
entity.</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="Core:String8"/>
   </xsd:simpleType>
   <!-- CLASS: 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.

Note: Identifier is currently 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="xlink:ID">
```

```
         <xsd:minLength value="1"/>
      </xsd:restriction>
   </xsd:simpleType>
   <!-- CLASS: Name -->
   <xsd:simpleType name="Name">
      <xsd:annotation>
         <xsd:documentation>A Name is a human-readable string  identifying an entity.
This can be used, for example, in GUI applications.

A Name must start with a character, and is limited to characters, digits, and the
underscore ("_").</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="Core:String8">
         <xsd:maxLength value="32"/>
         <xsd:pattern value="[a-zA-Z][a-zA-Z0-9_]*"/>
      </xsd:restriction>
   </xsd:simpleType>
   <!-- CLASS: UUID -->
   <xsd:simpleType name="UUID">
      <xsd:annotation>
         <xsd:documentation>A Universally Unique Identifier or UUID takes the form of
hexadecimal integers separated by hyphens, following the pattern 8-4-4-4-12 as defined by
the Open Group.</xsd:documentation>
      </xsd:annotation>
      <xsd:restriction base="Core:String8">
         <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>
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- <<structure>> Classes                        -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- CLASS: Comment -->
   <xsd:complexType name="Comment">
      <xsd:annotation>
         <xsd:documentation>A Comment 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
element.

Note: See Comment in UML2.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Elements:Metadata"/>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- 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 elements to allow
identification of documents.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Elements:NamedElement">
            <xsd:attribute name="Title" type="Core:String8" use="optional"/>
            <xsd:attribute name="Date" type="Core:DateTime" use="optional"/>
            <xsd:attribute name="Creator" type="Core:String8" use="optional"/>
            <xsd:attribute name="Version" type="Core:String8" use="optional"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- 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. links to external documentation, 3d animations, technical drawings, CAD models,
etc.), based on the XML linking language.

Note: This is similar to the xsd:documentation element in XML Schema.</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>
   <!-- CLASS: Element -->
   <xsd:complexType name="Element" abstract="true">
      <xsd:annotation>
         <xsd:documentation>An Element is the common base for almost all other language
elements. It serves as an abstract base class and does not introduce any attributes.

