EUROPEAN SPACE AGENCY
DIRECTORATE OF OPERATIONS AND INFRASTRUCTURE

SMP 2.0 Handbook

EGOS-SIM-GEN-TN-0099

Issue 1 Revision 2

28 October 2005

This Page is Intentionally left Blank

ABSTRACT

This is the SMP2 Handbook. It provides a high-level view of SMP2 and introduces all its concepts together with their realisation in C++. As a handbook, this document is less formal than the SMP2 specification, which consists of a Metamodel to describe SMP2 models, a component model including simulation services, and a complete mapping of this specification to the ANSI/ISO C++ platform. Other platform mappings are foreseen for the future.

This document targets different users of SMP2, including but not limited to model developers.

© EUROPEAN SPACE AGENCY, 2005

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		

© EUROPEAN SPACE AGENCY, 2005

DOCUMENT STATUS SHEET

1. Issue	**2. Revision**	**3. Date**	**4. Reason for Change**
0	Beta 1	13 April 2004	New document structure for SMP2 documents.
0	Beta 2	09 July 2004	Completely new document based on feedback on revision Beta 2 of SMDL Specification.
0	RC 1	06 August 2004	Completion of sections 5, 6 and 7.
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.

DOCUMENT CHANGE RECORD

DOCUMENT CHANGE RECORD			**DCR NO-**	N/A
Changes from SMP 2.0 Handbook Issue 1 Revision 1 to SMP 2.0 Handbook Issue 1 Revision 2			**DATE**	28 October 2005
			ORIGINATOR	SMP CCB
			APPROVED BY	Niklas Lindman
1. PAGE	**2. PARAGRAPH**	**3. ISSUE**	**4. REASON FOR CHANGE**	
15	1.2	45	Pre-requisites for readers added.	
17	1.4	46	SMP Handbook and Alpha Specification moved from applicable documents to reference documents. Version of date of SMP2 documents updated.	
29	2.7.1	48	Reference to Component Model document added.	
37	2.7.6.2	41	Footnote added to clarify that two event mechanisms exist.	
38	2.7.7	51	Text updated.	
43	2.8.8	42	Data representation restrictions for C++ added.	
49	3		Counter model updated for updated component model.	
50	3.2	49	Reference to IModel section updated.	
53	3.5	52	Term “helper class” removed from document.	
73	5		Mapping to C++ updated according to C++ Mapping.	
89	6		Code examples updated for SMP2 Issue 1.2.	
97	7		Code examples and XML files updated for SMP2 Issue 1.2.	
122	8.5		SMDL terms updated for changes in Metamodel.	

© EUROPEAN SPACE AGENCY, 2005

TABLE OF CONTENTS

ABSTRACT **3**

1. INTRODUCTION **15**
1.1 Purpose 15
1.2 Scope 15
1.3 Definitions, acronyms and abbreviations 16
1.4 References 17
1.4.1 Applicable Documents 17
1.4.2 Reference Documents 17
1.5 Overview 18

2. CONCEPTS **19**
2.1 Background 19
2.2 Objectives 20
2.2.1 Portability of Models between Simulation Environments 20
2.2.2 Portability of Models between Platforms 20
2.2.3 Support for Modern Software Engineering Techniques 20
2.2.4 Model Reuse 20
2.2.5 Model Integration 21
2.2.6 Model Development Productivity 21
2.2.7 Integration of System Engineering Data 21
2.2.8 Configurable and Flexible Simulation 21
2.2.9 Support for Metadata 21
2.2.10 Use of Open Standards 21
2.3 Fundamental Concepts 22
2.3.1 Introduction 22
2.3.2 High-Level View 22
2.3.3 Platform Independence 24
2.3.4 Separation of Run-Time and Design-Time 24
2.3.5 Dynamic Configuration 24
2.3.6 Components and Interfaces 24
2.3.7 Inheritance 24
2.3.7.1 Implementation Inheritance 24
2.3.7.2 Interface Inheritance 25
2.3.7.3 Single vs. Multiple Inheritance 25
2.4 Elements of the Standard 26
2.5 Architecture 27
2.6 Operational Phases 28
2.7 SMP2 Mechanisms 29
2.7.1 Components and Model Hierarchy 29
2.7.2 Simulation Services 30
2.7.2.1 Service Acquisition Mechanism 30
2.7.2.2 Service Categories 31
2.7.2.2.1 Mandatory Services 32
2.7.2.2.2 Optional Services 32
2.7.2.2.3 User-defined Services 32
2.7.3 Simulation Time Kinds 32
2.7.3.1 Simulation Time 32
2.7.3.2 Epoch Time 33
2.7.3.3 Mission Time 34
2.7.3.4 Zulu Time 34
2.7.4 Model Entry Points 35
2.7.5 Model Publication 35
2.7.6 Model Interactions 36
2.7.6.1 Interfaces 36
2.7.6.1.1 Properties 37

© EUROPEAN SPACE AGENCY, 2005

2.7.6.2 Events....37
2.7.6.3 Dataflow....38
2.7.7 Managed Models....38
2.8 Model Development Guidelines....40
2.8.1 Provide Documentation....41
2.8.2 Use Standard Languages....41
2.8.3 Avoid Compiler Specific Language Extensions....41
2.8.4 Use Standard Libraries Only....42
2.8.5 Limit Use of Essential Non-standard Libraries....42
2.8.6 Use the SMP2 Interfaces to interact with the Simulation Environment....42
2.8.7 Avoid Direct Input / Output Operations....43
2.8.8 Do Not Rely on Internal Representation of Data....43
2.8.9 Avoid Global Data Declarations....43
2.8.10 Avoid Common Global Names....44
2.8.11 Enable Multiple Instances....44
2.8.12 Minimise Number of Model Interfaces....44
2.8.13 Simplify Model Interfaces Provided....44
2.8.14 Only Select Suitable Candidates for Reuse....45
2.9 SMP2 Model and Simulation Environment Features....46
2.9.1 Statically Configured Simulations....46
2.9.1.1 Model Interfaces....46
2.9.1.2 Simulation Environment Features....46
2.9.2 Dynamically Configured Simulations....47
2.9.2.1 Model Interfaces....47
2.9.2.2 Simulator Features....47
2.9.3 Optional Features....48
2.9.3.1 Optional Model Interfaces....48
2.9.3.2 Optional Simulation Environment Features....48

3. GETTING STARTED....49
3.1 An existing class....49
3.2 How to turn it into an SMP2 Model....50
3.3 How to publish data to the Environment....52
3.4 How to send a message to the Logger....53
3.5 How to add the model to the Scheduler....53
3.6 How to register with a global Event....55
3.7 The complete model....55
3.7.1 The definition file....56
3.7.2 The implementation file....57
3.8 How to turn it into a Managed Model....58

4. THE SIMULATION ENVIRONMENT....61
4.1 Simulation Environment State Diagram....61
4.1.1 The `Building` state....62
4.1.2 The `Connecting` state....62
4.1.3 The `Initialising` state....63
4.1.4 The `Standby` state....63
4.1.5 The `Executing` state....63
4.1.6 The `Storing` state....63
4.1.7 The `Restoring` state....64
4.1.8 The `Exiting` state....64
4.1.9 The `Aborting` state....64
4.2 Simulation Environment Interfaces....64
4.2.1 The `ISimulator` interface....65
4.2.1.1 The `Publish()` method....66
4.2.1.2 The `Configure()` method....66
4.2.1.3 The `Connect()` method....66
4.2.1.4 The `Initialise()` method....67

 © EUROPEAN SPACE AGENCY, 2005

4.2.1.5 The Run() method 67
4.2.1.6 The Hold() method 67
4.2.1.7 The Store(in String8 filename) method 67
4.2.1.8 The Restore(in String8 filename) method 67
4.2.1.9 The Exit() method 67
4.2.1.10 The Abort() method 67
4.2.1.11 The GetState() method 67
4.2.1.12 The GetModels() method 67
4.2.1.13 The GetModel(in String8 name) method 68
4.2.1.14 The AddModel(in IModel model) method 68
4.2.1.15 The GetServices() method 68
4.2.1.16 The GetService(in String8 name) method 68
4.2.1.17 The AddService(in IService Service) method 68
4.2.1.18 The GetLogger() method 68
4.2.1.19 The GetScheduler() method 68
4.2.1.20 The GetTimeKeeper() method 68
4.2.1.21 The GetEventManager() method 68
4.2.2 The IComposite interface 69
4.2.2.1 The Models container 69
4.2.2.2 The Services container 69
4.2.3 The IPublication interface 69
4.2.4 The IDynamicSimulator interface 70
4.2.4.1 The RegisterFactory(...) method 70
4.2.4.2 The CreateInstance(in Uuid implUuid) method 70
4.2.4.3 The GetFactory(in Uuid implUuid) method 70
4.2.4.4 The GetFactories(in Uuid specUuid) method 71
4.3 Simulation Services 71
4.3.1 Mandatory Services 71
4.3.1.1 Logger 71
4.3.1.2 Scheduler 71
4.3.1.3 Time Keeper 71
4.3.1.4 Event Manager 71
4.3.2 Optional Services 71
4.3.2.1 Resolver 71
4.3.3 User-defined Services 71

5. MODEL DESIGN AND DEVELOPMENT 73
5.1 Class-based Design 73
5.1.1 Design elements 73
5.1.1.1 Models 73
5.1.1.2 Fields 73
5.1.1.3 Operations 73
5.1.1.4 Entry Points 74
5.1.2 Mapping to C++ 74
5.1.2.1 Mapping of Models 74
5.1.2.2 Mapping of Fields 74
5.1.2.3 Mapping of Operations 75
5.1.2.4 Mapping of Entry Points 75
5.2 Interface-based Design 76
5.2.1 Design elements 76
5.2.1.1 Interfaces 76
5.2.1.2 Operations 76
5.2.1.3 Properties 76
5.2.2 Mapping to C++ 77
5.2.2.1 Mapping of Interfaces 77
5.2.2.2 Mapping of Operations 77
5.2.2.3 Mapping of Properties 77

© EUROPEAN SPACE AGENCY, 2005

5.3 Component-based Design 78
5.3.1 Design elements 79
5.3.1.1 Aggregation....79
5.3.1.2 Composition 79
5.3.2 Mapping to C++....80
5.3.2.1 Mapping of Aggregation 80
5.3.2.2 Mapping of Composition 81
5.4 Event-based Design 82
5.4.1 Design elements 82
5.4.1.1 Event Types 82
5.4.1.2 Event Sources....82
5.4.1.3 Event Sinks 82
5.4.2 Mapping to C++....83
5.4.2.1 Mapping of Event Types....83
5.4.2.2 Mapping of Event Sources 83
5.4.2.3 Mapping of Event Sinks....83
5.5 Dataflow-based Design....84
5.5.1 Additional Design elements 84
5.5.2 Design elements 84
5.5.2.1 Namespace 84
5.5.2.2 Types....84
5.5.2.2.1 Float....84
5.5.2.2.2 Integer 85
5.5.2.2.3 Enumeration 85
5.5.2.2.4 Array 85
5.5.2.2.5 String....85
5.5.2.2.6 Structure 85
5.5.2.2.7 Class 85
5.5.3 Mapping to C++....85
5.5.3.1 Mapping of Namespaces....85
5.5.3.2 Mapping of Types 85
5.5.3.2.1 Mapping of Float....85
5.5.3.2.2 Mapping of Integer....85
5.5.3.2.3 Mapping of Enumeration....86
5.5.3.2.4 Mapping of Array....86
5.5.3.2.5 Mapping of String 86
5.5.3.2.6 Mapping of Structure 86
5.5.3.2.7 Mapping of Class 87

6. MODEL INTEGRATION 89
6.1 Model Integration using Source Code 89
6.1.1 Adding a Root Model to the Simulator 89
6.1.2 Building a Hierarchy of Model Instances 90
6.1.3 Creating Interface Links between Model Instances 90
6.1.4 Creating Event Links between Model Instances 91
6.2 Model Integration using Assemblies 92
6.2.1 Model Factories 92
6.2.2 Building a Hierarchy of Model Instances 93
6.2.3 Creating Interface Links between Model Instances 94
6.2.4 Creating Event Links between Model Instances 95
6.2.5 Scheduling Entry Points of the Model Instances....95
6.2.5.1 Create a Task of Entry Points....95
6.2.5.2 Schedule a Task 96

7. MODEL EXAMPLES....97
7.1 Class-based Example....97
7.1.1 UML Design 97
7.1.2 Catalogue Elements....97
7.1.3 Catalogue File 98

 © EUROPEAN SPACE AGENCY, 2005

7.1.4 Model Definition File 98
7.2 Interface-based Example 99
7.2.1 UML Design 99
7.2.2 Catalogue Elements 99
7.2.3 Catalogue File 100
7.2.4 Model Definition Files 100
7.2.5 Assembly Elements 102
7.2.6 Assembly File 103
7.3 Component-based Example 104
7.3.1 UML Design 104
7.3.2 Catalogue Elements 104
7.3.3 Catalogue File 105
7.3.4 Model Definition Files 105
7.3.5 Assembly Elements 106
7.3.6 Assembly File 106
7.4 Event-based Example 106
7.4.1 UML Design 107
7.4.2 Catalogue Elements 107
7.4.3 Catalogue File 108
7.4.4 Model Definition Files 108
7.4.5 Assembly Elements 109
7.4.6 Assembly File 111

8. APPENDIX A: GLOSSARY 113
8.1 Introduction 113
8.2 Conventions 113
8.3 Models 113
8.4 SMP2 Terms and Definitions 113
8.5 SMDL Modelling Elements 122

9. APPENDIX B: NOTES ON SUPPORTING TOOLS 133
9.1 XSIM Prototype Tools 133
9.2 Simulation Development Life-Cycle 133
9.2.1 Model Design: The SMDL Catalogue 133
9.2.2 Model Development: Implementing Models 133
9.2.3 Model Integration: The SMDL Assembly 134
9.2.4 Model Execution: The SMDL Schedule 134
9.3 Other Support Tools 134

© EUROPEAN SPACE AGENCY, 2005

This Page is Intentionally left Blank

© EUROPEAN SPACE AGENCY, 2005

LIST OF FIGURES AND TABLES

Figure 2-1: SMP2 Models embedded into Common Concepts and Common Type System ... 22
Figure 2-2: SMP2 High-Level View ... 23
Figure 2-3: Typical SMP2 Architecture ... 27
Figure 2-4: SMP2 operational phases ... 28
Figure 2-5: UML Class Diagram: Different types of components ... 29
Figure 2-6: UML Class Diagram: Component containment ... 30
Figure 2-7: Simulation Services ... 30
Figure 2-8: UML Sequence Diagram: Service Acquisition Mechanism ... 31
Figure 2-9: Interface-based Design ... 37
Figure 2-10: Event-based Design ... 38
Figure 2-11: Dataflow-based Design ... 38
Figure 2-12: Typical Architecture using a Model Manager ... 40
Figure 3-1: CounterClass.h: Definition of Counter class ... 49
Figure 3-2: CounterClass.cpp: Implementation of Counter class ... 50
Figure 3-3: Counter.h: Definition of model Counter ... 51
Figure 3-4: Publishing a field ... 52
Figure 3-5: Querying the logger service ... 53
Figure 3-6: Logging a message to the logger ... 53
Figure 3-7: Private class implementing the IEntryPoint interface ... 54
Figure 3-8: Registering the Count method with the scheduler ... 54
Figure 3-9: Registering the Reset method with the event manager ... 55
Figure 3-10: Counter.h: Definition of Counter model ... 56
Figure 3-11: Counter.cpp: Implementation of Counter model ... 57
Figure 3-12: ManagedCounter.h ... 59
Figure 3-13: ManagedCounter.cpp (modifications only) ... 60
Figure 4-1: UML State Diagram: Simulation Environment State Diagram with State Transition Methods ... 61
Figure 4-2: Example of Simulation Environment implementing standard Simulation Environment interfaces... 65
Figure 4-3: UML Class Diagram: ISimulator interface ... 66
Figure 4-4: UML Class Diagram: IComposite, IContainer, IManagedContainer interfaces ... 69
Figure 5-1: UML Class Diagram: Class-based Model ... 73
Figure 5-2: UML Class Diagram: Interface-based Model ... 76
Figure 5-3: UML Class Diagram: Component-based Model ... 78
Figure 5-4: UML Class Diagram: Event-based model ... 82
Figure 7-1: UML Class Diagram: Design for Class-based Example ... 97
Figure 7-2: Catalogue Elements for Class-based Example ... 97
Figure 7-3: Catalogue File for Class-based Example ... 98
Figure 7-4: UML Class Diagram: Design for Interface-based Example ... 99
Figure 7-5: Catalogue Elements for Interface-based Example ... 99
Figure 7-6: Catalogue File for Interface-based Example ... 100
Figure 7-7: Assembly Elements for Interface-based Example ... 102
Figure 7-8: Example Configuration using Interface Links ... 102
Figure 7-9: Assembly File for Interface-based Example ... 103
Figure 7-10: UML Class Diagram: Design for Component-based Example ... 104
Figure 7-11: Catalogue Elements for Component-based Example ... 104
Figure 7-12: Catalogue File for Component-based Example ... 105
Figure 7-13: Assembly Elements for Component-based Example ... 106
Figure 7-14: Assembly File for Component-based Example ... 106
Figure 7-15: UML Class Diagram: Design for Event-based Example ... 107
Figure 7-16: Catalogue Elements for Event-based Example ... 107
Figure 7-17: Catalogue File for Event-based Example ... 108
Figure 7-18: Assembly Elements for Event-based Example ... 110
Figure 7-19: Example configuration using Event Links ... 110
Figure 7-20: Assembly File for Event-based Example ... 111
Figure 9-1: Possible Simulation Development Life-Cycle using XSIM Prototype Tools ... 134

© EUROPEAN SPACE AGENCY, 2005

This Page is Intentionally left Blank

© EUROPEAN SPACE AGENCY, 2005

1. INTRODUCTION

This document is the Handbook for the Simulation Model Portability 2 (**SMP2**) Standard. The standard as well as this handbook is an evolution of the Simulation Model Portability 1 (**SMP1**) Standard developed by and in use at the European Space Agency (**ESA**) [RD-1].

The document describes the general concepts behind the SMP2 Standard, and provides instructions for the accomplishment of the main tasks involved in using SMP2. Additional documents provide a complete technical reference of all elements of the standard: a Metamodel to define SMP2 models [AD-1], called the Simulation Model Definition Language (**SMDL**), a component model with a standard set of simulation services [AD-2], and a C++ mapping [AD-3], which describes the mapping of the platform independent models to the ANSI/ISO C++ target platform. Further platform mappings are foreseen for the future.

1.1 Purpose

The purpose of the SMP2 Standard is to promote portability of models among different simulation environments and operating systems, and to promote the reuse of simulation models. This document explains the main elements of the standard, outlines their application to different modelling approaches, and provides examples including source code.

1.2 Scope

This document is intended as an introduction into the SMP2 Standard. It is not a reference manual, i.e. it does not claim to be complete. The focus is on explaining the **usage** of the standard rather than the standard itself. Note that the content of this document is not normative as it mainly describes the normative elements of the standard (see 2.4) from different perspectives, targeting the following types of SMP2 users:

Simulator Designers: They have to build a simulator, i.e. they have to design and integrate it. We assume they have already an SMP2-compliant simulation environment, and have decided to use SMP2 models. They may choose different options for the design approach and supporting tools, as well as for model integration.

Typically, simulator designers would need tool support to effectively apply SMP2 (see Appendix B: Notes on Supporting Tools). Apart from that, they may want to pay special attention to section 2 (Concepts).

Model Developers: They work in conjunction with the simulator designers or they may be building a library of models. They need to understand the mechanisms of interfacing with the simulation environment as well as those for inter-model communication.

Model Developers with a background in SMP1 or C++ may start reading section 3 (Getting Started). Apart from this, model developers may pay special attention to section 2.7 (SMP2 Mechanisms). As model developers would typically make use of supporting tools (see Appendix B: Notes on Supporting Tools) for their day-to-day work, we foresee that a tool handbook – rather than this handbook – would be their main working material.

Tool Developers: They develop tools supporting the simulator designers or model developers. These tools may be stand-alone tools, e.g. an editor for designing a simulator or a validator for model validation, or tools interfacing SMDL to other tools, e.g. document generation, code generation, or import of models designed in other tools (for example based on the Unified Modelling Language (**UML**)).

Tool Developers may pay special attention to Appendix B: Notes on Supporting Tools.

Environment Furnishers: They want to adapt an existing simulation environment to be SMP2-compliant. They need to know which services they have to implement in their environment to make it SMP2-compliant, and how loading, initialisation, scheduling and persistence work.

Environment Furnishers should pay special attention to section 4 (The Simulation Environment).

© EUROPEAN SPACE AGENCY, 2005

Readers will benefit from the following pre-requisites:

- The concept of Object Orientation (**OO**), which is mandatory to understand SMP2.
- The Unified Modelling Language (**UML**) that has been used for most of the diagrams.
- The **C++** programming language that has been used for the example code.
- The eXtensible Markup Language (**XML**) that has been used for the SMDL example files.

1.3 Definitions, acronyms and abbreviations

AD	Applicable Document
API	Application Programming Interface
CCB	Configuration Control Board
CCM	CORBA Component Model
CDF	Concurrent Design Facility
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
DLL	Dynamic Link Library
DSO	Dynamic Shared Object
ESA	European Space Agency
ESOC	European Space Operations Centre
EuroSim	EUropean Real-time Operations SIMulator
GMT	Greenwich Mean Time
GPTB	Generic Project Test Bed
HITL	Hardware-In-The-Loop
IDL	Interface Definition Language
J2EE	Java 2 Enterprise Edition
MAEL	Mobile Aeronautics Education Laboratory
MDA	Model Driven Architecture
MDK	Model Development Kit
MJD	Modified Julian Date
N/A	Not Applicable
OMG	Object Management Group
OO	Object Oriented
PIM	Platform Independent Model
PSM	Platform Specific Model
RD	Reference Document
SIMSAT-2000	Software Infrastructure for the Modelling of SATellites - 2000
SMDL	Simulation Model Definition Language
SMI	Simulation Model Interface
SMP	Simulation Model Portability

© EUROPEAN SPACE AGENCY, 2005

SMP1	Simulation Model Portability 1
SMP2	Simulation Model Portability 2
SysML	Systems Modeling Language
TBC	To be confirmed
TBD	To be defined
UML	Unified Modelling Language
UTC	Coordinated Universal Time
UUID	Universally Unique Identifier
XML	Extensible Markup Language
XSIM	XML Based Simulation

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 Metamodel
EGOS-SIM-GEN-TN-0100, 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 Design Patterns – Elements of Reusable Object-Oriented Software
Erich Gamma, Richard Helm, Ralph E. Johnson, John Vlissides (The Gang of Four)
Addison-Wesley 1994, ISBN 0-201-63361-2

© EUROPEAN SPACE AGENCY, 2005

1.5 Overview

This section presents an overview of what is contained in the remainder of this document.

2 CONCEPTS

This section introduces the major concepts of the SMP2 Standard. All new users of the SMP2 can benefit from an understanding of the material in this section.

3 GETTING STARTED

This section starts with a sample class that is turned into an SMP2 model, and then step-by-step enhanced to make use of SMP2 services. This section especially addresses the Model Developer.

4 THE SIMULATION ENVIRONMENT

This section summarises mandatory and optional elements that each SMP2 simulation environment may provide. Further, it introduces certain states as well as state transitions the environment has to support.

5 MODEL DESIGN AND DEVELOPMENT

This section introduces different model design approaches. It starts with class-based design, then moves to interface-based design, which further is expanded for component-based design. In addition, both the event-based design and dataflow-based design are explained. For each design method, the SMDL elements needed are explained, with a summary of how this maps to C++.

6 MODEL INTEGRATION

When a library of model implementations is available, a configuration of model run-time instances has to be created. This process is called model integration, and explained in this section.

7 MODEL EXAMPLES

This section contains space related examples for the different design approaches. For each example, a UML design, the Catalogue Elements, the Catalogue file and the model definition files (header files) are shown. Where applicable, Assembly Elements and the Assembly file are shown as well.

8 APPENDIX A: GLOSSARY

This section provides a glossary of essential terms and definitions used throughout the SMP2 specification, as well as a glossary of all SMDL modelling elements.

9 APPENDIX B: NOTES ON SUPPORTING TOOLS

This section collects ideas and plans for tools that support the SMP2 standard. As such, tools are not part of the SMP2 standard itself, but rather intended to be built on top of the standard in order to support it.

© EUROPEAN SPACE AGENCY, 2005

2. CONCEPTS

This section introduces the major concepts of the SMP2 Standard. All new users of the SMP2 can benefit from an understanding of the material in this section.

2.1 Background

The European Space Agency has been involved in space simulation development for a number of years, and is developing simulations for a variety of applications including analysis, engineering operations preparation and training.

Different departments perform developments of these simulators, they are running on several different platforms and they are using different computer languages. A variety of subcontractors are involved in these projects and as a result a wide range of simulation models have been developed.

A Simulation Model Portability (**SMP**) study was performed to define a standard that addresses the issues related to portability and reuse of simulation models. The main outputs of this study were as follows:

- The SMP1 Handbook [RD-1], which describes the main SMP1 scope, concepts, purpose and objectives.
- A software implementation of SMP1 named Simulation Model Interface (**SMI**).

The main purpose of the SMP1 is to promote portability and reuse of simulation models [RD-1*, p. 19*]. These goals are shared and highly regarded by both the Agency and Industry. It is achieved via four objectives [RD-1*, p. 19*]:

- Minimise model interactions with environment
- Standardise the interface used by models
- Make the models own interface simple
- Make the model understandable for other developers

The SMI implementation fulfils these objectives with its simple and standardised method of communication between the Simulator models and the Simulator host. Due to this, the SMI has successfully been integrated with two of the main simulation infrastructures used within ESA, namely EuroSim and SIMSAT-2000 (previously SIMSAT-NT).

In addition to the SMI, the SMP1 also puts down guidelines for model development to ensure that it is as portable as possible. This includes issues like development guidelines, model design patterns, document templates, interchange data file formats and compliance tests. Hence the SMP1 provides a complete framework for developing reusable and portable models.

The SMP1 has now been successfully applied on a number of projects, including the following:

- Generic Project Test Bed (GPTB) at ESTEC
- Rosetta/Mars Express/Venus Express/Cryosat Operational Spacecraft Simulators for ESOC
- AQUA Payload Simulator (script based models) for NASA
- Galileo System Simulator Facility for ESTEC

Many of these projects have benefited from the SMP1, particularly with respect to portability of the models between different simulation environments. SMP1 is currently in its first issue, and as with all standards with this level of maturity, the application of the standard to real projects has also highlighted some areas of the SMP1 that could be improved. As described above, the SMI can be regarded as the realisation of parts of the SMP1 standard. Some of the limitations of the SMP1/SMI include:

- SMP1/SMI does standardise *inter-model communication* on the basis of dynamic invocation only, which is not well suited for many applications. Typically, an operational simulator is built using native inter-model communication, i.e. based on direct calls between models, possibly via interfaces, which can be checked at compile-time to ensure *type-safety*.
- SMP1/SMI does not provide support for *object-oriented* design (inheritance, methods) and interface- and component-based technology.
- SMP1/SMI *publication* calls are typically coded manually, which is error prone and tedious
- SMP1/SMI *scheduling* mechanisms are too primitive
- SMP1/SMI lacks support for additional *metadata* such as a description or a unit
- SMP1/SMI does not provide a concept to properly handle *initial values* of models
- SMP1/SMI does not provide the ability to access or change the *simulation state*
- SMP1/SMI does not support *dynamic configuration* of simulations

Recognising this and building on the knowledge of SMP1/SMI, the agency, in partnership with the industry, decided to launch a new initiative, which would address these limitations: the SMP2 standard.

2.2 Objectives

This section gives an overview of the objectives driving the SMP2 standard, which are basically an outcome of the user requirements for the new SMP2 standard defined by the SMP Configuration Control Board (**CCB**) in 2003. Some of these objectives have also been valid for SMP1/SMI (e.g. portability of models), while others have been identified in order to improve the SMP1 standard to be able to handle advanced user requirements (e.g. support for modern software engineering techniques).

2.2.1 Portability of Models between Simulation Environments

The standard shall support the portability of simulation models between different simulation environments by providing a standard interface between the simulation environment and the models. Models can therefore be plugged into a different simulation environment without requiring any modification to the model source code as the interface offered by the simulation environment to the model or by the model to the simulation environment does not change.

2.2.2 Portability of Models between Platforms

The standard shall support the portability of models between different operating systems and hardware. The standard defines guidelines to the model developer on how to avoid developing models that make use of the operating system – such as making calls to operating specific APIs – or hardware specific dependencies.

2.2.3 Support for Modern Software Engineering Techniques

The standard shall support and encourage the use of modern software engineering techniques and in particular object-orientation and component-based design. This ensures the maximum utilisation of modern techniques that have proven to improve software development productivity.

2.2.4 Model Reuse

Lack of model reuse is one of the main obstacles for improving productivity and medium- and long-term reliability in the development of simulators. Reuse therefore requires that models be designed to be reusable. It is one of the main aims of the standard to help the modeller in developing reusable models that can be deployed across different simulations, used across many different simulation environments and exchanged between different organisations and missions. Reuse requires the following:

- Breaking dependencies between models
- Well-defined interfaces that unambiguously describe how a model plugs into its environment

 © EUROPEAN SPACE AGENCY, 2005

- Self-describing models that expose their needs and features to the environment (both at design-time and at run-time)
- Definition of how a model is deployed on the target platform

Component-based technology is designed to address the above issues and is now the mainstream approach for the development of large complex software intensive systems. The standard therefore supports a component-based development approach to promote model reuse.

2.2.5 Model Integration

The standard shall support the integration of individual models to form a complete system simulation. Improved model integration is closely related to model reuse (see above), as it depends on well defined reusable models that can be plugged together based on standard mechanisms.

2.2.6 Model Development Productivity

The standard shall help reduce the effort required by the model developer to implement the infrastructure elements of a model, such that it can be integrated into the simulation environment or with other models. The modeller therefore shall focus on the development of the functional aspects of the model, and not on the infrastructure aspects.

2.2.7 Integration of System Engineering Data

The standard shall support the improved use of simulation in the system design process, by making it easier to assimilate system-engineering data, or even completely configure a simulation from this data. This improves both the fidelity of the simulation, as it is more tightly coupled to the engineering information defining the real product and also productivity, as the development of the simulation can be more automated.

2.2.8 Configurable and Flexible Simulation

The standard shall provide support for developing highly configurable and flexible simulations, such that the simulated system configuration can be rapidly changed or evolved. This is particularly important for early design phases, where the real system design is very fluid and many design considerations and trade-offs are being considered.

2.2.9 Support for Metadata

The standard shall provide support for specifying additional machine-readable information about a model, which may be used for purposes of automation (such as document generation) or improving the presentation of model information at both run-time and design-time.

2.2.10 Use of Open Standards

The standard shall be based on open standards, to eliminate dependencies on proprietary technologies, which may limit the portability of models or require the need for development run-time licences.

© EUROPEAN SPACE AGENCY, 2005

2.3 Fundamental Concepts

2.3.1 Introduction

The main purpose of SMP2 is to promote *platform independence*, *interoperability* and *reuse* of simulation models. In order to achieve these goals, two fundamental requirements can be formulated:

- **Common Concepts**: All SMP2 models must be built using common high-level concepts addressing fundamental modelling issues. This enables the development of models on an abstract level, which is essential for platform independence and reuse of models.
- **Common Type System**: All SMP2 models must be built upon a common type system. This enables that different models have a common understanding of the syntax and semantics of basic types, which is essential for interoperability between different models.

In other words, the first requirement specifies that all models are derived from the same fundamental concepts, and the second requirement specifies that all models be built upon common ground. Thus, specific models 'live' in between these two common layers as is shown in Figure 2-1

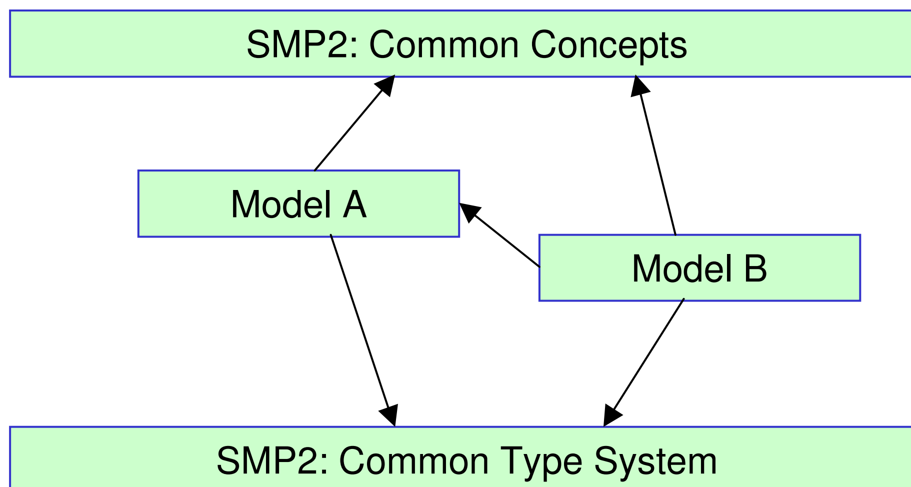

Figure 2-1: SMP2 Models embedded into Common Concepts and Common Type System

2.3.2 High-Level View

Figure 2-2 below presents a high-level view of all elements of SMP2. The figure has three main layers that correspond to different levels of abstraction.

The top layer represents the real world and deals with the system being modelled. SMP2 provides no modelling concepts in this layer, although it recognises that there is an important relationship with this layer, particularly with respect to the reuse of system engineering data within a simulation. We foresee both UML 2.0 and the Systems Modeling Language (**SysML**) to become widely used in this area (see http://www.omg.org/uml and http://www.sysml.org/, respectively).

The second layer represents the Platform Independent Model (**PIM**) of the simulated system, i.e. the specification level of SMP2 models. The main concern in this layer is to define model specifications (in a *Catalogue* file), to define how model instances are integrated and configured (in an *Assembly* file), and to define how model instances are scheduled (in a *Schedule* file). The Simulation Model Definition Language (SMDL) defines the format of all these XML files; see the Metamodel [AD-1] for details. Note that the Assembly and Schedule files in this layer are optional and mainly provide support for dynamic and flexible simulations that are configured and scheduled using external XML files independent of the model

 © EUROPEAN SPACE AGENCY, 2005

implementation (see section 6 on Model Integration and section 7 for Model Examples). The Catalogue file has added value for every SMP2 model as it specifies the exact interface of the model and thus may be seen as machine-readable model documentation.

The third layer is the Platform Specific Model (**PSM**) of the simulated system, i.e. the implementation level of SMP2 models, currently ANSI/ISO C++. Although SMP2 models may be implemented manually (see section 3), large parts of the model implementation is concerned with integration and may be generated from information in the Catalogue. To support this, the component model [AD-2] describes the standard interfaces of SMP2 components (models and services) and a set of standard simulation services. Finally, the C++ mapping [AD-3] describes the mapping of metamodel and component model artefacts into ANSI/ISO C++.

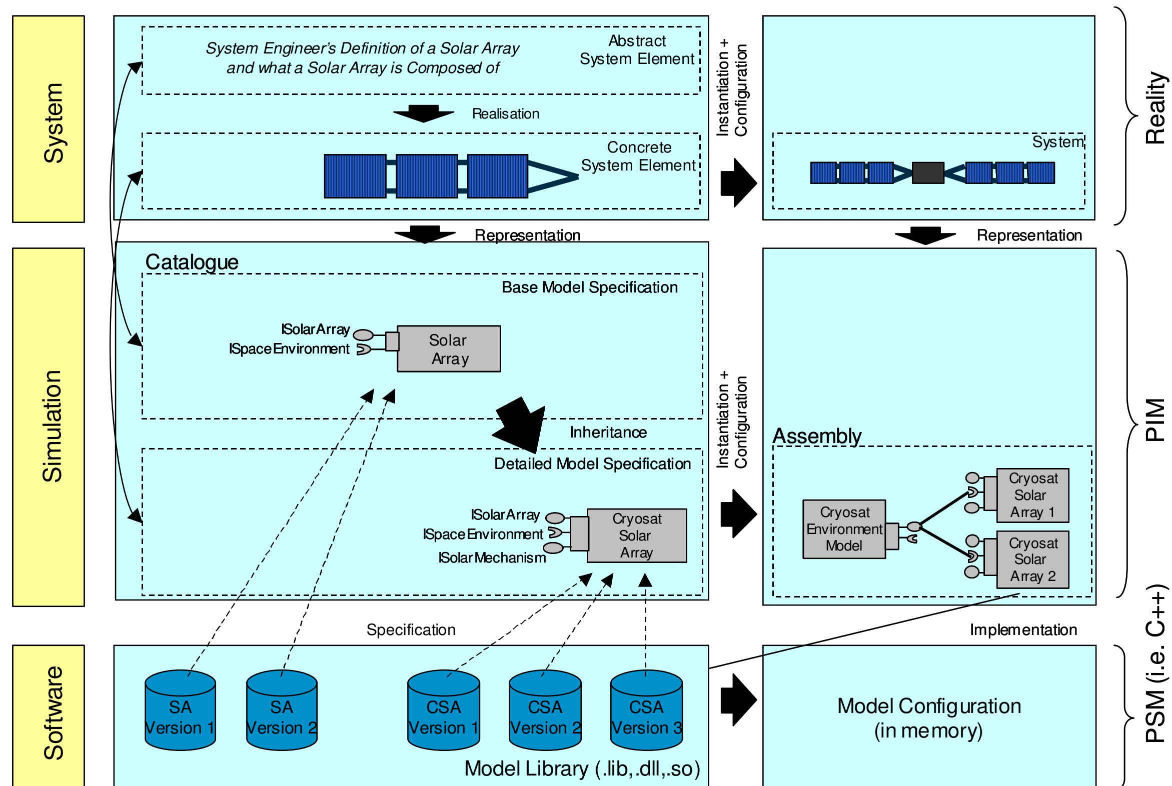

Figure 2-2: SMP2 High-Level View

Figure 2-2 above is split into two columns, where the left hand column corresponds to the definitions of the artefacts (corresponding to types in terms of object oriented (**OO**) technology) and the right hand column corresponds to the artefacts themselves (corresponding to instances in OO).