Note: In UML2, an element is a constituent of a model, see 3.6 in UML2 Infrastructure.
</xsd:documentation>
      </xsd:annotation>
   </xsd:complexType>
   <!-- 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>
   <!-- CLASS: NamedElement -->
   <xsd:complexType name="NamedElement" abstract="true">
      <xsd:annotation>
         <xsd:documentation>A 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:complexContent>
         <xsd:extension base="Elements:Element">
            <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:extension>
      </xsd:complexContent>
   </xsd:complexType>
</xsd:schema>
```

11.4 The Core.Types Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:Elements="http://www.esa.int/2005/10/Core/Elements"
xmlns:Types="http://www.esa.int/2005/10/Core/Types"
xmlns:Core="http://www.esa.int/2005/10/Core/PrimitiveTypes"
xmlns:xlink="http://www.w3.org/1999/xlink"
targetNamespace="http://www.esa.int/2005/10/Core/Types" elementFormDefault="unqualified"
attributeFormDefault="unqualified">
   <xsd:annotation>
      <xsd:documentation>
UML Model Information:
    Filename:        file:/c:/PEllsiepen/cvswork/Metamodel/uml/flat-Smdl.xml
    Last modified:   Fri Oct 28 01:20:00 CEST 2005
    Model name:      Data
    Model comments:
Author:Peter Ellsiepen, Peter Fritzen.
Created:.
Title:SMDL Specification (SMP 2.0, Issue 1.2).
Comment:This is the specification of the Simulation Model Definition Language (SMDL).

    XMI version:     1.2
    XMI exporter:    MagicDraw UML 9.5
    XMI metamodel:   UML 1.4

XSLT Processing Information:
    Processing date: 2005-10-28+02:00 01:20:00+02:00
    XSLT Processor:  SAXON 8.1.1 from Saxonica
    XSLT Version:    2.0
    XSLT Stylesheet: xmi-to-xsd.xslt
      </xsd:documentation>
   </xsd:annotation>
   <!-- ======================================== -->
   <!-- IMPORT COMPONENT: Core.Elements -->
   <!-- ======================================== -->
   <xsd:import namespace="http://www.esa.int/2005/10/Core/Elements"
schemaLocation="../Core/Elements.xsd"/>
   <!-- ======================================== -->
   <!-- IMPORT COMPONENT: Core.PrimitiveTypes -->
   <!-- ======================================== -->
   <xsd:import namespace="http://www.esa.int/2005/10/Core/PrimitiveTypes"
schemaLocation="../Core/PrimitiveTypes.xsd"/>
   <!-- ======================================== -->
   <!-- IMPORT COMPONENT: Core.xlink -->
   <!-- ======================================== -->
   <xsd:import namespace="http://www.w3.org/1999/xlink"
schemaLocation="../Core/xlink.xsd"/>
   <!-- ======================================== -->
   <!-- PACKAGE: Types -->
   <!-- ======================================== -->
   <xsd:annotation>
      <xsd:documentation>This package provides basic types and typing mechanisms,
together with appropriate value specification mechanisms.</xsd:documentation>
   </xsd:annotation>
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- Enumerations                             -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- ENUMERATION: ParameterDirectionKind -->
   <xsd:simpleType name="ParameterDirectionKind">
      <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:restriction>
   </xsd:simpleType>
   <!-- ENUMERATION: VisibilityKind -->
   <xsd:simpleType name="VisibilityKind">
      <xsd:restriction base="xsd:string">
         <xsd:enumeration value="public">
            <xsd:annotation>
               <xsd:documentation>The element is globally visible.</xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
         <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:restriction>
   </xsd:simpleType>
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- <<structure>> Classes                        -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- 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 language element. This provides a similar mechanism
as tagged values in UML, xsd:appinfo in XML Schema, or attributes in the .NET framework.
A possible application of using attributes could be to decorate an SMDL model with
information needed to guide a code generator, for example to taylor the mapping 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>Types:AttributeType</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
               <xsd:element name="Value" type="Types:Value"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: AttributeType -->
   <xsd:complexType name="AttributeType">
      <xsd:annotation>
         <xsd:documentation>An AttributeType defines a new type available for adding
attributes to elements. The Usage element defines to which Metaclasses 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>Types:ValueType</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
               <xsd:element name="Default" type="Types:Value"/>
               <xsd:element name="Usage" type="Core:String8" minOccurs="0"
maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="AllowMultiple" type="Core:Bool" default="false"
use="optional"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: Field -->
   <xsd:complexType name="Field">
      <xsd:annotation>
         <xsd:documentation>A field is a feature that is typed by a value type, and that
may have a Default value.
The View and State attributes define how the field is published to the simulation
environment. Only fields with a View value of true are visible in the Run-Time
Environment. Only fields with a State of true are stored using external persistence. If
both flags are set to false, then the field may not be published at all. By default, both
attributes are set to true.
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>Types:ValueType</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
               <xsd:element name="Default" type="Types:Value" minOccurs="0"/>
            </xsd:sequence>
            <xsd:attribute name="View" type="Core:Bool" default="true" use="optional"/>
            <xsd:attribute name="State" type="Core:Bool" default="true" use="optional"/>
            <xsd:attribute name="Input" type="Core:Bool" default="false" use="optional"/>
            <xsd:attribute name="Output" type="Core:Bool" default="false"
use="optional"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: FieldValue -->
   <xsd:complexType name="FieldValue">
      <xsd:annotation>
         <xsd:documentation>A FieldValue links to the defining Field and contains a value
holding the actual Value.</xsd:documentation>
      </xsd:annotation>
      <xsd:sequence>
         <xsd:element name="Field" type="xlink:SimpleLink">
            <xsd:annotation>
               <xsd:documentation>Link destination type: Types:Field</xsd:documentation>
               <xsd:appinfo>Types:Field</xsd:appinfo>
            </xsd:annotation>
         </xsd:element>
         <xsd:element name="Value" type="Types:Value"/>
      </xsd:sequence>
   </xsd:complexType>
   <!-- CLASS: LanguageType -->
   <xsd:complexType name="LanguageType" abstract="true">
      <xsd:annotation>
```

```
         <xsd:documentation>A LanguageType is the abstract base metaclass for value types
(where instances are defined by their value), and references to value types. Please note
that SMDL Catalogues define reference types (where instances are defined by their
reference, i.e. their location in memory) as being derived from language type as well. As
reference types are specific to software systems, they are not introduced in
Core.Types.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:Type"/>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: NativeType -->
   <xsd:complexType name="NativeType">
      <xsd:annotation>
         <xsd:documentation>A native type specifies a type with any number of platform
mappings.</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>
   <!-- CLASS: Operation -->
   <xsd:complexType name="Operation">
      <xsd:annotation>
         <xsd:documentation>An Operation may have an arbitrary number of parameters, and
the Type is its return type. If the type is absent, the operation is a void function
(procedure) without return value.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:VisibilityElement">
            <xsd:sequence>
               <xsd:element name="Type" type="xlink:SimpleLink" minOccurs="0">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type:
Types:LanguageType</xsd:documentation>
                     <xsd:appinfo>Types:LanguageType</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
               <xsd:element name="Parameter" type="Types:Parameter" minOccurs="0"
maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: Parameter -->
   <xsd:complexType name="Parameter">
      <xsd:annotation>