In the example, the modeller first describes in abstract form what a solar array is and what it is composed of. This information can be expressed, with respect to the concern of simulation, using a Model called *Solar Array* in an SMP2 Catalogue, as shown in Figure 2-2. This model may then be used to derive a more detailed model like the *Cryosat Solar Array* in the example, which defines the additional interface *ISolarMechanism*. In the Assembly, instances of these models may be specified and configured, resolving references like *ISpaceEnvironment* in the example, and selecting an appropriate model implementation (by UUID). On the implementation level (typically ANSI/ISO C++), appropriate model factories (see 4.2.4) point back to their associated model specification in the Catalogue (by UUID), allowing tools to provide intelligent user interfaces for the task of selecting a model implementation for a given model specification.

2.3.3 Platform Independence

A main objective of SMP2 is to promote platform independence of the models. This is achieved by the definition of a platform independent model (in the Catalogue) that can then be mapped into a platform specific model, such as C++ or Java. Further, the Assembly and Schedule files allow specifying integration, configuration and scheduling of model instances in a platform independent manner.

This concept is closely aligned with OMG's Model Driven Architecture (MDA).

2.3.4 Separation of Run-Time and Design-Time

Run-time models capture aspects of a running software system that can be different from aspects captured at design-time. A design-time model of a system might contain information such as authors, sizes, textual description, version info, etc. This is shown in Figure 2-2 above, where the right hand column of the figure shows the run-time view of the system (assembly and model configuration) and the left hand column shows the design time view of the system (catalogue and model library).

2.3.5 Dynamic Configuration

When building an assembly, models may be exchanged dynamically, based on interface compatibility (i.e. two models are compatible if they adhere to the same interface). There are a number of cases where dynamic configuration is useful.

Dynamic configuration could be used to support refinement in the development process. Initially a model implementation may be very simple, but later a more accurate model may be developed that can then be switched with the simple model.

Dynamic configuration may also be used to support models of different fidelity. For example, the user may select different implementation algorithms for an Orbit Propagator (e.g. Backward Euler, Runge-Kutta 4th order, or Adams-Bashforth-Moulton with variable order and step-size) depending on the required accuracy.

Dynamic configuration is also useful in support of test-bed simulations, where simulated models may be switched with a proxy to the real equipment – i.e. a component that simulates hardware equipment may be dynamically switched with a component that internally interfaces with the real equipment. This is useful during earlier phases of the development process when real equipments are not available, or when design changes are being planned and real equipment can be switched with a simulated version of the equipment that simulates the proposed design change.

2.3.6 Components and Interfaces

Components are the main building blocks of a component-based architecture. Interfaces provide the connection points that components offer to other components and to their environment. Furthermore, interfaces leverage the idea of an abstract contract that a realising component fulfils. This enables dynamic configuration and exchangeability of models (plug-in concept).

2.3.7 Inheritance

Inheritance is the main driver for reuse and abstraction in OO technologies. There are two fundamentally different types of inheritance, namely implementation inheritance and interface inheritance.

2.3.7.1 Implementation Inheritance

Implementation Inheritance is the inheritance of the implementation of a more general element. This *includes* the inheritance of the *interface* (see below).

© EUROPEAN SPACE AGENCY, 2005

Implementation inheritance is the standard inheritance mechanism in most OO programming languages. Inheriting from some existing implementation and overriding or extending functionality in the derived entity enables to reuse large quantities of existing code.

In SMP2, a typical example for implementation inheritance is the creation of a new model based on an existing model in order to change or add some functionality or to increase model fidelity. By applying implementation inheritance in the model catalogue we leverage code reuse in the derived model implementations.

2.3.7.2 Interface Inheritance

Interface Inheritance is the inheritance of interfaces of a more general element. The derived element only inherits the *contract* from the more general element, *not* its *implementation*.

In C++, interfaces as well as interface inheritance are not part of the language itself. Interfaces are usually implemented using abstract classes, which only contain pure virtual methods, and interface inheritance is implemented by deriving from such an abstract class.

In SMP2, a typical example for interface inheritance is a model that implements one or more interfaces, where the model is a concrete implementation of some functionality and the interfaces represent the abstract contract.

2.3.7.3 Single vs. Multiple Inheritance

It is well known that multiple inheritance of implementation, as supported for example by C++, leads to difficult situations when a class inherits from two classes that share a common base (diamond inheritance). Therefore, most current programming languages and platforms (e.g. Java, C#, CORBA) only support single inheritance of implementation, but multiple-inheritance of interfaces. This pattern allows for the greatest possible flexibility while at the same time maintaining simplicity and un-ambiguity in the implementation.

Therefore, SMP2 follows the same pattern and supports single inheritance of implementation and multiple inheritance of interfaces.

© EUROPEAN SPACE AGENCY, 2005

2.4 Elements of the Standard

The SMP 2.0 Standard includes a number of documents.

SMP 2.0 Handbook

This is the current document, which describes SMP2 from different perspectives.

This document is *not* normative.

SMP 2.0 Metamodel [AD-1]

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

This document is normative.

SMP 2.0 Component Model [AD-2]

This document provides a platform independent definition of the components used within an SMP2 simulation, where components include models and services, but also the simulator itself. A set of mandatory interfaces that every model has to implement is defined by the standard, and a number of optional interfaces for advanced component mechanisms are specified.

Additionally, this document includes a chapter on **Simulation Services**. Services are components that the models can use to interact with a Simulation Environment. SMP2 defines interfaces for mandatory services that every simulation environment has to provide, and optional interfaces for advanced functionality. Furthermore, mechanisms for user-defined services are explained.

This document is normative.

SMP 2.0 C++ Mapping [AD-3]

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

This document is normative.

 © EUROPEAN SPACE AGENCY, 2005

2.5 Architecture

The SMP2 Architecture covers two types of components: A **Simulation** with **Model Instances** that provide the application specific behaviour, and a **Simulation Environment** that provides **Simulation Services**. This architecture is depicted in Figure 2-3.

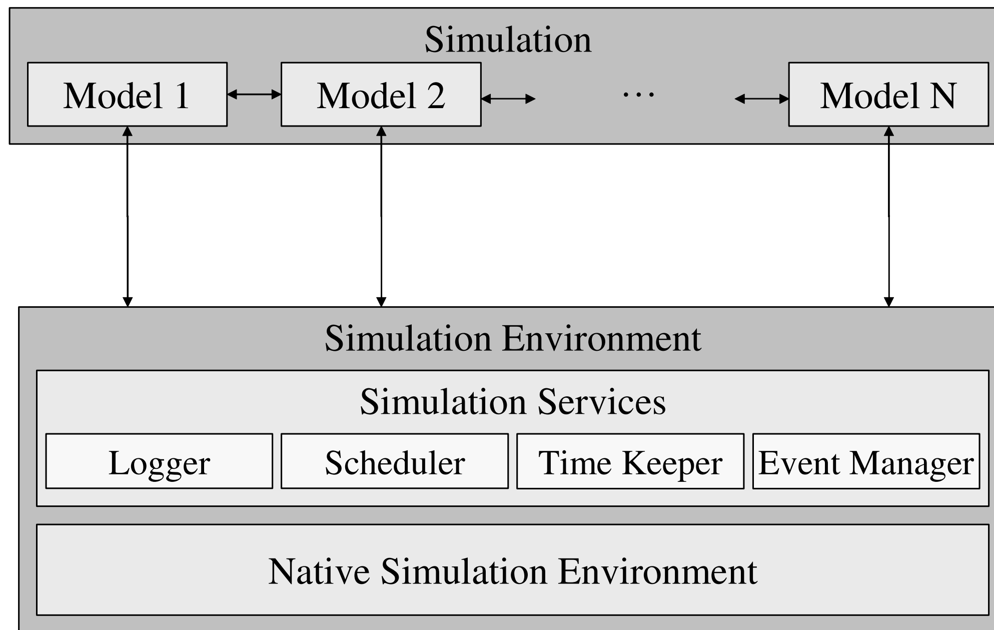

Figure 2-3: Typical SMP2 Architecture

Typically, a simulation environment is based on some existing **Native Simulation Environment** that it wraps (or adapts) to make it SMP2 compliant. The simulation environment has to provide four mandatory simulation services:

Logger: The logger allows logging messages of different kind consistently, for example information, event, warning, and error messages. It is used by services as well as by models.

Scheduler: The scheduler calls entry points based on timed or cyclic events. It closely depends on the time keeper service.

Time Keeper: This service provides the four different time kinds of SMP2: A relative simulation time, an absolute epoch time, a relative mission time, and Zulu time which relates to the clock time of the computer.

Event Manager: This service provides mechanisms for global asynchronous events: Event handlers can be registered, events can be broadcasted, and user-specific event types can be defined as well.

The arrows in Figure 2-3 indicate interaction between components. In SMP2, communication is typically performed via interfaces[1]. Two different types of interfaces can be identified in this architecture:

1. Interfaces between Model and Simulation Environment, and
2. Interfaces between Model and Model (inter-model communication).

All these interfaces need to be established before executing models. Therefore, SMP2 defines different operational phases.

---

[1] *Note that inter-model communication in SMP1 goes via the simulation environment based on the publication information (without compile-time checking), while in SMP2 it is direct via interfaces and compile-time checking.*

2.6 Operational Phases

The operational phases for a typical SMP2 system are illustrated below.

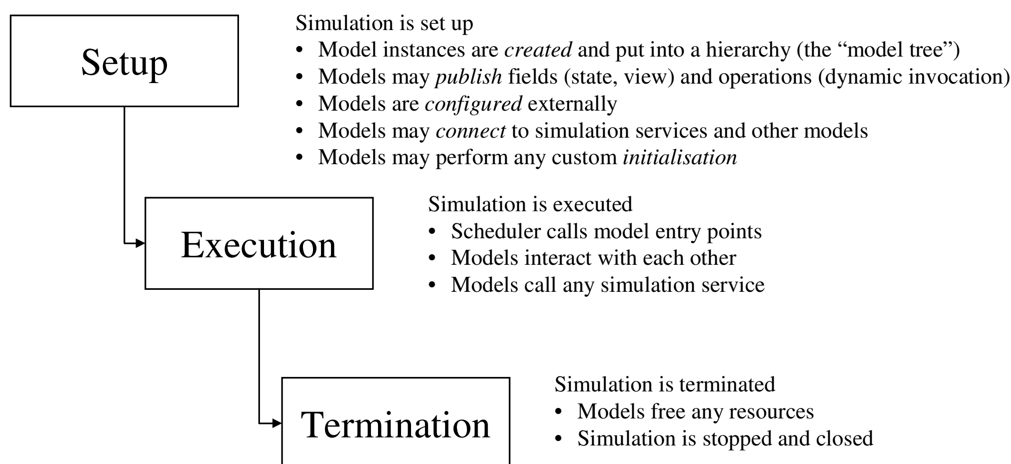

Figure 2-4: SMP2 operational phases

The diagram shows that the SMP2 system has three phases of operation, which are outlined below.

Setup: In SMP2, the core of a simulation is a hierarchical composition of model instances, called a configuration. This configuration needs to be assembled by *creating* all model instances from the available model implementations and building up a hierarchy of models (the "model tree").

SMP2 provides mechanisms to offload models from certain standardised tasks. These tasks include mechanisms to access fields externally (e.g. for visualisation purposes), and for storing and restoring simulation state vectors. To support these mechanisms, models can *publish* their fields (called "data items" in SMP1) to an external component which is passed to the `Publish()` method. Typically, the publication receiver is the simulation environment, but other scenarios are possible (e.g. in a distributed environment).

In dynamically configured simulations, the configuration is loaded from an SMP2 assembly file (see 2.7.7). The model manager needs to *configure* the model instances by setting initial values and by connecting the model instances. SMP2 defines various connection mechanisms to support different design approaches. In addition, each model instance can perform custom configuration steps in its `Configure()` method.

Before model instances can be executed in an active simulation, they get the chance to *connect* themselves to simulation services and/or to connection points of other models. The simulation environment traverses the model tree and calls the `Connect()` method of each model, passing a reference to the simulator. Models may use this to query any services they need, and to call any service operations. Further, they may resolve references to other model instances if needed.

Finally, models may need to perform custom *initialisation*. For this purpose, the simulation environment calls dedicated initialisation entry points that have been defined during the setup phase.

Execution: When the simulation setup phase has been completed, the simulation environment (namely the scheduler service) starts scheduling the model instances. Model instances can now use all simulation services, but they will typically interact with each other without involving the environment, depending on how they are inter-connected.

Termination: After a simulation has been executed, it needs to be terminated. This may either be done by properly cleaning up all resources used (exit), or by an abnormal termination (abort).

 © EUROPEAN SPACE AGENCY, 2005

2.7 SMP2 Mechanisms

As part of the standard, certain mechanisms have been defined allowing connecting components together, and exchanging information between them.

2.7.1 Components and Model Hierarchy

SMP2 promotes a component-based architecture, where each model instance is a component of its own[2]. Further, the simulation environment as well as each of its services is a component. This allows treating model instances similar to simulation services. One example is that every component has to provide a `Name` and `Description` on request. This holds for model instances as well as for simulation services.

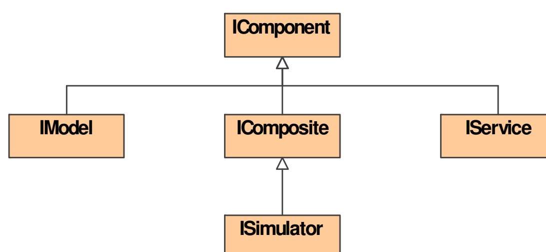

Figure 2-5: UML Class Diagram: Different types of components

In SMP2, all components build a hierarchical tree. Every component knows about and returns its parent composite on request (using the `GetParent()` method of the `IComponent` interface). This parent is either the simulation environment (implementing the `ISimulator` interface), or another model instance (implementing the `IModel` interface). With this mechanism, it is always possible to navigate up in the hierarchy. The simulation environment itself is the top-most component of a simulation, and hence the only component without a parent. A model instance that is a direct child of the simulation environment is called a **Root Model (Instance)**.

A component with child components is called a **Composite**, and provides access to all its children. The simulation environment is not only the parent of all root models, but as well the parent of the simulation services. Hence, the simulation environment has two types of children, which it maintains in two different **Containers**. When another component wants to query the simulation environment for its children, it can either use the convenience methods `GetModel()` or `GetService()` or it has to specify which of the containers it is interested in. In a similar way, a composite model may contain different types of child models, e.g. transmitters and receivers of a ground station. Again, when asking a model instance for its child model instances, a specific container has to be specified. To allow for this two-step implementation of asking for child components, the optional interface `IComposite` gives access to the available containers (as `IContainer`), and each container gives access to the child components.

[2] *See section 1.5 of the SMP2 Component Model [AD-2] for an introduction into the Component Model.*

© EUROPEAN SPACE AGENCY, 2005

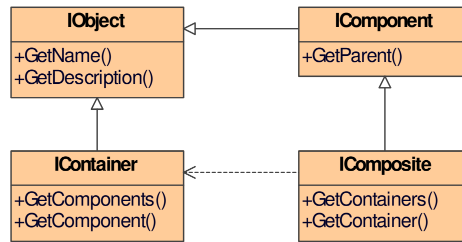

Figure 2-6: UML Class Diagram: Component containment

Via these two mechanisms (`GetParent()` and `GetContainers()/GetComponents()`), it is possible to navigate from each component in a simulation to any other component (and hence from each model instance to any other model instance). As every component has a name, this navigation works via absolute and relative names ("paths").

2.7.2 Simulation Services

Simulation services are provided by the simulation environment[3], and consumed by the models. As all mechanisms in SMP2, simulation services are based on standardised interfaces. The `IModel` interface that every model implements has a dependency on the `ISimulator` interface, and this again depends on the `IService` base interface that every simulation service has to implement.

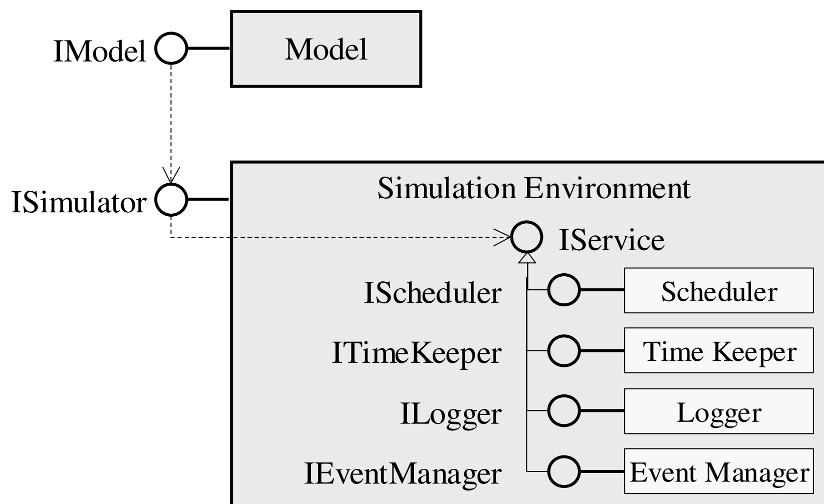

Figure 2-7: Simulation Services

2.7.2.1 Service Acquisition Mechanism

In SMP2, a configuration is a hierarchy of model instances. After these instances have been created, they need to be informed how they can acquire simulation services. This is done as follows:

- The simulation environment (which provides services) needs to implement the `ISimulator` interface.

[3] *As with every component-based design, the simulation environment does not have to **implement** the services, but has to **provide access** to them via their interfaces.*

 © EUROPEAN SPACE AGENCY, 2005

- A model (which can consume services) needs to implement the `IModel` interface, which has a `Connect()` method that gets passed a reference to the `ISimulator`.
- In the `Connecting` state (see 2.6), it is ensured that the `Connect()` method of every model is called, passing it the reference to the `ISimulator`.
- The model can now use the `ISimulator` interface and call its `GetService()` method, passing it the name of a service. This will return a reference to this service (if supported). To avoid explicit dependencies of this mechanism on specific simulation services, only the generic `IService` interface is returned, which the model may then cast to the specific service interface.
- Finally, the model can use the reference to the service and call its specific methods. The simulation environment has to guarantee that all services can be used when models get access to them (i.e. before the `Connect()` method on the model is called).
- All simulation services can be used afterwards as well.

The service acquisition mechanism is displayed in a UML sequence diagram in Figure 2-8.

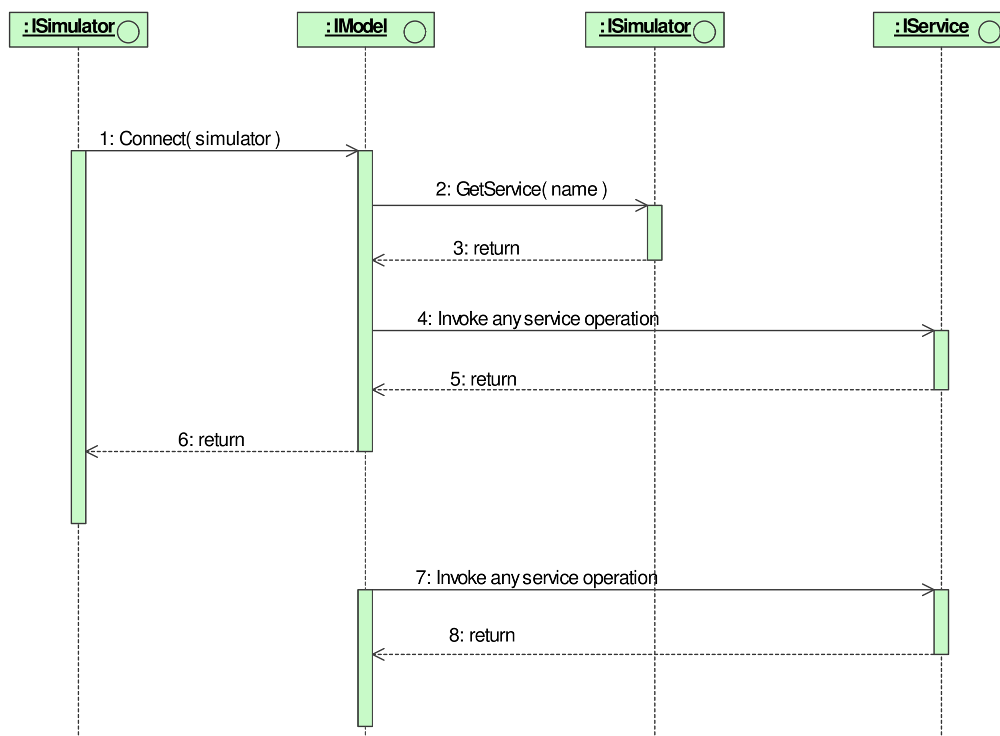

Figure 2-8: UML Sequence Diagram: Service Acquisition Mechanism

2.7.2.2 Service Categories

The service acquisition mechanism defined uses a name to identify a service. While this introduces the risk of miss-spelling a name (which can be overcome by defining constants), it provides a flexible mechanism where new services can be added easily. SMP2 defines three categories of simulation services:

1. **Mandatory services** that every simulation environment needs to provide. Every model can assume that these services do exist, and that they are implemented as specified in the standard.

2. **Optional services** that are only needed for specific types of simulations. However, if they are provided, they need to be provided as specified in the standard.

3. **User-defined services**, which are not standardised at all. However, the same service acquisition mechanism can be used for these services as well.

2.7.2.2.1 Mandatory Services

The standard specifies four services as being mandatory:

Logger: The logger allows logging information, event, warning, and error messages consistently. It is used by services as well as by models.

Scheduler: The scheduler calls entry points based on timed or cyclic events. It closely depends on the time keeper service.

Time Keeper: The time keeper service provides the four different time kinds of SMP2: A relative simulation time, an absolute epoch time, a relative mission time, and the absolute Zulu time which relates to the clock time of the computer.

Event Manager: The event manager service provides mechanisms for global events: Event handlers can be registered, events can be broadcasted, and user-specific event types can be defined as well.

Although every simulation environment shall provide these services, it is recommended that a model checks the return value of the `GetService()` method when asking for one of these services. As an alternative to the `GetService()` method, dedicated, typed methods exist for the four mandatory services, e.g. `GetLogger()`.

2.7.2.2.2 Optional Services

One additional service is standardised, but classified as optional.

Resolver: The resolver service allows resolving references to other model instances in the model hierarchy. It works with relative names as well as with absolute names.

As a simulation environment does not need to provide optional services, it is strongly recommended that a model checks the return value of the `GetService()` method when asking for an optional service.

2.7.2.2.3 User-defined Services

These services are not specified by the standard. However, the `ISimulator` interface provides a method `AddService()` which allows for registration of user-defined services. It gets passed the name of the new service and a reference to its `IService` interface. Note that every user-defined service has to implement the `IService` interface as well in order to make use of the standardised service acquisition mechanism.

For a detailed description of simulation services, see the SMP 2.0 Component Model document [AD-2].

2.7.3 Simulation Time Kinds

In SMP2, time is managed by a dedicated simulation service, called the **Time Keeper**, which provides four different kinds of time. It keeps track of simulation time and puts it into relation with epoch time and mission time. Further, it provides Zulu time based on the clock of the computer.

2.7.3.1 Simulation Time

Simulation time is a relative time. It only exists within the time keeper service. The following holds for simulation time:

1. Simulation time is a non-negative value measured in nanoseconds, which is the lowest level of granularity supported for time in SMP2.

2. Simulation time is stored in a signed 64-bit integer value. This allows specifying time values of more than 290 years.

 © EUROPEAN SPACE AGENCY, 2005

3. Simulation time can be queried using the `GetSimulationTime()` method of the time keeper (via the `ITimeKeeper` interface).

4. Simulation time is initialised to 0 at the beginning of the initialisation phase. That is, during initialisation the time keeper service will return a simulation time of 0.

5. Simulation time is only progressed when the simulation environment is in `Executing` state.

6. When storing a state vector, simulation time is stored as well.

7. When restoring a state vector, simulation time is restored as well.

The standard does not define how quickly simulation time is progressed when the simulator is in `Executing` state. Typical examples are:

a) **Real-Time**: The simulation time progresses with real-time, where real-time is typically defined by the computer clock. Note that two types of real-time simulations exist: hard real-time and soft real-time simulations. In a hard real-time simulation, strict requirements on timing have to be met, for example to keep hardware in the loop devices “alive”. In a soft real-time simulation, the requirements are less demanding such that latencies in a certain range are allowed, which is called real-time slip.

b) **Accelerated**: The simulation time progresses relative to real-time using a constant acceleration factor. This factor may be larger than 1.0, which relates to “faster than real-time”, smaller than 1.0, which means “slower than real-time”, or 1.0, which coincides with real-time.

c) **Free Running**: The simulation time progresses as fast as possible, and is not related to real-time. Typically, the speed is coordinated with the timed events of the scheduler, which underlines the close relationship between these two services (Time Keeper and Scheduler).

d) **Debugging**: The simulation is executed in a step-by-step manner using break points in order to inspect data or trace calls within the simulation.

SMP2 does not mandate which of these modes a simulation environment has to support.

2.7.3.2 Epoch Time

Epoch time is an absolute time, i.e. it defines a definite point in time. It is not only used as a way to express date and time, but also to determine all time-dependent variables at that time, such as barycentric positions of all solar system bodies. Epoch time is stored as a number relative to a reference date, which has been defined as the 1st of January 2000 mid-day (01.01.2000, 12:00). Epoch time is maintained using a fixed offset to simulation time, and hence progresses together with simulation time, except for when the offset is changed. The following holds for epoch time:

1. Epoch time is a value measured in nanoseconds, which is the lowest level of granularity supported for time in SMP2.

2. Epoch time is returned as a signed 64-bit integer value, relative to the epoch reference time (01.01.2000, 12:00, Modified Julian Date (**MJD**) 2000+0.5). This allows specifying time values roughly between 1710 and 2290.

3. Epoch time can be queried using the `GetEpochTime()` method of the time keeper (via the `ITimeKeeper` interface).

4. Epoch time is initialised to 0 (i.e. 01.01.2000, 12:00) at the beginning of the initialisation phase, but can be changed already before entering the execution phase.

5. Epoch time is progressed linearly with simulation time (i.e. with a fixed offset to simulation time). Using the `SetEpochTime()` method of the time keeper (via the `ITimeKeeper` interface), the offset between simulation time and epoch time can be changed.

6. When storing a state vector, epoch time (i.e. its offset to simulation time) is stored as well.

7. When restoring a state vector, epoch time (i.e. its offset to simulation time) is restored as well.

2.7.3.3 Mission Time

Mission time is a relative time, i.e. it measures elapsed time from a definite point in time (called the mission start). Mission time is stored as a number relative to the mission start date. Mission time is maintained using a fixed offset to epoch time, and hence progresses together with simulation and epoch time, except for the case when the offset is changed. The following holds for mission time:

1. Mission time is a value measured in nanoseconds, which is the lowest level of granularity supported for time in SMP2.
2. Mission time is returned as a signed 64-bit integer value, relative to a mission start date and time (which itself is not stored).
3. Mission time can be queried using the `GetMissionTime()` method of the time keeper (via the `ITimeKeeper` interface).
4. Mission time is initialised to 0 at the beginning of the initialisation phase, but can be changed already before entering the execution phase. As epoch time is initialised to 01.01.2000, 12:00, the default mission start is 01.01.2000, 12:00 as well.
5. Mission time is progressed linearly with epoch and simulation time (i.e. with a fixed offset to epoch time). Using either the `SetMissionTime()` method or the `SetMissionStart()` method of the time keeper (via the `ITimeKeeper` interface), the offset between simulation time and mission time can be changed. The following holds for mission time and mission start:

   `MissionStart + MissionTime = EpochTime`

6. When storing a state vector, mission time is stored as well.
7. When restoring a state vector, mission time is restored as well.

2.7.3.4 Zulu Time

From the Mobile Aeronautics Education Laboratory (**MAEL**) of the NASA, the following definition of Zulu Time is cited (http://www.grc.nasa.gov/WWW/MAEL/ag/zulu.htm):

> “The world is divided into 24 time zones. For easy reference in communications, a letter of the alphabet has been assigned to each time zone. The "clock" at Greenwich, England is used as the standard clock for international reference of time in communications, military, maritime and other activities that cross time zones. The letter designator for this clock is **Z**.
>
> Times are usually written in military time or 24 hour format such as 1830**Z** (6:30 pm). To pronounce this, the phonetic alphabet is used for the letter Z, or Zulu. This time is sometimes referred to as Zulu Time because of its assigned letter. Its official name is Coordinated Universal Time or **UTC**. Previously it had been known as Greenwich Mean Time or **GMT** but this has been replaced with UTC.”

In SMP2, Zulu time is not related to simulation time, but typically to the computer clock (or to some external clock). The following holds for Zulu time:

1. Zulu time is measured in nanoseconds.

 © EUROPEAN SPACE AGENCY, 2005

2. Zulu time is returned as a signed 64-bit integer value, relative to the epoch reference time (see Epoch Time above).

3. Zulu time represents the current time at Greenwich, England, called UTC or GMT.

As Zulu time is not managed by the time keeper service, but provided based on an external clock (typically the computer clock), it is not related to simulation time, and progresses independently of the state of the simulation environment.

When a simulator interfaces to an external system, e.g. a ground station or some Hardware-In-The-Loop (**HITL**), Zulu time is often used as a time stamp.

2.7.4 Model Entry Points

Both the Scheduler Service and the Event Manager Service call operations of the models. As scheduler and event manager neither pass parameters to the operations, nor evaluate any return values, they can only call operations that do not take parameters and do not return a value (so called `void`-`void` operations). Such an operation is called an **Entry Point**.

As SMP2 models are classes, their operations are not simply functions, but member functions of instances of this class. This is a notable difference since member functions can only be called together with an instance pointer (in C++, this is the "`this`" pointer that can be used within the member function). This distinction is important, because an SMP2 simulation may contain several instances of the same model, and each instance has its own entry points.

For this reason, SMP2 cannot use function pointers to refer to entry points in the way they have been used in SMP1, i.e. as memory addresses. Another limitation of function pointers is that they only work within a single address space. While this allows for multi-threading or shared memory architectures, it typically neither works for multi-process applications, nor for distributed simulations. For the same reasons, the Common Object Request Broker Architecture (**CORBA**) does not even allow defining function pointers, or passing them around.

Concluding, SMP2 takes an interface-based approach to refer to entry points. An entry point can only be called via the `IEntryPoint` interface, which has a single method called `Execute()`. Interfaces are supported by all target platforms (including C++, CORBA, J2EE, COM, .NET), and may even allow distribution (when using a middleware). As the `IEntryPoint` interface is so simple, it can be implemented with a few lines of code (see example in Figure 3-7 on page 54). In a C++ implementation, an appropriate template class can reduce the definition of an entry point to a single line of code.

2.7.5 Model Publication

In SMP1, a model publishes its services (called "operations" in SMP2) and data items (called "fields" in SMP2) to the SMI. A Model Manager is used "to control the interactions between models" [RD-1, *p. 21*], but "any of the system components can get the data details from the SMI […]. This bypasses the access mode control that the SMI services implement" [RD-1, *p. 25*].

In SMP1, publication of information is therefore used for two purposes:

1. To tell the simulation environment about available data items and services, e.g. to enable it to show this information, make it available for visualisation (data) or scripting (services), or to store and restore simulation state vectors.

2. To allow models using this information to establish connections between models, i.e. to read and write data from other models, and to call services of other models.

While the first purpose has been widely accepted and used in existing SMP1 projects, inter-model communication was often not performed via the SMI by simply integrating all models into one global

SMP1 model. Therefore, communication between the individual sub-models was invisible to the SMI, and not based on the information published to it. Further, by-passing the SMI mechanisms for inter-model communication means that models get full read/write access to all published fields of other models, even to “private” fields that have been published only because they are part of the state vector.

While SMP2 still provides publication of fields and operations, it mainly aims at using this information for the first purpose. For inter-model communication, SMP2 provides other mechanisms based on modern software design and engineering principles. These allow models to control how other models can access their internals. For a detailed description of the SMP2 publication mechanism, see the SMP 2.0 C++ Mapping [AD-3] document.

2.7.6 Model Interactions

SMP2 promotes a component-based architecture, where each model is a component on its own, and communicates with other models via dedicated mechanisms. These are introduced in this section.

Depending on the platform used, native mechanisms may exist in addition. Taking the C++ platform as an example, each model is represented by a `class`. A class can access public members of other classes and even protected or private members if it has been declared as a `friend`. However, this is only possible if classes know about each other’s implementation, which creates dependencies between classes. This is still a class-based design, but not a component-based design.

The most important concept for inter-model communication, which is a core concept of SMP2, is that of an interface giving access to operations. In addition, properties can be used to give controlled access to data (typically fields). For event-based programming, SMP2 provides optional mechanisms for event sources and event sinks. Finally, models participating in a dataflow simulation have to give read access to their output fields, and write access to their input fields.

2.7.6.1 Interfaces

Interface-based design is a programming discipline that is based on the separation of the public interface from its implementation. Software engineers who discovered that using distinct interfaces could make their software, especially large applications, easier to maintain and extend, pioneered it in languages such as C++ and Smalltalk. The creators of Java saw the elegance of interface-based programming and consequently built support for it directly into their language, and interfaces are a fundamental concept of the Microsoft .NET platform and especially the C# programming language.

Existing middleware solutions, especially the Common Object Request Broker Architecture (**CORBA**) or the Component Object Model (**COM**), make heavy use of interfaces. These are typically defined in an Interface Definition Language (**IDL**), and the implementation is left unspecified.

An **Interface** declares a set of public features, typically operations and properties. These features are provided by a model (a **Provider**), and exposed to the outside via a provided interface. The interface thereby does not contain any information about the actual implementation of these features. However, if a model provides an interface it is guaranteed that the model supports all features declared in the interface.

Another model (a **Consumer**) may consume features of the interface, i.e. it requires a provider with an actual implementation. The consumer can make use of the necessary features through the provided interface, without direct knowledge of or dependency on the implementation of the provider.

© EUROPEAN SPACE AGENCY, 2005

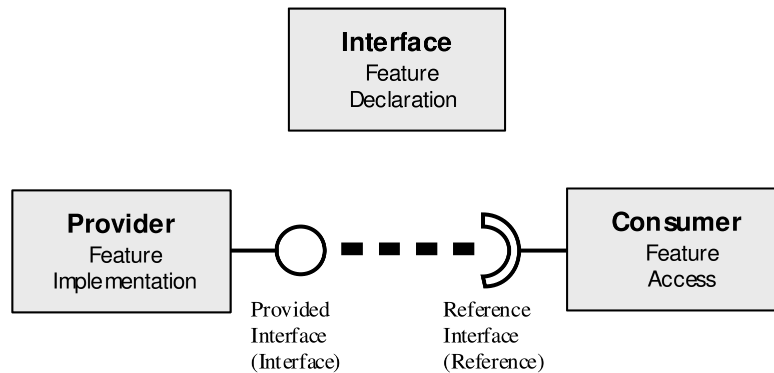

Figure 2-9: Interface-based Design

2.7.6.1.1 Properties

Typically, interfaces may only contain operations, but no fields. This is due to the fact that access to fields would mean direct access to certain memory locations, which is not supported by most middleware. However, interfaces need to support transfer of data. Access to a field can be provided by a set of two operations: the **getter** operation for reading the value, and the **setter** operation for writing the value. Such a pair of operations is called a **Property**.

While this is a direct translation of a field into operations, properties are much more powerful. They allow restricting access to read-only or write-only simple by providing just one of the two possible operations. Further, the model is aware of every access to its data, as it gets explicitly called for reading and writing (via the getter and setter operations). Finally, a property does not even need to represent an internal field of a model: A system may have a read-only property `Mass` that does not return the value of a field, but may sum up and return the masses of all subsystems.

Summarising, properties are an example of encapsulation and promote reuse of models.

2.7.6.2 Events

Event-based design allows spontaneous signal propagation between models[4]. A model holding an event source (a **Provider**) can be connected to one or more models holding an event sink (**Consumers**) of the same event type. Whenever the event is triggered in the provider, the corresponding event handlers of all consumers are invoked. In order to distinguish between different kinds of events, an event is assigned an event type. This type defines which data are passed to the event handlers on invoke (i.e. it defines their signature).

[4] *Note that SMP2 does not only support inter-model events, which are described here, but we well global events via an Event Manager service.*

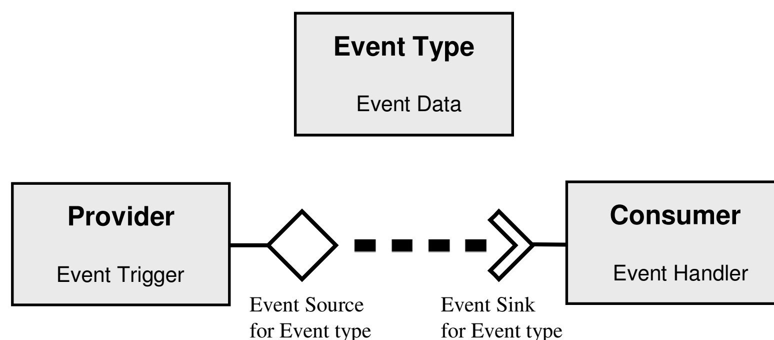

Figure 2-10: Event-based Design

2.7.6.3 Dataflow

In dataflow-based architectures, data is transferred between model instances based on links between output fields and input fields. While this is another mechanism of inter-model communication, it differs from the other concepts introduced in this section, as model instances do not actively interact with others: Data transfer is initiated by some other component(s), which reads a value from an output field of the **Source** model instance and stores it into the input field of the **Target** model instance (see Figure 2-11).