         <xsd:documentation>A Parameter has a type and a Direction. The parameter
direction may be
· 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, or
· out: the parameter is write-only to the operation, i.e. its value is unspecified on
call, and must be set by the operation; or
· inout: the parameter must be specified on call, and may be changed by the operation.
When referencing a value type, a parameter may have an additional Default value, which
can be used by languages that support default values.
</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>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>
   <!-- CLASS: PlatformMapping -->
   <xsd:complexType name="PlatformMapping">
      <xsd:annotation>
         <xsd:documentation>A platform mapping specifies how a native type is mapped to a
particular platform.</xsd:documentation>
      </xsd:annotation>
      <xsd:attribute name="Name" type="Core:String8" 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="Core:String8" use="required">
         <xsd:annotation>
            <xsd:documentation>Name of the type on the platform.</xsd:documentation>
         </xsd:annotation>
      </xsd:attribute>
      <xsd:attribute name="Namespace" type="Core:String8" use="optional">
         <xsd:annotation>
            <xsd:documentation>Namespace on the platform.</xsd:documentation>
         </xsd:annotation>
      </xsd:attribute>
      <xsd:attribute name="Location" type="Core:String8" 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>
   <!-- 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 Integer, Float and
Enumeration types.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:ValueType"/>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- 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 may have a Uuid element representing a Universally
Unique Identifier (UUID). 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:sequence>
               <xsd:element name="Uuid" type="Elements:UUID" minOccurs="0"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: Value -->
   <xsd:complexType name="Value" abstract="true">
      <xsd:annotation>
         <xsd:documentation>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.</xsd:documentation>
      </xsd:annotation>
   </xsd:complexType>
   <!-- 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>Types:ValueType</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- 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>
   <!-- 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"
default="private" use="optional"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- ========================================= -->
   <!-- PACKAGE: ValueTypes -->
   <!-- ========================================= -->
   <xsd:annotation>
      <xsd:documentation>This package provides mechanisms to specify value
types.</xsd:documentation>
   </xsd:annotation>
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- <<structure>> Classes                     -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- 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.</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>Types:ValueType</xsd:appinfo>
                 </xsd:annotation>
              </xsd:element>
           </xsd:sequence>
           <xsd:attribute name="Size" type="Core:Int64" use="required"/>
        </xsd:extension>
     </xsd:complexContent>
  </xsd:complexType>
  <!-- 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.</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>
  <!-- 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="Core:Int32" use="required"/>
        </xsd:extension>
     </xsd:complexContent>
  </xsd:complexType>
  <!-- CLASS: Float -->
  <xsd:complexType name="Float">
     <xsd:annotation>
        <xsd:documentation>A Float type represents floating point values of type Double,
but 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. Furthermore the Unit element can hold a physical unit that can be used
by applications to ensure physical unit integrity across models.</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>Types:PrimitiveType</xsd:appinfo>
                 </xsd:annotation>
              </xsd:element>
           </xsd:sequence>
           <xsd:attribute name="Minimum" type="Core:Float64" use="optional"/>
           <xsd:attribute name="Maximum" type="Core:Float64" use="optional"/>
           <xsd:attribute name="MinInclusive" type="Core:Bool" default="true"
use="optional"/>
           <xsd:attribute name="MaxInclusive" type="Core:Bool" default="true"
use="optional"/>
           <xsd:attribute name="Unit" type="Core:String8" use="optional"/>
        </xsd:extension>
     </xsd:complexContent>
  </xsd:complexType>
  <!-- CLASS: Integer -->
  <xsd:complexType name="Integer">
     <xsd:annotation>
```

```
         <xsd:documentation>An Integer type represents integer values of type Int32, but
with a given range of valid values (via the Minimum and Maximum
attributes).</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>Types:PrimitiveType</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
            <xsd:attribute name="Minimum" type="Core:Int64" use="optional"/>
            <xsd:attribute name="Maximum" type="Core:Int64" use="optional"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- 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. A primitive type references a NativeType, which specifies
the platform mapping.
Typically, this mechanism is not used to define new types.
</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:SimpleType">
            <xsd:sequence>
               <xsd:element name="NativeType" type="xlink:SimpleLink">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type:
Types:NativeType</xsd:documentation>
                     <xsd:appinfo>Types:NativeType</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: String -->
   <xsd:complexType name="String">
      <xsd:annotation>
         <xsd:documentation>A String type represents fixed Length string values base on
Char. The String language element defines an Array of Char 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). Nevertheless, strings are used internally for names, which
are supposed to be constant during a simulation.
</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:ValueType">
            <xsd:attribute name="Length" type="Core:Int64" use="required"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- 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.
</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:ValueType">
            <xsd:sequence>
```

```
               <xsd:element name="Field" type="Types:Field" minOccurs="0"
maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- ======================================== -->
   <!-- PACKAGE: Values -->
   <!-- ======================================== -->
   <xsd:annotation>
      <xsd:documentation>This package provides mechanisms to specify values of value
types.</xsd:documentation>
   </xsd:annotation>
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- <<structure>> Classes                   -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- CLASS: ArrayValue -->
   <xsd:complexType name="ArrayValue">
      <xsd:annotation>
         <xsd:documentation>An array value 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>
   <!-- CLASS: SimpleValue -->
   <xsd:complexType name="SimpleValue">
      <xsd:annotation>
         <xsd:documentation>The SimpleValue metaclass is used for values of simple types.
As each simple type corresponds to a different XML Schema type, the Value element is not
typed.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:Value">
            <xsd:sequence>
               <xsd:element name="Value" type="Core:AnySimple"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: StringValue -->
   <xsd:complexType name="StringValue">
      <xsd:annotation>
         <xsd:documentation>A StringValue holds a value for a string, represented by the
Value attribute. As opposed to the array value, the string value uses a single string
instead of an array of values.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:Value">
            <xsd:sequence>
               <xsd:element name="Value" type="Core:String8"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: StructureValue -->
   <xsd:complexType name="StructureValue">
      <xsd:annotation>
         <xsd:documentation>A StructureValue holds field values for all fields of the
corresponding structure type.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:Value">
            <xsd:sequence>
               <xsd:element name="FieldValue" type="Types:FieldValue" minOccurs="0"
maxOccurs="unbounded"/>
            </xsd:sequence>
```