Software based on dataflow architecture consists of a collection of independent components running in parallel that communicate via data links or data channels. Such a design can be succinctly depicted graphically. A node is a computational component, and an arrow is a data channel. A control algorithm is divided into nodes first. Each concurrently executing node is a self-contained software part with well-defined functionality. Data channels provide the sole mechanism by which nodes can interact and communicate with each other, ensuring lower coupling and greater reusability. Data channels can also be implemented transparently between processors to carry messages between components that are physically distributed.

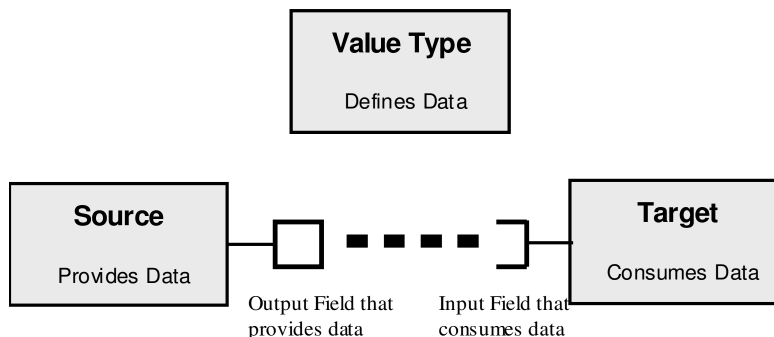

Figure 2-11: Dataflow-based Design

Choosing this component model for embedded control software alleviates many of the negative aspects of the more traditional main program- and-subroutine organization. More importantly, however, it also opens up the possibility of developing a library of commonly recurring, standardized control software functions encapsulated in reusable data flow components.

2.7.7 Managed Models

Before a simulation can enter the execution phase (see 2.6), several steps have to be performed in the setup phase. These include:

- Model run-time instances have to be created from existing model implementations

 © EUROPEAN SPACE AGENCY, 2005

- Model run-time instances have to be linked together
  a. Interface links connect required and provided interfaces together
  b. Event links connect event sinks and event sources together
  c. Field links connect output fields and input fields together
- Model run-time instances have to be initialised to their initial (field) values
- Scheduling of entry points has to be defined

All these steps can be done from source code, and some of it can be done from the source code of the models (e.g. a model may register its entry points with the scheduler). This results in a **statically configured** simulation (static configuration), where changes of the model hierarchy, of initial values[5], of model links, or of model scheduling require changing some source code. While this is a completely valid approach for SMP2, and well applicable to projects where the architecture is well known and fixed (e.g. operational simulators), it is not very appropriate for other types of simulations like those used for design and validation (design support simulators). In such a simulation, it is for example required

- to quickly assemble a simulation using generic base models
- to replace a model with another model in order to add some specific behaviour
- to add/remove a model to/from the hierarchy to react on design changes
- to run different simulations with different initial values for trade-offs
- to switch between a software and a hardware model

The Generic Project Test Bed (**GPTB**) is an example of a model library supporting such type of simulations, and the Concurrent Design Facility (**CDF**) at ESTEC uses simulation in a similar way.

As an answer to the growing demand for **dynamically configured** simulations (dynamic configuration), SMP2 defines three mechanisms to support such a process:

- A **Catalogue** defines a model library with additional metadata for each model
- An **Assembly** defines a hierarchy of model instances with initial values and links
- A **Schedule** defines how the entry points of the model instances shall be scheduled

Catalogue, Assembly and Schedule are defined as XML file formats (based on XML Schema definitions). XML is a platform independent, text based format with powerful import and export mechanisms, for example via transformations described in the XML Stylesheet Language (XSLT).

When creating a simulation based on XML files, the following steps have to be performed:

1. An Assembly XML document has to be loaded and processed:
   a. For each **Model Instance** defined within the assembly, an instance of the corresponding model Implementation has to be created.
   b. For each model instance, **Name**, **Description** and **Parent** have to be set as specified in the assembly.
   c. For each **Field Value** specified for a model instance, the default value of the model instance has to be changed for the specified value.
   d. For each **Interface Link** specified for a model instance, a connection to the interface provider has to be established.

---

[5] *Clearly, a statically configured simulator may choose to read the initial values from some external file or database to allow for changing initial values without the need for recompilation. However, this is not covered by SMP2 and specific to the individual simulator. The standardised SMP2 mechanisms are described in the remainder of this section.*

e. For each **Event Link** specified for a model instance, a connection between the event consumer and the event provider has to be established.

2. A Schedule XML document has to be loaded

   a. Each **Task** of the schedule has to be exposed to the scheduler

   b. Each **Timed Event** of the schedule has to be registered with the scheduler

Typically, all these steps are performed by an external component that connects to the simulation environment via its `IDynamicSimulator` interface. As this component manages the load and initialisation process, it is called a **Model Manager**.

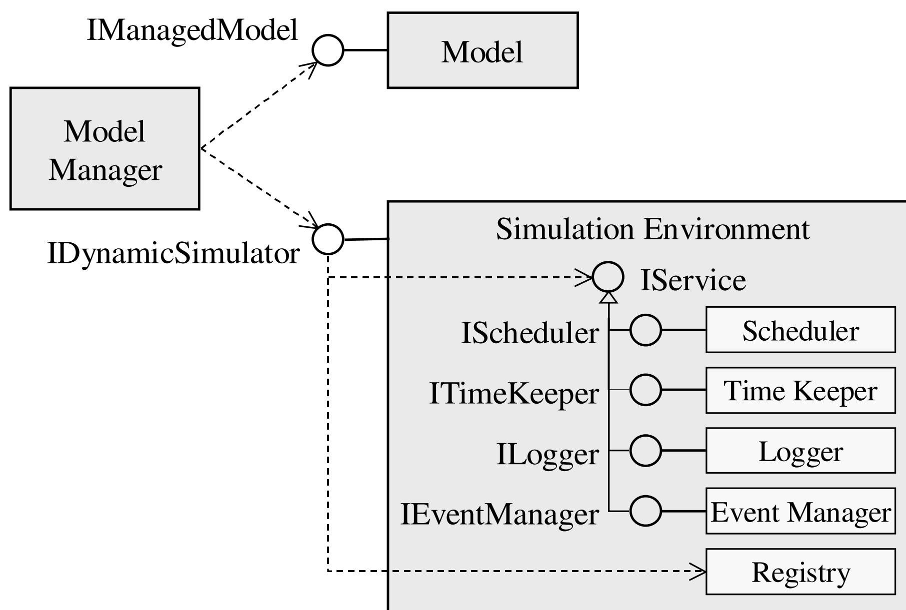

Figure 2-12: Typical Architecture using a Model Manager

While the existence of a Model Manager component is not required by the standard, the standard recommends using a Model Manager for dynamically configured simulations. Figure 2-12 shows a typical architecture for the use of a Model Manager.

The Model Manager needs to get access to the models in order to initialise them and to the simulator in order to pass it to the models and in order to add (root) models to the simulation. As XML is an ASCII based file format, all these steps have to be performed on a "**call-by-name**" basis, i.e. the strings found in the XML documents have to be used to pass information to the models, or to acquire the required information from them. This requires that all models involved in a dynamic configuration have to implement additional interfaces.

For a detailed description of Catalogue, Assembly, Schedule, Model Instance, Interface Link, Event Link, Field Link, Task, Timed Event see the SMP 2.0 Metamodel [AD-1].

For a detailed description of Name, Description, Parent, of mandatory interfaces a model has to provide, and of optional interfaces a managed model may provide see the SMP 2.0 Component Model [AD-2].

2.8 Model Development Guidelines

This section provides the complete set of SMP2 model development guidelines. For each guideline a description, rationale and compliance test are provided.

This section does not discuss the portability aspects of individual languages, as it is assumed that the model developers will be sufficiently experienced in their own language to be aware of language specific portability issues.

 © EUROPEAN SPACE AGENCY, 2005

A set of general guidelines is presented here, that provide a general checklist of coding practices and compliance tests, aimed at maximising code portability and re-use. The guidelines are presented in the following subsections.

2.8.1 Provide Documentation

Good documentation is one of the most important factors for software reuse as it improves the ease of understanding it. Software that is easy to understand is much more likely to be reused than software that is incomprehensible. For this reason we recommend that each model is accompanied by an SMDL Catalogue file and additional documentation similar to the template specified in SMP1 [RD-1]. It is recognised that the required documentation highly depends on the deployment process adopted within the organisation, and that the template needs to be adapted to SMP2. It may become part of a future release of this standard.

Guideline: Provide SMDL Catalogue file and complete the document template.

Rationale: Good documentation increases ease of understanding and correspondingly the re-use potential.

Compliance Test: Inspect catalogue and documentation.

2.8.2 Use Standard Languages

SMP2 is a platform independent standard. However, the ANSI/ISO C++ mapping (see [AD-3]) is currently the only platform mapping that is part of the standard. Therefore, ANSI/ISO C++ can be seen as today's SMP2 language integration platform. As SMP2 models are typically developed by different organizations, it is unrealistic to require every model to be implemented in ANSI/ISO C++. However, every SMP2 compliant model implementation must have an ANSI/ISO C++ interface. Models written in other languages can be integrated into an SMP2 environment by providing an appropriate ANSI/ISO C++ wrapper, where certain rules have to be followed.

Guideline: SMP2 models must have an ANSI/ISO C++ interface. When using other languages for the implementation of model functionality, one of the following recommended standard languages should be used, provided that binary integration with ANSI/ISO C++ is possible via an appropriate ANSI/ISO C++ wrapper:

- ANSI/ISO C++ (required for the interface)
- ANSI C
- Fortran 77/90/95/2000
- Ada 83/95

Rationale: Using one of these standard languages maximises the chances of compiler availability and reuse on another platform.

Compliance Test: Compilation with one of the standard compilers and integrating into an ANSI/ISO C++ wrapper using an ANSI/ISO conformant C++ compiler on the target platform.

2.8.3 Avoid Compiler Specific Language Extensions

The restriction to a set of standard languages is not sufficient to ensure that re-compilation across platforms is achievable. Different compilers often have language extensions to "improve" the language. While the features are often useful, they are usually compiler specific and hence reduce portability of the code.

Guideline: SMP2 models should use the particular language's standard language features only.

Rationale: Restricting the code to standard language features only maximises the chances of recompilation in different compilers.

Compliance Test: Compile with compiler option to flag non-standard language features as errors.

2.8.4 Use Standard Libraries Only

Restricting code development to standard language features maximises the possibility of re-compilation with different compilers. However any realistic code development will need to make use of standard libraries to perform essential tasks.

Guideline: SMP2 models should only use the libraries listed in the following table

Language	Standard Libraries
ANSI/ISO C++	Standard C Library Functions Standard Template Library (STL)
ANSI C	Standard C Library Functions
Fortran 77/90/95/2000	Standard Fortran Library Functions
Ada 83/95	Library packages recommended in LRM

Rationale: Use of standard language libraries should ensure their availability on each platform.

Compliance Test: Search for included library files in source code and verify by inspection that only standard libraries are referenced. Alternatively, try and link the model into a system that is known to have only standard libraries.

2.8.5 Limit Use of Essential Non-standard Libraries

Sometimes it is completely impractical to develop a model without using a library that is not one of the standard libraries. Where a non-standard library is used, the library should itself be equally portable or widely available so that the model’s portability is not compromised by its use.

Guideline: Non-standard libraries should only be used if they are themselves portable. The library should either be provided with the model, or the dependence clearly documented.

Rationale: The model can use non-standard libraries and still maintain portability if the libraries themselves are portable.

Compliance Test: Investigate library availability on different platforms or verify its portability to different platforms.

2.8.6 Use the SMP2 Interfaces to interact with the Simulation Environment

The models need to interact with the simulation environment to receive the support services that it provides. SMP2 provides standardised interfaces for the model to use, so that it may interface to the environment in a fixed way. The SMP2 interfaces (component model and simulation services) act as an adapter translating the standardised environment actions to native environment specific ones.

Guideline: All interactions with the environment should be through SMP2 standard interfaces.

Rationale: By using only SMP2 standard interfaces the model will be more portable.

Compliance Test: Search for used interfaces/services and verify that they are SMP2 interfaces/services.

 © EUROPEAN SPACE AGENCY, 2005

2.8.7 Avoid Direct Input / Output Operations

Unfortunately even with standard languages, input / output operations remain prone to portability problems. For example, file-opening operations allow operating system specific modes of file operation even with standardised languages.

Because of the difficulty in producing code that deals with input / output that is reliably portable, input / output should be avoided, though in practice, it is recognised that this may not always be achievable.

In cases where avoidance is not possible, file input should be done external to the model, and SMP2 mechanisms shall be used to get data from files into the models. Output should be through setting published field values or using the logger service only.

Guideline: Avoid input / output operations.

Rationale: Removing all input and output operations from the model to prevent potential portability problems.

Compliance Test: Search for file-access commands (such as open) in the source code.

2.8.8 Do Not Rely on Internal Representation of Data

Even if standard language and standard library features are being used, it is still possible to produce code that is not portable. One of the main causes of this is the reliance on the representation in memory of data items.

A common example of this type of reliance is mapping different variables to the same location, in order to perform conversions. In addition, the following elements are dangerous in C/C++:

1. C/C++ bitfields
2. C/C++ structures: In particular these may contain padding between components.
3. C/C++ enums: These can have rather variable sizes
4. C/C++ unions: In particular it is often thought that it is safe to store a value into one component of a union and then to read it using a different component. The C standard is quite explicit that the result of this is undefined.

Guideline: Do not write code that relies on the representation of data in memory.

Rationale: The representation of data in memory is compiler specific and cannot be guaranteed to be the same across platforms.

Compliance Test: General inspection. Check for language constructs that allow the mapping of variables to the same location.

2.8.9 Avoid Global Data Declarations

When integrating models together, name clashes will often occur if models declare data names that are visible in the global namespace. Because it is quite likely that two models could have data items with the same name, the model data should be kept out of the global data namespace.

Guideline: Avoid putting data names into the global namespace.

Rationale: Global data names lead to name clashes, which reduces the re-use potential of the model.

Compliance Test: Search for global data declarations and list all global data declarations.

2.8.10 Avoid Common Global Names

With some languages it is necessary to place the model operation names in the global namespace, as otherwise they would not be visible at all. When naming items that must be placed in the global namespace they should be provided with a unique prefix that identifies that the service is related to the model.

Guideline: Give all global names associated with the model the same prefix.

Rationale: The prefix should maximise the chances of the names being unique, reducing the chances of name clashes and hence, maximising re-use potential.

Compliance Test: Search for all global name declarations and verify they all have the same prefix.

2.8.11 Enable Multiple Instances

Quite often a model user may want several instances of the same model in their simulation. It may be that a model is originally developed in a system where only a single instance is required. It is then quite possible that a future user could not have multiple instances of that model in their simulation.

Guideline: If it is likely that multiple instances are going to be required in a future system, then design the model so that multiple instances of the model may exist in the final system.

Rationale: The re-use potential for a model that is likely to be used with multiple instances is improved by designing it so that multiple instances are possible.

Compliance Test: Incorporate multiple instances of model in test harness. Run model tests on several instances and verify they behave correctly and independently.

2.8.12 Minimise Number of Model Interfaces

Minimising the number of interfaces to the model improves portability and re-use potential by making the model simpler to understand and use.

Guideline: Design the model so that the developer needs only a few interfaces to use the model.

Rationale: Fewer interfaces simplify the protocol of model use, which improves re-use potential by making the model understandable.

Compliance Test: Count the services to the model.

2.8.13 Simplify Model Interfaces Provided

Minimising the number of interfaces will not improve the ease with which the model can be used if the interfaces are very complex. As well as trying to reduce the number of interfaces, the complexity of each interface should be kept as simple as possible.

Guideline: Design model interfaces to be as simple as possible.

Rationale: Simpler interfaces simplify the protocol of model use, which improves re-use potential by making the model understandable.

Compliance Test: Get the interface reviewed by another developer.

 © EUROPEAN SPACE AGENCY, 2005

2.8.14 Only Select Suitable Candidates for Reuse

Experience producing generic models has shown that what constitutes re-usable models is not always immediately obvious. For example AOCS subsystems exist on every spacecraft in some form or other, and so one might expect that an AOCS subsystem is a good candidate for a generic model.

After some investigation it was found that producing a general overall generic model was unfeasible as the numerous variety of control law sensors and actuators made it unfeasible to provide a black box generic AOCS model. Instead it was decided a library of generic AOCS components should be produced, as it seemed that these could be reasonably useful. The wide diversity of individual AOCS components meant each one was provided with numerous tailoring features and functionality that could be adjusted to be able to make the models represent all forms of the particular AOCS component.

After providing the generic AOCS models to the first users it was found that due to the complexity of tailoring capabilities, the users had to perform more work to tailor the generic AOCS components to perform the specific task required. The AOCS generic model showed that there is a point at which the reuse requirements make the model so generic that its complexity of use outweighed the reuse benefits.

Guideline: Only make models reusable if they will be effective to reuse.

Rationale: It can actually be *more* difficult to reuse a model when it needs more tailoring for specific use than is required to develop the functionality from scratch.

Compliance Test: None.

© EUROPEAN SPACE AGENCY, 2005

2.9 SMP2 Model and Simulation Environment Features

As SMP2 models and simulation environments can be used in two different ways (unmanaged models with static integration, or managed models with dynamic integration), the interfaces a model or simulation environment has to implement depend on the intended usage. This section explains which interfaces a model or a simulation environment has to implement in order to support statically or dynamically configured simulations.

2.9.1 Statically Configured Simulations

These are the requirements on components to be used in statically configured simulations.

2.9.1.1 Model Interfaces

These are the requirements on models to be used in statically configured simulations.

Table 2-1: Required Model Interfaces for Statically Configured Simulations

Feature	Interfaces required
SMP2 Model	`IModel`
Model with event sources	`IEventSource` for each event source
Model with event sinks	`IEventSink` for each event sink
Model with composition (children)	`IComposite` for the model `IContainer` for each container
Model with aggregation (references)	`IAggregate` for the model `IReference` for each reference
Model with entry points	`IEntryPoint` for each entry point

2.9.1.2 Simulation Environment Features

These are the requirements on a simulator to be used in statically configured simulations.

Table 2-2: Required Simulation Environment Features for Statically Configured Simulations

Description	Features required
SMP2 Simulation Environment	Interface `ISimulator` Managed container named “`Models`” Managed container named “`Services`” Interface `IManagedContainer` for each container Component implementing `IPublication` `Logger` service `TimeKeeper` service `Scheduler` service `EventManager` service

 © EUROPEAN SPACE AGENCY, 2005

2.9.2 Dynamically Configured Simulations

These are the requirements on components to be used in dynamically configured simulations.

2.9.2.1 Model Interfaces

These are the requirements on models to be used in dynamically configured simulations.

Table 2-3: Required Model Interfaces for Dynamically Configured Simulations

Feature	Interfaces required
SMP2 Model	`IManagedModel`
Model with event sources	`IEventProvider` `IEventSource` for each event source
Model with event sinks	`IEventConsumer` `IEventSink` for each event sink
Model with composition (children)	`IComposite` for the model `IManagedContainer` for each container
Model with aggregation (references)	`IAggregate` for the model `IManagedReference` for each reference
Model with entry points	`IEntryPointPublisher` `IEntryPoint` for each entry point

2.9.2.2 Simulator Features

These are the requirements on a simulator to be used in dynamically configured simulations.

Table 2-4: Required Simulation Environment Features for Dynamically Configured Simulations

Description	Features required
SMP2 Simulation Environment	Interface `IDynamicSimulator` Managed container named "`Models`" Managed container named "`Services`" Interface `IManagedContainer` for each container Component implementing `IPublication` `Logger` service `TimeKeeper` service `Scheduler` service `EventManager` service `Resolver` service

2.9.3 Optional Features

These features are optional.

2.9.3.1 Optional Model Interfaces

These are the optional interfaces for models.

Table 2-5: Optional Model Interfaces

Feature	Interfaces required
Self-Persistence	`IPersist`
Dynamic Invocation	`IDynamicInvocation`

2.9.3.2 Optional Simulation Environment Features

These are the optional features for a simulation environment.

Table 2-6: Optional Simulation Environment Features

Description	Features required
Self-Persistence of Components	Component implementing `IStorageReader` Component implementing `IStorageWriter`

© EUROPEAN SPACE AGENCY, 2005

3. GETTING STARTED

This section starts with a sample class that is turned into an SMP2 model, and then step-by-step enhanced to make use of SMP2 services. This section especially addresses the Model Developer.

A **warning** from the start**:** We do *not* foresee that a model developer actually implements all the C++ code shown in the examples, but rather that most of the code is actually generated from information in the SMDL Catalogue and that the implementation is eased by a Model Development Kit (MDK); for details see Appendix B: Notes on Supporting Tools. Typically, a model developer will only have to fill in the implementation code of all model properties and operations (i.e. the model functionality), without having to care about all "boilerplate" code that accomplishes the binding to the simulation environment (most notably publication) and to other models (most notably references and containers). Where applicable, we put footnotes to explain which parts could be eased by an MDK.

In a first step, only the mandatory changes are performed, i.e. only mandatory interfaces are implemented. In following sections, the model is further enhanced to make use of most of the mandatory SMP2 services, including the Logger, the Scheduler, and the Event Manager. Finally, the model is turned into a managed model. For clarity, error checking and exception handling is omitted from the code.

3.1 An existing class

The example introduced here is a class called `Counter`. It provides:

- a protected field called `counter` which holds the current value of the counter
- a public, parameter-less method `Count` to increase the counter
- a public, parameter-less method `Reset` that resets the counter
- a public method `get_Counter` that returns the value of the counter
- a public constructor that initialises the private field `counter` to 0

The definition of the class is shown in Figure 3-1, the implementation in Figure 3-2.

```
#include "Smp/SimpleTypes.h"

// Definition of CounterClass
class CounterClass
{
protected:
        Smp::Int64 counter;    // Protected field to store counter

public:
        // Public methods to count, to reset and to return the counter
        virtual void Count(void);
        virtual void Reset(void);
        virtual Smp::Int64 get_Counter(void);

        // Public constructor.
        CounterClass() : counter(0) {}
};
```

Figure 3-1: `CounterClass.h`: Definition of Counter class

```
// Include header files
#include "CounterClass.h"

void CounterClass::Count(void)
{
        counter++;
}

void CounterClass::Reset(void)
{
        counter = 0;
}

Smp::Int64 CounterClass::get_Counter(void)
{
        return counter;
}
```

Figure 3-2: `CounterClass.cpp`: Implementation of Counter class

3.2 How to turn it into an SMP2 Model

Some lines of code have to be added to the class to make it an SMP2 model. This section shows how to manually turn the class into an SMP2 model[6].

An SMP2 Model has to provide the following, which is defined as the `IModel` interface[7]:

- a method `GetName()` which returns the name of the model instance
- a method `GetDescription()` which returns a description for the model
- a method `GetParent()` which returns the parent of the model in a model hierarchy
- a method `GetState()` which returns the current state of the model
- a method `Publish()` which asks the model to publish its fields
- a method `Configure()` which allows the model to perform custom configuration
- a method `Connect()` which connects the model to the simulator and allows to subscribe to services or other models' connection points

Each of these methods can be implemented in a single line in the definition[8], as shown in Figure 3-3. The implementation can be left unchanged.

---

[6] *Note that in practice it would be easier to implement an SMP2 model by using a code generator and a supporting C++ MDK (see Appendix B: Notes on Supporting Tools). However, this section relies on the platform mapping only and therefore shows the complete code – though parts could be generated or hidden in the MDK.*

[7] *Strictly speaking, this is defined via the three interfaces `IObject`, `IComponent` and `IModel`, where `IModel` inherits from `IComponent`, which again inherits from `IObject`.*

[8] *Note that this is not the recommended implementation, but the simplest implementation of `IModel`. In this section, you will learn how to properly implement `Publish()` and `Connect()`.*

 © EUROPEAN SPACE AGENCY, 2005

```
#include "CounterClass.h"
#include "Smp/IModel.h"
#include "Smp/IEntryPoint.h"
#include "Smp/Services/ILogger.h"
#include "Smp/Services/IScheduler.h"
#include "Smp/Services/IEventManager.h"

/// This class turns the CounterClass into an Smp model.
class Counter : public CounterClass, public virtual Smp::IModel
 {
private:
    char* name;                                ///< Name of model.
    Smp::ModelStateKind state;                 ///< Model state.
    Smp::IComposite *parent;                   ///< Parent component.
    void Init();                               ///< Init private fields.
public:
    /// Public constructor with name and parent.
    Counter(Smp::String8 name, Smp::IComposite *parent) : CounterClass()
    {
        Init();
        this->name   = strdup(name);
        this->state  = Smp::MSK_Created;
        this->parent = parent;
    }
    /// Virtual destructor is always recommended.
    virtual ~Counter()
    {
        if (name) free(name);
    }
    virtual void Reset(void);                  ///< Reset method logs a message

    // IModel methods
    virtual Smp::String8 GetName()          const { return name; }
    virtual Smp::String8 GetDescription()   const { return "Counter Model"; }
    virtual Smp::IComposite* GetParent()    const { return parent; }
    virtual Smp::ModelStateKind GetState() const { return state; }
    virtual void Publish(Smp::IPublication* receiver)
           throw (Smp::IModel::InvalidModelState)
    {
        state = Smp::MSK_Publishing;
    }
    virtual void Configure(Smp::Services::ILogger* logger)
           throw (Smp::IModel::InvalidModelState)
    {
        state = Smp::MSK_Configured;
    }
    virtual void Connect(Smp::ISimulator* simulator)
           throw (Smp::IModel::InvalidModelState)
    {
        state = Smp::MSK_Connected;
    }
};
```

Figure 3-3: `Counter.h`: Definition of model Counter

Note that the model's constructor needs to be given the name and parent of the new instance, as the model provides read-only access to these two fields only. This allows having different instances of the same model with different names, and in different places in the model hierarchy. In a proper implementation, the state transition methods `Publish()`, `Configure()` and `Connect()` should check the model state, and throw an exception if called in an invalid state – as shown later in this section.

For detailed information about the methods of the `IModel` interface, see section 3.2.2.4 of the SMP 2.0 Component Model [AD-2].

3.3 How to publish data to the Environment

Although the example class has been turned into a valid SMP2 model, it does not publish its `counter` field to the simulation environment. Therefore, the environment can neither make this field available for external applications (e.g. for visualisation purposes), nor can it store and restore its state.

The sample model already implements the `Publish()` method, which gets passed a reference to the `IPublication` interface. Via this interface, it can publish its private field, as shown in Figure 3-4.

```
// Publish fields to environment
void Counter::Publish(Smp::IPublication *receiver) throw
(Smp::IModel::InvalidModelState)
{
    if (state == Smp::MSK_Created)
    {
        state = Smp::MSK_Publishing;

        if (receiver)
        {
            receiver->PublishField("counter", "Counter state", &counter);
        }
    }
    else
    {
        throw Smp::IModel::InvalidModelState(state, Smp::MSK_Created);
    }
}
```

Figure 3-4: Publishing a field

The generic `PublishField()` method gets passed name, description and memory address of the field, and one of the available types. Optional parameters allow excluding a field from the visible fields (`view`), or from storing its value on store (`state`). They both default to `true`, which means that the field will be visible and included in breakpoints. The code shown here uses an overloaded method that identifies the field type (here: `Int64`) from the pointer passed to it.

For details about the publication mechanism, see section 6 of the C++ Mapping [AD-3].

 © EUROPEAN SPACE AGENCY, 2005

3.4 How to send a message to the Logger

Now we assume that we want to send a message to the logger whenever the counter is reset. This can be accomplished using the Logger Service of the simulation environment. This service can be queried via the `ISimulator` interface.

The model already implements the `Connect()` method, which gets passed a reference to the `ISimulator` interface. This method can be used to query for a reference to the logger service, which can be stored in a private field. See Figure 3-5 for the code snippets needed.

```
using namespace Smp::Services;
...
    ILogger* logger;              ///< Logger service
...
void Counter::Connect(Smp::ISimulator* simulator)
{
    if (state == Smp::MSK_Configured)
    {
        state = Smp::MSK_Connected;

        if (simulator)
        {
            // Get simulation service
            logger = simulator->GetLogger();
        }
    }
    else
    {
        throw Smp::IModel::InvalidModelState(state, Smp::MSK_Configured);
    }
}
```

Figure 3-5: Querying the logger service

With the reference to the logger service, the model can easily add a message in the `Reset()` method, as shown in Figure 3-6.

```
void Counter::Reset(void)
{
    CounterClass::Reset();

    logger->Log(this, "Reset counter", LMK_Information);
}
```

Figure 3-6: Logging a message to the logger

For details about the Logger Service, see the SMP 2.0 Component Model document [AD-2, Section 4.1.1].

3.5 How to add the model to the Scheduler

Now, the objective is to register the `Count()` method with the scheduler service. As with all other services, the model has to query a reference to the scheduler service before it can use it. Again, this can be done in the `Connect()` method.

Only entry points (methods that have neither parameters nor a return value) can be added to the scheduler. The `Count()` method is such an entry point. Before it can be added, it has to be turned into an instance of

the IEntryPoint interface. For this purpose, a private, nested class CounterEntryPoint has been defined. Its implementation is shown in Figure 3-7[9].

```
/// Private helper class for entry points
class CounterEntryPoint : public Smp::IEntryPoint
{
private:
    char* name;                          ///< Name of entry point.
    char* description;                   ///< Description.
    Counter* publisher;                  ///< Entry point publisher.
    void (Counter::*entryPoint)(void);   ///< Instance method.
public:
    /// Constructor with name, description, publisher and entry point.
    CounterEntryPoint(
        Smp::String8 name,
        Smp::String8 description,
        Counter* publisher,
        void (Counter::*entryPoint)(void))
    {
        this->name        = strdup(name);
        this->description = strdup(description);
        this->publisher   = publisher;
        this->entryPoint  = entryPoint;
    }
    virtual Smp::String8 GetName()        const { return name; }
    virtual Smp::String8 GetDescription() const { return description; }
    virtual Smp::IComponent* GetOwner()   const { return publisher; }
    /// Execute the entry point.
    void Execute() const { (publisher->*entryPoint)(); }
};
```

Figure 3-7: Private class implementing the IEntryPoint interface

The private class CounterEntryPoint provides an implementation of the Execute() method of the IEntryPoint interface by simply calling the instance method with the instance passed to it in the constructor. With this class, the entry point can be registered as new CounterEntryPoint("Count", "Increment Counter", this, &Counter::Count), e.g. as a cyclic event with start and cycle time (see Figure 3-8).

```
    IScheduler* scheduler;                      ///< Scheduler service
    CounterEntryPoint* count;                   ///< Entry point for Count
...
void CounterModel::Connect(Smp::ISimulator* simulator)
{
    logger    = simulator->GetLogger();
    scheduler = simulator->GetScheduler();

    count     = new CounterEntryPoint("Count", "Increment Counter", this,
                         &Counter::Count);

    scheduler->AddSimulationTimeEvent(count, 0, 1000000000);
}
```

Figure 3-8: Registering the Count method with the scheduler

Like SMP1, SMP2 measures time in nanoseconds[10].

[9] *Note that in a supporting MDK, a template class could be provided to minimise the implementation effort.*

[10] *Note that a supporting MDK could define convenience functions to specify time values in seconds or milliseconds as well, but the examples in this section only use the platform mapping.*

 © EUROPEAN SPACE AGENCY, 2005

For details about the scheduler service, see the SMP 2.0 Component Model [AD-2, Section 4.1.3].

Note that the above example only explains *self-scheduling*, where the model adds its own entry points to the scheduler. However, the scheduler service may also be used in managed simulations with *external scheduling*, in which case it is typically called by the Model Manager that gets the scheduling information from an SMDL Schedule file; for details see 2.7.7 (Managed Models) and the SMP 2.0 Metamodel [AD-1]

3.6 How to register with a global Event

Finally, we want to listen to state changes of the simulation environment. Whenever the state changes to `Standby`, the counter should be reset using its `Reset()` method. This can be accomplished using the Event Manager Service. As with all other services, the model has to query a reference to the event manager service before it can use it. Again, this can be done in the `Connect()` method.

As with the scheduler service, only entry points can be registered with the event manager. With the nested class defined above, the `Reset()` method can be easily registered with the event manager (see Figure 3-9). For the state transition to `Standby` state (which is typically triggered by the `Hold()` command), a predefined event name and identifier exist. In a call to the scheduler, the `Reset()` method is (via the `IEntryPoint` mechanisms) registered with this event.

```
private:
    // Private references to services
    Smp::Services::ILogger* logger;             // Reference to Logger
    Smp::Services::IScheduler* scheduler;       // Reference to Scheduler
    Smp::Services::IEventManager* eventManager; // Reference to EventManager
    CounterEntryPoint* count;                   // Entry point for Count
    CounterEntryPoint* reset;                   // Entry point for Reset
...
// Get access to services, register entry points with scheduler and event manager.
void CounterModel::Connect(Smp::ISimulator *simulator) throw
(Smp::IModel::InvalidModelState)
{
    logger       = simulator->GetLogger();
    scheduler    = simulator->GetScheduler();
    eventManager = simulator->GetEventManager();

    // Register cyclic event with scheduler
    count = new CounterEntryPoint("Count", "Increment Counter", this,
                      &Counter::Count);
    scheduler->AddSimulationTimeEvent(count, 0, 1000000000);

    // Register event handler with event manager
    reset = new CounterEntryPoint("Reset", "Reset Counter",     this,
                      &Counter::Reset);
    eventManager->Subscribe(SMP_EnterStandbyId, reset);
}
```

Figure 3-9: Registering the `Reset` method with the event manager

For details about the Event Manager Service, see the SMP 2.0 Component Model [AD-2, Section 4.1.4].

3.7 The complete model

In this section, the complete definition and implementation of the model are summarised. Note that some include files have been added, which assume that the header files are named according to the namespaces and interfaces used in the SMP 2.0 Component Model.

Note: The `Configure()` method implementation is empty for this model, except for the state transition.

3.7.1 The definition file

```
#include "CounterClass.h"
#include "Smp/IModel.h"
#include "Smp/IEntryPoint.h"
#include "Smp/Services/ILogger.h"
#include "Smp/Services/IScheduler.h"
#include "Smp/Services/IEventManager.h"

/// This class turns the CounterClass into an Smp model.
class Counter : public CounterClass, public virtual Smp::IModel
{
private:
    /// Private helper class for entry points
    class CounterEntryPoint : public Smp::IEntryPoint
    {
    private:
        char* name;                         ///< Name of entry point.
        char* description;                  ///< Description.
        Counter* publisher;                 ///< Entry point publisher.
        void (Counter::*entryPoint)(void);  ///< Instance method.
    public:
        /// Constructor with name, description, publisher and entry point.
        CounterEntryPoint(
            Smp::String8 name,
            Smp::String8 description,
            Counter* publisher,
            void (Counter::*entryPoint)(void))
        {
            this->name        = strdup(name);
            this->description = strdup(description);
            this->publisher   = publisher;
            this->entryPoint  = entryPoint;
        }
        virtual Smp::String8 GetName()        const { return name; }
        virtual Smp::String8 GetDescription() const { return description; }
        virtual Smp::IComponent* GetOwner()   const { return publisher; }
        void Execute() const { (publisher->*entryPoint)(); }
    };
private:
    char* name;                                 ///< Name of model.
    Smp::ModelStateKind state;                  ///< Model state.
    Smp::IComposite *parent;                    ///< Parent component.
    Smp::Services::ILogger *logger;             ///< Logger service.
    Smp::Services::IScheduler *scheduler;       ///< Scheduler service.
    Smp::Services::IEventManager *eventManager; ///< EventManager service.
    CounterEntryPoint *count;                   ///< Entry point for Count.
    CounterEntryPoint *reset;                   ///< Entry point for Reset.
    void Init();                                ///< Init private fields.
public:
    Counter(Smp::String8 name, Smp::IComposite *parent) : CounterClass()
    {
        Init();
        this->name   = strdup(name);
        this->state  = Smp::MSK_Created;
        this->parent = parent;
    }
    virtual ~Counter()
    {
        if (name) free(name);
    }
    virtual void Reset(void);                   ///< Reset method logs a message
    // IModel methods
    virtual Smp::String8 GetName()         const { return name; }
    virtual Smp::String8 GetDescription()  const { return "Counter Model"; }
    virtual Smp::IComposite* GetParent()   const { return parent; }
    virtual Smp::ModelStateKind GetState() const { return state; }
    virtual void Publish(Smp::IPublication* receiver)      throw (InvalidModelState);
    virtual void Configure(Smp::Services::ILogger* logger) throw (InvalidModelState);
    virtual void Connect(Smp::ISimulator* simulator)       throw (InvalidModelState);
};
```

Figure 3-10: `Counter.h`: Definition of Counter model

© EUROPEAN SPACE AGENCY, 2005

3.7.2 The implementation file

```
#include <iostream>
#include "Counter.h"
#include "Smp/ISimulator.h"
#include "Smp/IPublication.h"

// Initialise private fields
void Counter::Init(void)
{
    // Initialise references
    logger       = NULL;
    scheduler    = NULL;
    eventManager = NULL;
    // Turn Count and Reset methods into EntryPoints
    count = new CounterEntryPoint("Count", "Increment Counter", this,
                                  &Counter::Count);
    reset = new CounterEntryPoint("Reset", "Reset Counter",     this,
                                  &Counter::Reset);
}

// Log a message to the logger on Reset
void Counter::Reset(void)
{
    CounterClass::Reset();
    // Send an information message to the logger
    logger->Log(this, "Reset counter", Smp::Services::LMK_Information);
}

// Publish fields to environment
void Counter::Publish(Smp::IPublication *receiver)
{
    if (state == Smp::MSK_Created)
    {
        state = Smp::MSK_Publishing;
        receiver->PublishField("counter", "Counter state", &counter);
    }
    else
    {
        throw Smp::IModel::InvalidModelState(state, Smp::MSK_Created);
    }
}

// Perform custom configuration.
void Counter::Configure(Smp::Services::ILogger*) throw
(Smp::IModel::InvalidModelState)
{
    if (state == Smp::MSK_Publishing)
        state = Smp::MSK_Configured;
    else
        throw Smp::IModel::InvalidModelState(state, Smp::MSK_Publishing);
}

// Get access to services, and register entry points.
void Counter::Connect(Smp::ISimulator *simulator)
{
    if (state == Smp::MSK_Configured)
    {
        state = Smp::MSK_Connected;
        logger       = simulator->GetLogger();
        scheduler    = simulator->GetScheduler();
        eventManager = simulator->GetEventManager();

        scheduler->AddSimulationTimeEvent(count, 0, 1000000000);
        eventManager->Subscribe(Smp::Services::SMP_EnterStandbyId, reset);
    }
    else
    {
        throw Smp::IModel::InvalidModelState(state, Smp::MSK_Configured);
    }
}
```

Figure 3-11: `Counter.cpp`: Implementation of Counter model

The Counter model provides all mechanisms needed in order to use it in a simulation:

1. It can be given a name passing it to its constructor
2. It can be added to the model tree passing its parent component to its constructor
3. It actively publishes its counter field
4. It actively adds the Count() entry point to the scheduler
5. It actively adds the Reset() entry point to the event manager

The above list shows the functionality that the model provides, which allows integrating instances of this model into a (managed) simulation. This is shown in the following code snippet, where a new instance is created and then added to a managed container (its parent). If the managed container is the "Models" container of the simulator, the model is a root model; otherwise it is a child model of some other model.

```
Smp::Management::IManagedContainer* container;
...
Counter* counterModel = new Counter("Counter", container);
container->AddComponent(counterModel);
```

3.8 How to turn it into a Managed Model

This final subsection shows how the same model can be turned into a managed model. To be managed, the model has to implement IManagedModel, which gives other components (typically a model manager) write access to its name, description and parent, and read and write access to its fields. As the counter model has entry points, it needs to publish these via the IEntryPointPublisher interface.