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

11.5 The Smdl.Catalogue Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:Elements="http://www.esa.int/2005/10/Core/Elements"
xmlns:Types="http://www.esa.int/2005/10/Core/Types"
xmlns:Core="http://www.esa.int/2005/10/Core/PrimitiveTypes"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:Catalogue="http://www.esa.int/2005/10/Smdl/Catalogue"
targetNamespace="http://www.esa.int/2005/10/Smdl/Catalogue"
elementFormDefault="unqualified" attributeFormDefault="unqualified">
   <xsd:annotation>
      <xsd:documentation>
UML Model Information:
    Filename:        file:/c:/PEllsiepen/cvswork/Metamodel/uml/gen/flat-Smdl.xml
    Last modified:   Fri Oct 28 01:20:00 CEST 2005
    Model name:      Data
    Model comments:
Author:Peter Ellsiepen, Peter Fritzen.
Created:.
Title:SMDL Specification (SMP 2.0, Issue 1.2).
Comment:This is the specification of the Simulation Model Definition Language (SMDL).

    XMI version:     1.2
    XMI exporter:    MagicDraw UML 9.5
    XMI metamodel:   UML 1.4

XSLT Processing Information:
    Processing date: 2005-10-28+02:00 01:20:00+02:00
    XSLT Processor:  SAXON 8.1.1 from Saxonica
    XSLT Version:    2.0
    XSLT Stylesheet: xmi-to-xsd.xslt
      </xsd:documentation>
   </xsd:annotation>
   <!-- ========================================= -->
   <!-- IMPORT COMPONENT: Core.Elements -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.esa.int/2005/10/Core/Elements"
schemaLocation="../Core/Elements.xsd"/>
   <!-- ========================================= -->
   <!-- IMPORT COMPONENT: Core.Types -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.esa.int/2005/10/Core/Types"
schemaLocation="../Core/Types.xsd"/>
   <!-- ========================================= -->
   <!-- IMPORT COMPONENT: Core.PrimitiveTypes -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.esa.int/2005/10/Core/PrimitiveTypes"
schemaLocation="../Core/PrimitiveTypes.xsd"/>
   <!-- ========================================= -->
   <!-- IMPORT COMPONENT: Core.xlink -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.w3.org/1999/xlink"
schemaLocation="../Core/xlink.xsd"/>
   <!-- ========================================= -->
   <!-- PACKAGE: Catalogue -->
   <!-- ========================================= -->
   <xsd:annotation>
      <xsd:documentation>SCHEMA: Smdl/Catalogue.xsd

This schema defines the language for the specification of SMDL Catalogue
documents.</xsd:documentation>
   </xsd:annotation>
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- <<XSDelement>> Objects                 -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- OBJECT: 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>
```

```
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- Enumerations                                 -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- ENUMERATION: AccessKind -->
   <xsd:simpleType name="AccessKind">
      <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>
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- <<structure>> Classes                        -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- 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>Types:LanguageType</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- 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>
   <!-- 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.</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:
Catalogue:Class</xsd:documentation>
                     <xsd:appinfo>Catalogue:Class</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
               <xsd:element name="Property" type="Catalogue:Property" minOccurs="0"
maxOccurs="unbounded"/>
               <xsd:element name="Operation" type="Types:Operation" minOccurs="0"
maxOccurs="unbounded"/>
               <xsd:element name="Association" type="Catalogue:Association" minOccurs="0"
maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="Abstract" type="Core:Bool" default="false"
use="optional"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- 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 elements 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 allows specifying a default model (DefaultModel). SMDL support
tools may use this during instantiation (i.e. creation of an assembly) to select an
initial implementation for newly created contained elements.
</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>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>Catalogue:Model</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
            <xsd:attribute name="Lower" type="Core:Int64" default="1"/>
            <xsd:attribute name="Upper" type="Core:Int64" default="1"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: EntryPoint -->
   <xsd:complexType name="EntryPoint">
      <xsd:annotation>
         <xsd:documentation>An EntryPoint is a named element of a Model. 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 should have their Input attribute set to true) and Output fields (which should
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>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>Types:Field</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: EventSink -->
   <xsd:complexType name="EventSink">
      <xsd:annotation>
         <xsd:documentation>An EventSink is used to specify that a Model can consume 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>Catalogue:EventType</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: EventSource -->
   <xsd:complexType name="EventSource">
      <xsd:annotation>
         <xsd:documentation>An EventSource is used to specify that a Model publishes a
specific event under a given name. The Multicast attribute can be used to specify whether
(in an Assembly) any number of sinks can connect to the source (the default), or only a
single sink can connect (Multicast=false).</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>Catalogue:EventType</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
            <xsd:attribute name="Multicast" type="Core:Bool" default="true"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- 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>Types:SimpleType</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: Interface -->
   <xsd:complexType name="Interface">
      <xsd:annotation>
         <xsd:documentation>An Interface is an abstract reference type that serves as a
contract in a loosely coupled architecture. It has the ability to contain properties and
operations (inherited from ReferenceType). An interface may inherit from other interfaces
(interface inheritance), which is represented via the Base links.
</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>Catalogue:Interface</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: Model -->
   <xsd:complexType name="Model">
      <xsd:annotation>
         <xsd:documentation>The Model Metaclass is a reference type and hence inherits
properties and operations from its base class. In addition, it provides various optional
elements in order to allow various different modelling approaches. As a Model
semantically forms a deployable unit (a component), it may use the available component
mechanisms as specified in the SMP 2.0 Component Model.
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 events 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 models (Aggregation).
For an Event based design, a Model may support inter-model events via the EventSource and
EventSink 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: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:Model</xsd:documentation>
                     <xsd:appinfo>Catalogue:Model</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>Catalogue:Interface</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
               <xsd:element name="Container" type="Catalogue:Container" minOccurs="0"
maxOccurs="unbounded"/>
               <xsd:element name="Reference" type="Catalogue:Reference" minOccurs="0"
maxOccurs="unbounded"/>
               <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="NestedType" type="Types:ValueType" minOccurs="0"
maxOccurs="unbounded"/>
               <xsd:element name="Field" type="Types:Field" minOccurs="0"
maxOccurs="unbounded"/>
               <xsd:element name="Association" type="Catalogue:Association" minOccurs="0"
maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- 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>
   <!-- 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 LanguageType. However, the semantics is very different: A property can
be assigned an Access attribute limiting access to one of readWrite, readOnly, and
writeOnly. Typically, this is mapped to one or two operations, called the "setter" and
"getter" of the property.
Furthermore, a Property can be assigned a Category attribute to be used as ordering or
filtering criterion in applications, e.g. in a property grid.</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>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>Types:Field</xsd:appinfo>
                  </xsd:annotation>
```

```
               </xsd:element>
            </xsd:sequence>
            <xsd:attribute name="Access" type="Catalogue:AccessKind"
default="readWrite"/>
            <xsd:attribute name="Category" type="Core:String8" default="Properties"
use="optional"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- 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 models that can be referenced is specified via the Interface link.
The Lower and Upper attributes specify the multiplicity, i.e. the number of possibly held
references to elements 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>Catalogue:Interface</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
            <xsd:attribute name="Lower" type="Core:Int64" default="1"/>
            <xsd:attribute name="Upper" type="Core:Int64" default="1"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: ReferenceType -->
   <xsd:complexType name="ReferenceType" abstract="true">
      <xsd:annotation>
         <xsd:documentation>A ReferenceType may hold any number of properties and
operations.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Types:LanguageType">
            <xsd:sequence>
               <xsd:element name="Property" type="Catalogue:Property" minOccurs="0"
maxOccurs="unbounded"/>
               <xsd:element name="Operation" type="Types:Operation" minOccurs="0"
maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
</xsd:schema>
```

11.6 The Smdl.Assembly Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:Elements="http://www.esa.int/2005/10/Core/Elements"
xmlns:Types="http://www.esa.int/2005/10/Core/Types"
xmlns:Core="http://www.esa.int/2005/10/Core/PrimitiveTypes"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:Assembly="http://www.esa.int/2005/10/Smdl/Assembly"
xmlns:Catalogue="http://www.esa.int/2005/10/Smdl/Catalogue"
targetNamespace="http://www.esa.int/2005/10/Smdl/Assembly"
elementFormDefault="unqualified" attributeFormDefault="unqualified">
   <xsd:annotation>
      <xsd:documentation>
UML Model Information:
    Filename:        file:/c:/PEllsiepen/cvswork/Metamodel/uml/gen/flat-Smdl.xml
    Last modified:   Fri Oct 28 01:20:00 CEST 2005
    Model name:      Data
    Model comments:
Author:Peter Ellsiepen, Peter Fritzen.
Created:.
Title:SMDL Specification (SMP 2.0, Issue 1.2).
Comment:This is the specification of the Simulation Model Definition Language (SMDL).

    XMI version:     1.2
    XMI exporter:    MagicDraw UML 9.5
    XMI metamodel:   UML 1.4

XSLT Processing Information:
    Processing date: 2005-10-28+02:00 01:20:00+02:00
    XSLT Processor:  SAXON 8.1.1 from Saxonica
    XSLT Version:    2.0
    XSLT Stylesheet: xmi-to-xsd.xslt
      </xsd:documentation>
   </xsd:annotation>
   <!-- ========================================= -->
   <!-- IMPORT COMPONENT: Core.Elements -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.esa.int/2005/10/Core/Elements"
schemaLocation="../Core/Elements.xsd"/>
   <!-- ========================================= -->
   <!-- IMPORT PACKAGE: Core.Types -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.esa.int/2005/10/Core/Types"
schemaLocation="../Core/Types.xsd"/>
   <!-- ========================================= -->
   <!-- IMPORT PACKAGE: Core.PrimitiveTypes -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.esa.int/2005/10/Core/PrimitiveTypes"
schemaLocation="../Core/PrimitiveTypes.xsd"/>
   <!-- ========================================= -->
   <!-- IMPORT PACKAGE: Core.xlink -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.w3.org/1999/xlink"
schemaLocation="../Core/xlink.xsd"/>
   <!-- ========================================= -->
   <!-- IMPORT PACKAGE: Smdl.Catalogue -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.esa.int/2005/10/Smdl/Catalogue"
schemaLocation="../Smdl/Catalogue.xsd"/>
   <!-- ========================================= -->
   <!-- PACKAGE: Assembly -->
   <!-- ========================================= -->
   <xsd:annotation>
      <xsd:documentation>SCHEMA: Smdl/Assembly.xsd

This schema defines the language for the specification of SMDL Assembly
documents.</xsd:documentation>
   </xsd:annotation>
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- <<XSDelement>> Objects                 -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- OBJECT: 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>
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- <<structure>> Classes                        -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- CLASS: Assembly -->
   <xsd:complexType name="Assembly">
      <xsd:annotation>
         <xsd:documentation>An Assembly is a document that is typed by a Model of a
catalogue. It is similar to a model instance. This is intended, as it will allow
replacing child model instances by sub-assemblies later if needed. For the same reason,
an assembly may specify which Implementation shall be used by specifying a Universally
Unique Identifier (UUID).
For each field of the referenced model, the Assembly can specify a FieldValue.
Depending on the containers of the model, the Assembly can hold a number of ModelInstance
elements.
An Assembly can link the references, event sinks and input fields of the referenced Model
to model instances via the Link elements.
</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Assembly:AssemblyNode">
            <xsd:attribute name="Title" type="Core:String8" use="optional"/>
            <xsd:attribute name="Date" type="Core:DateTime" use="optional"/>
            <xsd:attribute name="Creator" type="Core:String8" use="optional"/>
            <xsd:attribute name="Version" type="Core:String8" use="optional"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: AssemblyNode -->
   <xsd:complexType name="AssemblyNode" abstract="true">
      <xsd:complexContent>
         <xsd:extension base="Elements:NamedElement">
            <xsd:sequence>
               <xsd:element name="Model" type="xlink:SimpleLink">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type:
Catalogue:Model</xsd:documentation>
                     <xsd:appinfo>Catalogue:Model</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
               <xsd:element name="Implementation" type="Elements:UUID" minOccurs="0"/>
               <xsd:element name="ModelInstance" type="Assembly:ModelInstance"
minOccurs="0" maxOccurs="unbounded"/>
               <xsd:element name="FieldValue" type="Types:FieldValue" minOccurs="0"
maxOccurs="unbounded"/>
               <xsd:element name="Link" type="Assembly:Link" minOccurs="0"
maxOccurs="unbounded"/>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: EventLink -->
   <xsd:complexType name="EventLink">
      <xsd:annotation>
         <xsd:documentation>An EventLink resolves an event sink of a model instance 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 Publisher 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 event sink and event source, respectively.
To be semantically correct, the EventSource of the Publisher and the 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>Catalogue:EventSink</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>Assembly:AssemblyNode</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>Catalogue:EventSource</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: FieldLink -->
   <xsd:complexType name="FieldLink">
      <xsd:annotation>
         <xsd:documentation>A FieldLink resolves an input field of a model instance 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.
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 fields, 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>Types:Field</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
               <xsd:element name="Source" type="xlink:SimpleLink">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type:
Assembly:AssemblyNode</xsd:documentation>
                     <xsd:appinfo>Assembly:AssemblyNode</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>Types:Field</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: InterfaceLink -->
   <xsd:complexType name="InterfaceLink">
      <xsd:annotation>
```

```
         <xsd:documentation>An InterfaceLink resolves a reference of a model instance in
an assembly. Therefore, it links to the Reference 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 allow an assembly to be interpreted without the associated catalogue, the
xlink:title attribute of the Reference link shall contain the reference name (i.e. the
name of the Reference element in the catalogue).
In order to uniquely identify the link source, the InterfaceLink links to the Provider
model instance, which provides (via its Model) a matching Interface.
</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>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>Assembly:AssemblyNode</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: Link -->
   <xsd:complexType name="Link" abstract="true">
      <xsd:annotation>
         <xsd:documentation>The Link Metaclass does not introduce new attributes, but
serves as a base class for derived Metaclasses.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Elements:NamedElement"/>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: ModelInstance -->
   <xsd:complexType name="ModelInstance">
      <xsd:annotation>
         <xsd:documentation>A ModelInstance represents an instance of a model. Therefore,
it has to link to a Model. 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. To allow creating a run-time model instance, the model instance needs to
specify as well which Implementation shall be used when loading the assembly into a
simulation environment. This is done by specifying a Universally Unique Identifier
(UUID).
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.
For each field of the referenced model, the ModelInstance can specify a FieldValue.
Depending on the containers of the model, the ModelInstance can hold a number of child
model instances.
A ModelInstance can link the references, event sinks and input fields of the referenced
Model to model instances via the Link elements.
</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Assembly:AssemblyNode">
            <xsd:sequence>
               <xsd:element name="Container" type="xlink:SimpleLink">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type:
Catalogue:Container</xsd:documentation>
                     <xsd:appinfo>Catalogue:Container</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
```