The code in Figure 3-12 shows the modifications needed in the definition of the counter model to make it a managed model. The following has been changed (please note that the declaration does not include the throw clauses of the exceptions that may be thrown be the methods):

1. The class implements the interfaces IManagedModel and IEntryPointPublisher.
2. A private field description has been added, which is returned by GetDescription().
3. A private field entryPoints has been added, to store the collection of entry points.
4. A second constructor without arguments has been added, as a managed component allows setting name, description and parent later.

The code snippets in Figure 3-13 show the additional code to make the counter example a managed model. The following has been changed:

5. The private method Init() has been modified to initialise the entryPoints collection of entry points.
6. The methods of IManagedModel are implemented, except for GetArrayValue() and SetArrayValue(), which are defined to be empty in the header file[11].
7. The two methods of IEntryPointPublisher are implemented.

Note: A managed model would typically *not* register one of its entry points with the scheduler, as this is part of model management. This is the main reason why models may expose their entry points via the IEntryPointPublisher interface.

Note: For a managed model, it is important to initialise all private fields immediately from the constructor, as other models or a model manager may already use their values (e.g. via GetEntryPoints()). Calling Init() from the Publish() method may be too late, as entry points can be registered by an external model manager already in the Creating state, which is entered before the Publishing state.

---

[11] *In a real implementation, these methods should not be empty (as the model has no array fields), but should throw an exception InvalidFieldName.*

© EUROPEAN SPACE AGENCY, 2005

Changes (except for the different class name) are highlighted in yellow.

```
#include "CounterClass.h"
#include "Smp/IModel.h"
#include "Smp/IEntryPoint.h"
#include "Smp/Services/ILogger.h"
#include "Smp/Services/IScheduler.h"
#include "Smp/Services/IEventManager.h"
#include "Smp/Management/IManagedModel.h"
#include "Smp/Management/IEntryPointPublisher.h"

// Definition of Managed Counter
class ManagedCounter : public CounterClass,
    public virtual Smp::Management::IManagedModel,
    public virtual Smp::Management::IEntryPointPublisher
{
private:
    // Private class for entry points
    // This class has not changed: See Counter.h for its definition.
private:
    char* name;                                  ///< Name of model.
    char* description;                           ///< Description of model.
    Smp::ModelStateKind state;                   ///< Model state.
    Smp::IComposite *parent;                     ///< Parent component.
    Smp::Services::ILogger *logger;              ///< Logger service.
    Smp::Services::IScheduler *scheduler;        ///< Scheduler service.
    Smp::Services::IEventManager *eventManager; ///< EventManager service.
    CounterEntryPoint *count;                    ///< Entry point for Count.
    CounterEntryPoint *reset;                    ///< Entry point for Reset.
    Smp::EntryPointCollection *entryPoints;      ///< Collection of entry points.
    void Init();                                 ///< Init private fields.
public:
    // Public constructors
    ManagedCounter();
    ManagedCounter(Smp::String8 name, Smp::IComposite* parent) : CounterClass()
    {
        Init();
        this->name   = strdup(name);
        this->state  = Smp::MSK_Created;
        this->parent = parent;
    }
    virtual ~ManagedCounter()
    {
        if (name) free(name);
        if (description) free(description);
    }
    virtual void Reset(void);                    ///< Reset method logs a message
    // IObject, IComponent and IModel methods
    virtual Smp::String8 GetName()         const { return name; }
    virtual Smp::String8 GetDescription()  const { return description; }
    virtual Smp::IComposite* GetParent()   const { return parent; }
    virtual Smp::ModelStateKind GetState() const { return state; }
    virtual void Publish(Smp::IPublication *receiver)
                                          throw (Smp::IModel::InvalidModelState);
    virtual void Configure(Smp::Services::ILogger* logger)
                                          throw (Smp::IModel::InvalidModelState);
    virtual void Connect(Smp::ISimulator *simulator)
                                          throw (Smp::IModel::InvalidModelState);
    // IManagedObject, IManagedComponent and IManagedModel methods
    virtual void SetName(Smp::String8 name);
    virtual void SetDescription(Smp::String8 description);
    virtual void SetParent(Smp::IComposite *parent);
    virtual Smp::AnySimple GetFieldValue(Smp::String8 fullName);
    virtual void SetFieldValue(Smp::String8 fullName, const Smp::AnySimple value);
    virtual void GetArrayValue(Smp::String8 fullName, const Smp::AnySimpleArray
                                          values, const Smp::Int32 length) {}
    virtual void SetArrayValue(Smp::String8 fullName, const Smp::AnySimpleArray
                                          values, const Smp::Int32 length) {}
    // IEntryPointPublisher methods
    virtual const Smp::EntryPointCollection *GetEntryPoints() const;
    virtual Smp::IEntryPoint *GetEntryPoint(Smp::String8 entryPointName) const;
};
```

Figure 3-12: `ManagedCounter.h`

Changes (except for the different class name, and the new methods) are highlighted in yellow.

```
void ManagedCounter::Init(void)
{
    ...
    description = NULL;
    entryPoints = new Smp::EntryPointCollection();
    entryPoints->push_back(count);
    entryPoints->push_back(reset);
}

void ManagedCounter::SetName(Smp::String8 name)
{
    if (this->name) free(this->name);
        this->name = strdup(name);
}

void ManagedCounter::SetDescription(Smp::String8 description)
{
    if (this->description) free(this->description);
        this->description = strdup(description);
}

void ManagedCounter::SetParent(Smp::IComposite *parent)
{
    this->parent = parent;
}

Smp::AnySimple ManagedCounter::GetFieldValue(Smp::String8 fieldName)
{
    Smp::AnySimple returnValue;

    if (strcmp(fieldName, "counter") == 0)
    {
        returnValue.type = Smp::ST_Int64;
        returnValue.value.int64Value = counter;
        return returnValue;
    }
    else
        throw InvalidFieldName(fieldName);
}

void ManagedCounter::SetFieldValue(Smp::String8 fieldName, const Smp::AnySimple
value)
{
    if (strcmp(fieldName, "counter") == 0)
    {
        if (value.type == Smp::ST_Int64)
            counter = value.value.int64Value;
        else
            throw Smp::InvalidAnyType(value.type, Smp::ST_Int64);
    }
    else
        throw InvalidFieldName(fieldName);
}

const Smp::EntryPointCollection *ManagedCounter::GetEntryPoints() const
{
    return entryPoints;
}

Smp::IEntryPoint* ManagedCounter::GetEntryPoint(Smp::String8 entryPointName) const
{
        if (strcmp(entryPointName, "Count") == 0)
                return count;
        else if (strcmp(entryPointName, "Reset") == 0)
                return reset;
        else
                return NULL;
}
```

Figure 3-13: `ManagedCounter.cpp` (modifications only)

© EUROPEAN SPACE AGENCY, 2005

4. THE SIMULATION ENVIRONMENT

This section summarises mandatory and optional elements that each SMP2 simulation environment may provide. Further, it introduces certain states as well as state transitions the environment has to support. For a more detailed description of the simulation states and associated interfaces, see the SMP 2.0 Component Model [AD-2, Section 3.5].

4.1 Simulation Environment State Diagram

The Simulation Environment always has to be in a well-defined state. These states cover the different operational phases introduced in 2.6.

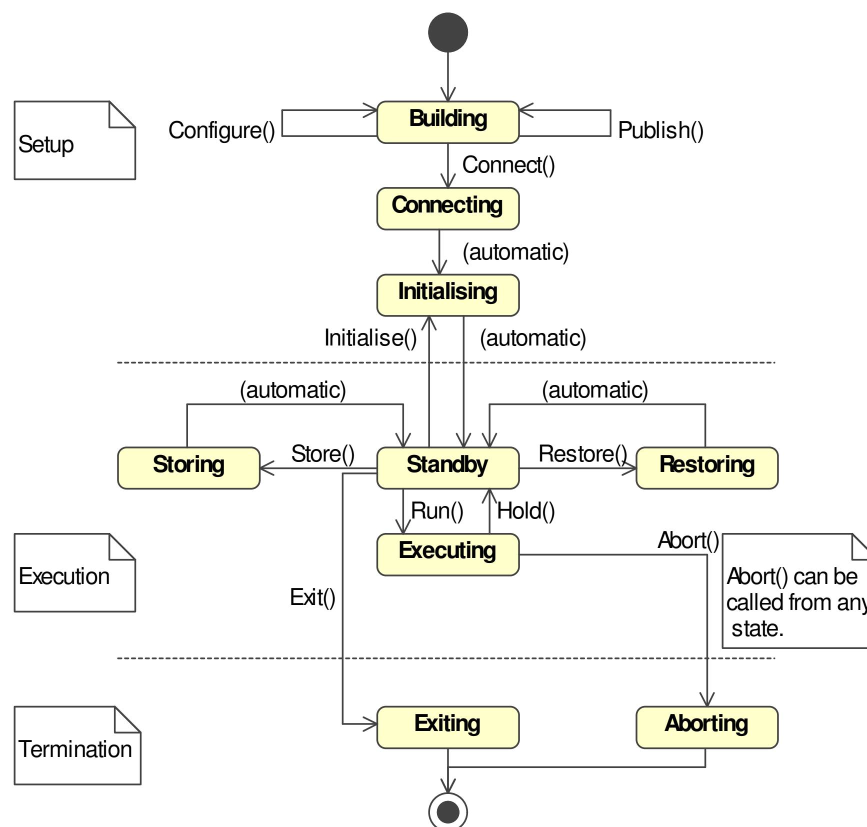

Figure 4-1: UML State Diagram: Simulation Environment State Diagram with State Transition Methods

While SMP2 standardises on a minimal number of states, a simulation environment may well support other states. A typical additional state would be `Debugging`. However, SMP2 does neither mandate additional states nor specific state transitions between them.

The simulation environment is required to emit an event via the event manager service whenever it leaves the current state or when it enters a new state. For model instances, the following six states are of main interest:

- `Standby`, to allow models to implement different behaviour in standby and executing modes. In `Standby` state, simulation time is not progressing.
- `Executing`, to allow models to implement different behaviour in standby and executing modes. In `Executing` state, simulation time is progressing.
- `Storing`, to allow models to update their state before it gets stored.

- `Restoring`, to allow models to update their state after it got restored.
- `Exiting`, to allow models clean up properly, e.g. releasing resources such as file handles.
- `Aborting`, which indicates an abnormal termination of a simulation, where only critical clean up (e.g. releasing external resources) shall be performed by the models.

For the other states, either no model instances do exist (`Creating` state), or the simulation environment calls them anyway (`Publishing`, `Connecting`, and `Initialising` state).

All mandatory states are explained below. State transitions are either initiated by calling methods of the `ISimulator` interface (see 4.2.1) or automatic after all tasks within a state have been completed. Standard events are associated with each state transition, which the simulation environment has to emit via the Event Manager service on leaving or entering each state. The event names thereby follow the scheme `Leave<State>` and `Enter<State>`. For details see the SMP 2.0 Component Model [AD-2, Section 4.1.4.2].

4.1.1 The `Building` state

This state is entered automatically after the simulation environment has performed its private initialisation (e.g. creation of simulation services).

In `Building` state, the model hierarchy is created and configured. This task includes

- creating all model instances,
- building the hierarchy of model instances,
- asking the model instances to publish their field, operations and properties,
- possibly setting initial values of fields (may also be done in the `Configuring` state),
- possibly connecting model instances by links (may also be done in the `Configuring` state).

This process can be done in one of the two following ways:

a. An external component creates a static scenario of model instances, sets their field values, and creates their relationships.

b. The simulation environment loads an SMDL assembly document, creates all model instances, sets their initial field values, and connects those using links.

While in Building state, the `Publish()` method can be called any time. This asks the simulator to walk through the hierarchy of model instance and call the `Publish()` method of all newly created model instances. This allows setting their field values, e.g. from information in an SMDL assembly document.

While in Building state, the `Configure()` method can be called any time. This asks the simulator to walk through the hierarchy of model instance and call the `Configure()` method of all model instances that are still in `Publishing` state. This should be done when these model instances have been fully configured.

In any case, the end of this phase has to be indicated by calling the `Connect()` method of the simulator (in other words: by initiating the `Connect()` state transition).

4.1.2 The `Connecting` state

This state is entered using the `Connect()` state transition.

The simulation environment traverses the complete model hierarchy and calls the `Connect()` method of each model, passing the `ISimulator` interface that allows models to query for services they need.

 © EUROPEAN SPACE AGENCY, 2005

The simulation environment has to ensure that all services have been created and are operational when entering the `Connecting` state. It needs to provide all mandatory simulation services, and all optional simulation services that it chooses to provide.

All services must be held in the `"Services"` container of the simulator, which provides an alternative way to access services. This allows, for example, iterating on all existing services in order to query services by type/interface – instead of querying by name via the `GetService()` method.

At the end of this phase, the simulation environment automatically enters the `Initializing` state.

4.1.3 The `Initialising` state

This state is entered automatically after the `Connecting` state, or using the `Initialise()` state transition from the `Standby` state.

The latter may be used to implement a custom *reset mechanism* in a simulation environment, which may first restore a state vector that has been initially stored after the `Connecting` state, and then allow models to perform their custom initialisation again, based on the values from the restored state vector.

In `Initialising` state, the simulation environment calls dedicated initialisation entry points (in the order they have been added to it using the `AddInitEntryPoint()` method), allowing them to perform any custom initialisation. Thereby it is guaranteed that all models have their initial values and are properly linked together.

At the end of this state, the simulation environment automatically enters the `Standby` state.

4.1.4 The `Standby` state

This state is entered automatically after the `Initialising` state, or using the `Hold()` state transition from the `Executing` state. It is also automatically entered from the `Storing` and `Restoring` states.

When in this state, the simulation environment (namely the Time Keeper Service) does not progress simulation time. Therefore, entry points registered with the Scheduler Service using simulation, mission, or epoch time are not executed. However, as Zulu time (which is typically connected to the system time of the computer) still progresses, events specified relative to this time will get executed.

In any case, the end of this state has to be indicated by calling one of the `Run()`, `Store()`, `Restore()`, `Initialise()`, or `Exit()` methods of the simulator.

4.1.5 The `Executing` state

This state is entered from the `Standby` state using the `Run()` state transition.

In this state, the simulation environment does progress simulation time. Entry points registered with any of the four available time kinds may get executed.

In any case, the end of this state has to be indicated by calling the `Hold()` method of the simulator.

4.1.6 The `Storing` state

This state is entered from the `Standby` state using the `Store()` state transition.

In this state, the simulation environment first stores the values of all fields published with the `State` attribute to storage (typically a file). Afterwards, the `Save()` method of all components implementing the optional `IPersist` interface is called, to allow custom storing of additional information.

© EUROPEAN SPACE AGENCY, 2005

While in this state, simulation time is not progressed, and models must not change their persistent state in order to ensure that consistent state information is stored.

At the end of this state, the simulation environment automatically enters the `Standby` state.

4.1.7 The `Restoring` state

This state is entered from the `Standby` state using the `Restore()` state transition.

In this state, the simulation environment first restores the values of all fields published with the `State` attribute from storage. Afterwards, the `Load()` method of all components implementing the optional `IPersist` interface is called, to allow custom restoring of additional information.

While in this state, simulation time is not progressed, and models must not change their persistent state in order to ensure that consistent state information is restored.

At the end of this state, the simulation environment automatically enters the `Standby` state.

4.1.8 The `Exiting` state

This state is entered from the `Standby` state using the `Exit()` state transition.

In this state, the simulation environment is properly terminating a running simulation. All components (especially model instances) are required to perform a clean up when required, e.g. to close opened files and release the file handlers. Note that models are not explicitly called during this state. However, they may register with the standard `EnterExiting` event to get a notification.

After exiting, the simulator is in an undefined state.

4.1.9 The `Aborting` state

This state is entered from any other state using the `Abort()` state transition.

In this state, the simulation environment performs an abnormal simulation shutdown. Only critical clean up (e.g. release of external resources) shall be performed. Note that models are not explicitly called during this state. However, they may register with the standard `EnterAborting` event to get a notification.

After aborting, the simulator is in an undefined state.

4.2 Simulation Environment Interfaces

Every simulation environment has to provide certain functionality that other components (typically model instances or a model manager) can use via dedicated interfaces. As SMP2 defines a component-based architecture, it is not required that the simulation environment has to implement all these functions itself. However, it has to

1. give access to the simulation state, and to state transition operations (`ISimulator`),
2. give access to the (root) models (`ISimulator`),
3. give access to the services (`ISimulator`), providing at least the mandatory services,
4. give access to the `"Models"` and `"Services"` containers (`IComposite`), and
5. give access to a publication mechanism (`IPublication`).

© EUROPEAN SPACE AGENCY, 2005

Optionally, a simulation environment may support managed models and the associated XML file formats. This includes to …

6. provide the optional services as well,
7. give access to component factories (`IDynamicSimulator`),
8. read model specification and metadata from catalogues,
9. read model initialisation information from assemblies,
10. read model scheduling information from schedules.