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

11.7 The Smdl.Schedule Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:Elements="http://www.esa.int/2005/10/Core/Elements"
xmlns:Types="http://www.esa.int/2005/10/Core/Types"
xmlns:Core="http://www.esa.int/2005/10/Core/PrimitiveTypes"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:Assembly="http://www.esa.int/2005/10/Smdl/Assembly"
xmlns:Catalogue="http://www.esa.int/2005/10/Smdl/Catalogue"
xmlns:Schedule="http://www.esa.int/2005/10/Smdl/Schedule"
targetNamespace="http://www.esa.int/2005/10/Smdl/Schedule"
elementFormDefault="unqualified" attributeFormDefault="unqualified">
   <xsd:annotation>
      <xsd:documentation>
UML Model Information:
    Filename:        file:/c:/PEllsiepen/cvswork/Metamodel/uml/gen/flat-Smdl.xml
    Last modified:   Fri Oct 28 01:20:00 CEST 2005
    Model name:      Data
    Model comments:
Author:Peter Ellsiepen, Peter Fritzen.
Created:.
Title:SMDL Specification (SMP 2.0, Issue 1.2).
Comment:This is the specification of the Simulation Model Definition Language (SMDL).

    XMI version:     1.2
    XMI exporter:    MagicDraw UML 9.5
    XMI metamodel:   UML 1.4

XSLT Processing Information:
    Processing date: 2005-10-28+01:00 01:20:00+02:00
    XSLT Processor:  SAXON 8.1.1 from Saxonica
    XSLT Version:    2.0
    XSLT Stylesheet: xmi-to-xsd.xslt
      </xsd:documentation>
   </xsd:annotation>
   <!-- ========================================= -->
   <!-- IMPORT COMPONENT: Core.Elements -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.esa.int/2005/10/Core/Elements"
schemaLocation="../Core/Elements.xsd"/>
   <!-- ========================================= -->
   <!-- IMPORT COMPONENT: Core.Types -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.esa.int/2005/10/Core/Types"
schemaLocation="../Core/Types.xsd"/>
   <!-- ========================================= -->
   <!-- IMPORT COMPONENT: Core.PrimitiveTypes -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.esa.int/2005/10/Core/PrimitiveTypes"
schemaLocation="../Core/PrimitiveTypes.xsd"/>
   <!-- ========================================= -->
   <!-- IMPORT COMPONENT: Core.xlink -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.w3.org/1999/xlink"
schemaLocation="../Core/xlink.xsd"/>
   <!-- ========================================= -->
   <!-- IMPORT COMPONENT: Smdl.Assembly -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.esa.int/2005/10/Smdl/Assembly"
schemaLocation="../Smdl/Assembly.xsd"/>
   <!-- ========================================= -->
   <!-- IMPORT COMPONENT: Smdl.Catalogue -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.esa.int/2005/10/Smdl/Catalogue"
schemaLocation="../Smdl/Catalogue.xsd"/>
   <!-- ========================================= -->
   <!-- PACKAGE: Schedule -->
   <!-- ========================================= -->
   <xsd:annotation>
      <xsd:documentation>SCHEMA: Smdl/Schedule.xsd
```

```
This schema defines mechanisms to describe the Scheduling of entry points in an SMDL
Assembly.</xsd:documentation>
   </xsd:annotation>
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- <<XSDelement>> Objects                       -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- OBJECT: 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>
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- Enumerations                                -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- ENUMERATION: TimeKind -->
   <xsd:simpleType name="TimeKind">
      <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 typically progresses with epoch time.</xsd:documentation>
            </xsd:annotation>
         </xsd:enumeration>
      </xsd:restriction>
   </xsd:simpleType>
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- <<structure>> Classes                       -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- CLASS: Activity -->
   <xsd:complexType name="Activity" abstract="true">
      <xsd:annotation>
         <xsd:documentation>An Activity links an EntryPoint defined in a Model of a
Catalogue to a specific 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="Elements:NamedElement"/>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: EpochEvent -->
   <xsd:complexType name="EpochEvent">
      <xsd:complexContent>
         <xsd:extension base="Schedule:Event">
            <xsd:attribute name="EpochTime" type="Core:DateTime"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: Event -->
   <xsd:complexType name="Event" abstract="true">
      <xsd:annotation>
```

```
         <xsd:documentation>An Event is a Trigger that is called by the scheduler
service. The CycleTime and Count are for cyclic events. A Count of 0 is for a single
e0u101 ?nt, a positive number indicates additional triggers, while a Count of -1
indicates that the event is to be repeated forever (default). For a cyclic event, the
CycleTime needs to be positive.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Elements:NamedElement">
            <xsd:sequence>
               <xsd:element name="CycleTime" type="Core:Duration"/>
               <xsd:element name="Count" type="Core:Int64" minOccurs="0" default="-1"/>
               <xsd:element name="Task" type="xlink:SimpleLink" minOccurs="0"
maxOccurs="unbounded">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type:
Schedule:Task</xsd:documentation>
                     <xsd:appinfo>Schedule:Task</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: MissionEvent -->
   <xsd:complexType name="MissionEvent">
      <xsd:complexContent>
         <xsd:extension base="Schedule:Event">
            <xsd:attribute name="MissionTime" type="Core:Duration"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- 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 (TimedEvent elements) triggering these
tasks.</xsd:documentation>
      </xsd:annotation>
      <xsd:complexContent>
         <xsd:extension base="Elements:Document">
            <xsd:sequence>
               <xsd:element name="EpochTime" type="Core: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="Core: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>
   <!-- CLASS: SimulationEvent -->
   <xsd:complexType name="SimulationEvent">
      <xsd:complexContent>
         <xsd:extension base="Schedule:Event">
            <xsd:attribute name="SimulationTime" type="Core:Duration"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: SubTask -->
   <xsd:complexType name="SubTask">
      <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>Schedule:Task</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
               <xsd:element name="Task" type="xlink:SimpleLink">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type:
Schedule:Task</xsd:documentation>
                     <xsd:appinfo>Schedule:Task</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- 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>
   <!-- CLASS: Transfer -->
   <xsd:complexType name="Transfer">
      <xsd:complexContent>
         <xsd:extension base="Schedule:Activity">
            <xsd:sequence>
               <xsd:element name="SourceNode" type="xlink:SimpleLink">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type:
Assembly:AssemblyNode</xsd:documentation>
                     <xsd:appinfo>Assembly:AssemblyNode</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: Trigger -->
   <xsd:complexType name="Trigger">
      <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>Assembly:FieldLink</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>Assembly:AssemblyNode</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>Catalogue:EntryPoint</xsd:appinfo>
                 </xsd:annotation>
              </xsd:element>
           </xsd:sequence>
        </xsd:extension>
     </xsd:complexContent>
  </xsd:complexType>
  <!-- CLASS: ZuluEvent -->
  <xsd:complexType name="ZuluEvent">
     <xsd:complexContent>
        <xsd:extension base="Schedule:Event">
           <xsd:attribute name="ZuluTime" type="Core:DateTime"/>
        </xsd:extension>
     </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```

11.8 The Smdl.Package Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:Core="http://www.esa.int/2005/10/Core/PrimitiveTypes"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:Types="http://www.esa.int/2005/10/Core/Types"
xmlns:Elements="http://www.esa.int/2005/10/Core/Elements"
xmlns:Catalogue="http://www.esa.int/2005/10/Smdl/Catalogue"
xmlns:Package="http://www.esa.int/2005/10/Smdl/Package"
targetNamespace="http://www.esa.int/2005/10/Smdl/Package"
elementFormDefault="unqualified" attributeFormDefault="unqualified">
   <xsd:annotation>
      <xsd:documentation>
UML Model Information:
    Filename:        file:/c:/PEllsiepen/cvswork/Metamodel/uml/flat-Smdl.xml
    Last modified:   Fri Oct 28 01:20:00 CEST 2005
    Model name:      Data
    Model comments:
Author:Peter Ellsiepen, Peter Fritzen.
Created:.
Title:SMDL Specification (SMP 2.0, Issue 1.2).
Comment:This is the specification of the Simulation Model Definition Language (SMDL).

    XMI version:     1.2
    XMI exporter:    MagicDraw UML 9.5
    XMI metamodel:   UML 1.4

XSLT Processing Information:
    Processing date: 2005-10-28+02:00 01:20:00+02:00
    XSLT Processor:  SAXON 8.1.1 from Saxonica
    XSLT Version:    2.0
    XSLT Stylesheet: xmi-to-xsd.xslt
      </xsd:documentation>
   </xsd:annotation>
   <!-- ========================================= -->
   <!-- IMPORT COMPONENT: Core.PrimitiveTypes -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.esa.int/2005/10/Core/PrimitiveTypes"
schemaLocation="../Core/PrimitiveTypes.xsd"/>
   <!-- ========================================= -->
   <!-- IMPORT COMPONENT: Core.xlink -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.w3.org/1999/xlink"
schemaLocation="../Core/xlink.xsd"/>
   <!-- ========================================= -->
   <!-- IMPORT COMPONENT: Core.Types -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.esa.int/2005/10/Core/Types"
schemaLocation="../Core/Types.xsd"/>
   <!-- ========================================= -->
   <!-- IMPORT COMPONENT: Core.Elements -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.esa.int/2005/10/Core/Elements"
schemaLocation="../Core/Elements.xsd"/>
   <!-- ========================================= -->
   <!-- IMPORT COMPONENT: Smdl.Catalogue -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.esa.int/2005/10/Smdl/Catalogue"
schemaLocation="../Smdl/Catalogue.xsd"/>
   <!-- ========================================= -->
   <!-- PACKAGE: Package -->
   <!-- ========================================= -->

   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- <<XSDelement>> Objects                    -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- OBJECT: 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>
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- <<structure>> Classes                        -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- CLASS: Implementation -->
   <xsd:complexType name="Implementation">
      <xsd:complexContent>
         <xsd:extension base="Elements:Element">
            <xsd:sequence>
               <xsd:element name="Type" type="xlink:SimpleLink">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type:
Types:Type</xsd:documentation>
                     <xsd:appinfo>Types:Type</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
            </xsd:sequence>
            <xsd:attribute name="Uuid" type="Elements:UUID" use="optional"/>
         </xsd:extension>
      </xsd:complexContent>
   </xsd:complexType>
   <!-- CLASS: Package -->
   <xsd:complexType name="Package">
      <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>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>
```

11.9 The Smdl.Workspace Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:Core="http://www.esa.int/2005/10/Core/PrimitiveTypes"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:Elements="http://www.esa.int/2005/10/Core/Elements"
xmlns:Workspace="http://www.esa.int/2005/10/Smdl/Workspace"
targetNamespace="http://www.esa.int/2005/10/Smdl/Workspace"
elementFormDefault="unqualified" attributeFormDefault="unqualified">
   <xsd:annotation>
      <xsd:documentation>
UML Model Information:
    Filename:        file:/c:/PEllsiepen/cvswork/Metamodel/uml/flat-Smdl.xml
    Last modified:   Fri Oct 28 01:20:00 CEST 2005
    Model name:      Data
    Model comments:
Author:Peter Ellsiepen, Peter Fritzen.
Created:.
Title:SMDL Specification (SMP 2.0, Issue 1.2).
Comment:This is the specification of the Simulation Model Definition Language (SMDL).

    XMI version:     1.2
    XMI exporter:    MagicDraw UML 9.5
    XMI metamodel:   UML 1.4

XSLT Processing Information:
    Processing date: 2005-10-28+02:00 01:20:00+02:00
    XSLT Processor:  SAXON 8.1.1 from Saxonica
    XSLT Version:    2.0
    XSLT Stylesheet: xmi-to-xsd.xslt
      </xsd:documentation>
   </xsd:annotation>
   <!-- ========================================= -->
   <!-- IMPORT COMPONENT: Core.PrimitiveTypes -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.esa.int/2005/10/Core/PrimitiveTypes"
schemaLocation="../Core/PrimitiveTypes.xsd"/>
   <!-- ========================================= -->
   <!-- IMPORT COMPONENT: Core.xlink -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.w3.org/1999/xlink"
schemaLocation="../Core/xlink.xsd"/>
   <!-- ========================================= -->
   <!-- IMPORT COMPONENT: Core.Elements -->
   <!-- ========================================= -->
   <xsd:import namespace="http://www.esa.int/2005/10/Core/Elements"
schemaLocation="../Core/Elements.xsd"/>
   <!-- ========================================= -->
   <!-- PACKAGE: Workspace -->
   <!-- ========================================= -->

   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- <<XSDelement>> Objects                    -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- OBJECT: Workspace -->
   <xsd:element name="Workspace" type="Workspace:Workspace"/>
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- <<structure>> Classes                     -->
   <!-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~ -->
   <!-- CLASS: Workspace -->
   <xsd:complexType name="Workspace">
      <xsd:complexContent>
         <xsd:extension base="Elements:Document">
            <xsd:sequence>
               <xsd:element name="Document" type="xlink:SimpleLink" minOccurs="0"
maxOccurs="unbounded">
                  <xsd:annotation>
                     <xsd:documentation>Link destination type:
Elements:Document</xsd:documentation>
                     <xsd:appinfo>Elements:Document</xsd:appinfo>
                  </xsd:annotation>
               </xsd:element>
```

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