A Simulation Environment could look like this.

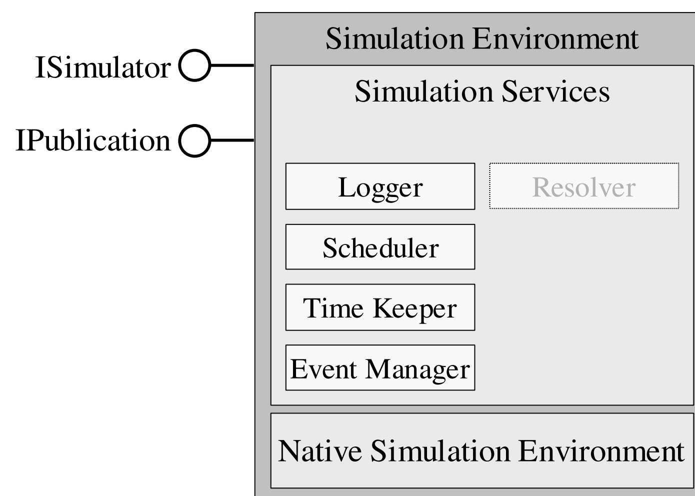

Figure 4-2: Example of Simulation Environment implementing standard Simulation Environment interfaces

4.2.1 The `ISimulator` interface

The simulation environment *shall* implement the interface `ISimulator`, which defines mandatory methods allowing to switch between simulation states and to query the simulation state. Furthermore, it gives simplified access to models and services via convenience methods working on the `"Models"` and `"Services"` containers.

© EUROPEAN SPACE AGENCY, 2005

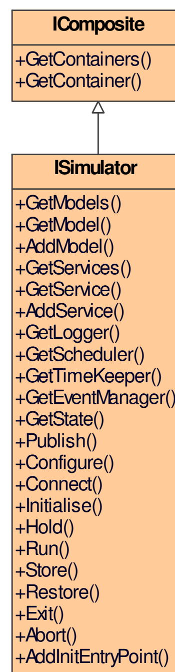

Figure 4-3: UML Class Diagram: `ISimulator` interface

4.2.1.1 The `Publish()` method

This method informs the simulation environment that new model instances have been created, and can now be published. Within this method, the simulation environment calls the `Publish()` method of each model instance in the model hierarchy that has been created (i.e. is in `Created` state).

This method can only be called when in `Building` state, see 1.1.1 for details.

This method is typically called by an external component, for example by a model manager, after creating new model instances.

4.2.1.2 The `Configure()` method

This method informs the simulation environment that all model instances have been configured, and can now perform their custom configuration. Within this method, the simulation environment calls the `Configure()` method of each model instance in the model hierarchy that has not been configured yet (i.e. is in `Publishing` state).

This method can only be called when in `Building` state, see 1.1.1 for details.

This method is typically called by an external component, for example by a model manager, after configuring new model instances.

4.2.1.3 The `Connect()` method

This method informs the simulation environment that the hierarchy of model instances has been created and configured, and can now be connected to the simulator. After this call, the simulation environment calls the `Connect()` method of every model in the model hierarchy, and then performs an automatic state transition from `Connecting` to `Initialising` state.

This method switches from `Building` to `Connecting` state, see 4.1.2 for details.

 © EUROPEAN SPACE AGENCY, 2005

This method is typically called by an external component, for example by a model manager, after creating and configuring all model instances.

4.2.1.4 The `Initialise()` method

This method asks the simulation environment to call all initialisation entry points again.

This method switches from `Standby` to `Initialising` state, see 4.1.3 for details.

4.2.1.5 The `Run()` method

This method switches from `Standby` to `Executing` state, see 4.1.5 for details.

4.2.1.6 The `Hold()` method

This method switches back from `Executing` to `Standby` state, see 4.1.4 for details.

4.2.1.7 The `Store(in String8 filename)` method

This method is used to store a state vector to file.

This method switches from `Standby` to `Storing` state, see 4.1.6 for details.

4.2.1.8 The `Restore(in String8 filename)` method

This method is used to restore a state vector from file.

This method switches from `Standby` to `Restoring` state, see 4.1.7 for details.

4.2.1.9 The `Exit()` method

This method is used for a normal termination of a simulation.

This method switches from `Standby` to `Exiting` state, see 4.1.8 for details.

4.2.1.10 The `Abort()` method

This method is used for an abnormal termination of a simulation.

This method switches from any other state to `Aborting` state, see 4.1.9 for details.

4.2.1.11 The `GetState()` method

This method returns the current simulation state. As it does not change the simulation state, it can be called from any state.

4.2.1.12 The `GetModels()` method

This method returns a collection of root models. Root models are immediate children of the `"Models"` container that the Simulator provides.

Root models that implement the `IComposite` interface may themselves provide models, called sub-models. These build a hierarchy of models. However, this method only returns root models, i.e. models that are not contained within any other model.

4.2.1.13 The `GetModel(in String8 name)` method

This method queries an existing root model by name. It either returns the `IModel` interface of the model, or null if the given model name could not be found.

This method does not resolve paths to sub-models, but only returns root models that reside in the `"Models"` container. Use the Resolver service to query sub-models.

4.2.1.14 The `AddModel(in IModel model)` method

This method allows adding a root model to the simulator, i.e. to the `"Models"` container.

4.2.1.15 The `GetServices()` method

This method returns a collection of all registered simulation services, which includes pre-defined services as well as user-defined services. Services are held in the `"Services"` container of the Simulator.

4.2.1.16 The `GetService(in String8 name)` method

This method queries an existing service by name. It either returns the `IService` interface of the service, or null if the given service name could not be found.

Standard names are defined for the standardised services, while user-defined services use custom names.

The existence of optional and user-defined services is not guaranteed, so models should expect to get null.

4.2.1.17 The `AddService(in IService Service)` method

This method allows adding a user-defined service to the simulator, i.e. to the `Services` container.

It is recommended that user-defined services include a project or company acronym as prefix in their name (similar to the predefined services using the `Smp` prefix), to avoid collision of service names.

4.2.1.18 The `GetLogger()` method

This method returns the logger service. It is a convenience method that avoids having to call `GetService()` first, and then casting the `IService` interface to the `ILogger` interface.

4.2.1.19 The `GetScheduler()` method

This method returns the scheduler service. It is a convenience method that avoids having to call `GetService()` first, and then casting the `IService` interface to the `IScheduler` interface.

4.2.1.20 The `GetTimeKeeper()` method

This method returns the time keeper service. It is a convenience method that avoids having to call `GetService()` first, and then casting the `IService` interface to the `ITimeKeeper` interface.

4.2.1.21 The `GetEventManager()` method

This method returns the event manager service. It is a convenience method that avoids having to call `GetService()` first, and then casting the `IService` interface to the `IEventManager` interface.

© EUROPEAN SPACE AGENCY, 2005

4.2.2 The `IComposite` interface

The simulation environment *shall* implement the interface `IComposite`, which gives access to the components within the simulation. As a minimum, the simulation environment can hold two types of child components: model instances and simulation services. To give access to both, it needs to return at least two containers in its implementation of `IComposite`. As both models and services can be added to the simulation environment from external components, both containers need to be managed containers, i.e. need to support the `AddComponent()` method.

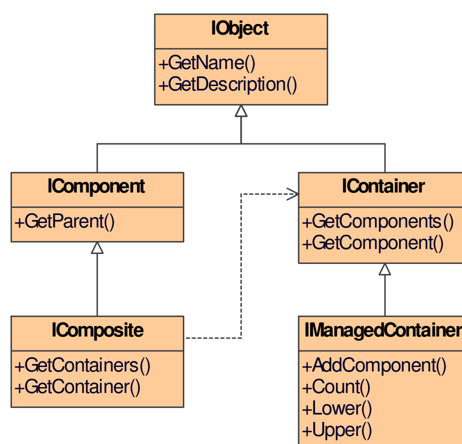

Figure 4-4: UML Class Diagram: `IComposite`, `IContainer`, `IManagedContainer` interfaces

4.2.2.1 The `Models` container

The simulation environment *shall* return an `IManagedContainer` when its `GetContainer()` method is called with the pre-defined container name `"Models"`. This container needs to be managed, as it needs to allow adding root models to the simulation via its `AddComponent()` method.

Note, however, that the standard way of adding root models is via the `AddModel()` method of the `ISimulator` interface.

4.2.2.2 The `Services` container

The simulation environment *shall* return an `IManagedContainer` when its `GetContainer()` method is called with the pre-defined container name `"Services"`. This container needs to be managed, so that user-defined services can be added to it via its `AddComponent()` method.

Note, however, that the standard way of adding services is via the `AddService()` method of the `ISimulator` interface.

4.2.3 The `IPublication` interface

The simulation environment needs to provide access to a component implementing the `IPublication` interface. While it typically implements this interface itself, it may as well delegate the functionality to another component. Publication is a platform dependent mechanism. Only its **existence** is specified by the platform independent model, not its **realisation**.

For the C++ language mapping, the `IPublication` interface provides methods to publish fields, operations and properties, but as well functions to register user-defined types, which can be floats, integers, enumerations, arrays, strings, structures and classes. All user-defined types derive from the `IType`

interface, and for convenience, a mapping of the pre-defined simple types to this interface is provided as well.

For the unique identification of types, Universally Unique Identifiers (**UUID**s) are used.

See the SMP 2.0 Metamodel document for user-defined types [AD-1].

See the SMP 2.0 C++ Mapping document for publication in C++ [AD-3].

4.2.4 The `IDynamicSimulator` interface

The simulation environment *may* implement the `IDynamicSimulator` interface. This interface extends the `ISimulator` interface and adds methods to dynamically create components (typically models) from component factories. It makes use of the `IFactory` interface for component factories.

This interface is typically needed to support dynamically configured simulations, where the configuration is loaded from external information, typically from XML documents specified by the standard (the SMP 2.0 Metamodel [AD-1]). In order to identify the model specification (in a catalogue) and the model implementation (in an assembly and in a model library), Universally Unique Identifiers (**UUID**) are used.

The implementation of this interface is platform specific and therefore not covered by SMP2. For example, one implementation could dynamically load factories from each Dynamic Link Library (**DLL**) or Dynamic Shared Object (**DSO**) that contain associated model implementations, and then register all factories in a global registry. On the other hand, another implementation may statically link all factory implementations with the implementation of the models themselves, and register them programmatically.

4.2.4.1 The `RegisterFactory(...)` method

This method registers a component factory with the dynamic simulator. The dynamic simulator can use this factory to create component instances of the component implementation in its `CreateInstance()` method.

This method is typically called early in the `Creating` state to register the available component factories before the hierarchy of model instances is created. The implementation identifiers of the registered factories need to be unique.

4.2.4.2 The `CreateInstance(in Uuid implUuid)` method

This method creates an instance of the component with the given implementation identifier.

A call to this method would look similar to the following C++ code:

```
IDynamicSimulator* dynSim = dynamic_cast<IDynamicSimulator>(simulator);

IComponent* model = dynSim->CreateInstance(modelUuid);
```

The implementation of the `CreateInstance()` method shall use the *factory pattern*; see [RD-3].

This method is typically called during the `Creating` state when building the hierarchy of models.

4.2.4.3 The `GetFactory(in Uuid implUuid)` method

This method returns the factory of the component with the given implementation identifier.

The returned factory may be null if no factory with the given implementation identifier could be found.

 © EUROPEAN SPACE AGENCY, 2005

4.2.4.4 The `GetFactories(in Uuid specUuid)` method

This method returns a collection of all factories of components with the given specification identifier.

The returned collection may be empty if no factories have been registered for the given specification identifier.

4.3 Simulation Services

For a detailed description of simulation services, see the SMP 2.0 Component Model [AD-2].

4.3.1 Mandatory Services

The standard specifies four services as being mandatory. Although every simulation environment shall provide these services, it is recommended that models check the return value of the `GetService()` method when asking for one of these services.

4.3.1.1 Logger

The logger allows logging information, event, warning, and error messages consistently. It is used by services as well as by models. In addition, the logger allows defining user-specific log message kinds as well.

4.3.1.2 Scheduler

The scheduler calls entry points based on timed or cyclic events. It closely depends on the time keeper service, as it calls events based on time.

4.3.1.3 Time Keeper

This service provides the four different time kinds of SMP2: A relative simulation time, an absolute epoch time, a relative mission time, and an absolute Zulu time which relates to the clock time of the computer.

4.3.1.4 Event Manager

This service provides mechanisms for global events: Event handlers can be registered, events can be broadcasted, and user-specific event types can be defined as well.

4.3.2 Optional Services

One additional service is standardised, but is optional. However, if the simulation environment chooses to implement a similar service as described in this section, then it *shall* use the specified semantics and the corresponding mapping into the target platform. As a simulation environment does not need to provide optional services, it is strongly recommended that models check the return value of the `GetService()` method when asking for one of these services.

4.3.2.1 Resolver

The Resolver service allows resolving references to other model instances in the model hierarchy. It works with relative names as well as with absolute names.

4.3.3 User-defined Services

These services are not specified by the standard. The `ISimulator` interface allows adding user-defined services via the `AddService()` method; see section 4.2.1.17. Note that this is a convenience method, which actually adds the service to the managed `"Services"` container of the simulation environment; see

© EUROPEAN SPACE AGENCY, 2005

section 4.2.2.2. It gets passed a reference to the `IService` interface of the user-defined service (that every user-defined service has to implement as well). As the `IService` interface is derived from `IObject`, the name of the user-defined service is defined via the `GetName()` method.

The simulation environment has to ensure that each user-defined service has a unique name, as services are requested using the `GetService()` method of the `ISimulator` interface via their names; otherwise, the `DuplicateName` exception shall be raised.

 © EUROPEAN SPACE AGENCY, 2005

5. MODEL DESIGN AND DEVELOPMENT

This section introduces different model design approaches. It starts with class-based design, then moves to interface-based design, which is further expanded to component-based design. In addition, both the event-based design and dataflow-based design are explained. Each design method is mapped to SMDL elements, with a summary of how this maps to C++.

All elements supported by SMP2 models are defined in the catalogue section of the SMP 2.0 Metamodel [AD-1]. They are mapped to C++ language elements as defined in the SMP 2.0 C++ Mapping [AD-3].

5.1 Class-based Design

A class-based design is composed out of models with fields to store an internal state, operations to implement the required behaviour, and entry points to react on timed or dynamic events. Classes provide the mechanisms of implementation inheritance, but models in SMP2 provide only single implementation inheritance.

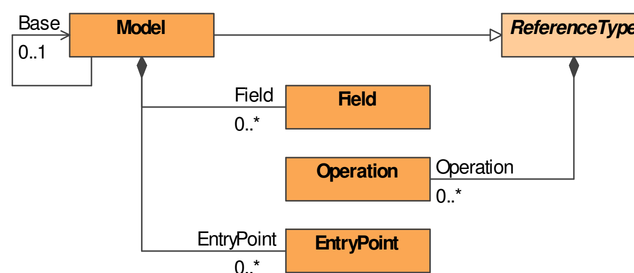

Figure 5-1: UML Class Diagram: Class-based Model

5.1.1 Design elements

The design elements needed for a class-based design are explained below.

5.1.1.1 Models

The core element of every model design in SMDL is the Model. As a minimum, it provides fields for its internal state, operations to model behaviour, and entry points to be able to register these with the scheduler or event manager services.

5.1.1.2 Fields

A field is part of the internal state of a model. Every field references a type, and may in addition have a default value. For reference types (i.e. references to classes or other models), the implicit default value is null.

A field has a visibility, which defines whether it can be accessed by the model only (`private`), by models derived from the model via implementation inheritance (`protected`), by all models in the same package (`package`), or by all models (`public`).

5.1.1.3 Operations

A method of a model can be called to perform a certain operation. It may get passed parameters, and it may have a return value. An operation of a model is a member function of this model. Whenever it gets called, it is passed a reference to a model instance (called “`this`” in C++). It then has access to all fields of the model instance.

© EUROPEAN SPACE AGENCY, 2005

An operation has a visibility, which defines whether it can be called by the model only (`private`), by models derived from the model via implementation inheritance (`protected`), by all models in the same package (`package`), or by all models (`public`).

5.1.1.4 Entry Points

An entry point is a very simple operation of a model. It translates into a member function with no parameters and no return value. Entry points are always public so that any other model or component can call them.

5.1.2 Mapping to C++

A class-based design maps to classes with fields and operations.

5.1.2.1 Mapping of Models

A `Model` is mapped as an ISO/ANSI C++ `class` that implements the `IModel` interface, i.e. has functions to return name, description, and parent, to publish fields, and to set a service provider. Optionally a model may inherit the implementation of a single base class.

Example:

```
class MyModel : public virtual Smp::IModel
{
public:
    /// Returns the name of the model.
    Smp::String8 GetName() const;

    /// Returns the description of the model.
    Smp::String8 GetDescription() const;

    /// Returns the parent component of the model.
    Smp::IComposite* GetParent() const;

    /// Get model state.
    Smp::ModelStateKind GetState() const;

    /// Asks the model to publish its fields.
    void Publish(Smp::IPublication* receiver)
    throw (Smp::IModel::InvalidModelState);

    /// Allow the model to perform custom configuration.
    void Configure(Smp::Services::ILogger* logger)
    throw (Smp::IModel::InvalidModelState);

    /// Provides access to simulation services.
    void Connect(Smp::ISimulator* simulator)
    throw (Smp::IModel::InvalidModelState);
};
```

5.1.2.2 Mapping of Fields

A `Field` is mapped to ISO/ANSI C++ as a member variable of a C++ `struct` or a C++ `class`, or as a class variable of a C++ `class` if it is static.

Example:

```
class MyModel : public virtual Smp::IModel
```

 © EUROPEAN SPACE AGENCY, 2005

```
{
private:
    /// Internal field to store the mass.
    Smp::Float64 mass;
public:
    [Skip implementation of IModel here]
};
```

5.1.2.3 Mapping of Operations

An `Operation` is mapped as an ISO/ANSI C++ member function of a C++ `struct` or a C++ `class`, or as a class function of a C++ `class` if it is static.

Example:

```
class MyModel : public virtual Smp::IModel
{
public:
    [Skip implementation of IModel here]

    /// Operation with two arguments and a return type
    virtual Smp::Float64 Product(
        const Smp::Float64 x,
        const Smp::Float64 y);
};
```

5.1.2.4 Mapping of Entry Points

An `EntryPoint` is mapped as an ISO/ANSI C++ member function with no arguments and return value. When exposing this entry point to e.g. a service, it has to be wrapped into a class implementing the `class` `IEntryPoint`, which has only one member function of type `void` with no parameters to execute the entry point.

```
class IEntryPoint
{
public:
      /// Entry point owner.
      virtual IComponent* GetOwner(void) const = 0;
      /// Execute the entry point.
      virtual void Execute(void) = 0;
};
```

Example:

```
class MyModel : public virtual Smp::IModel
{
public:
    [Skip implementation of IModel here]

    /// Entry point to be called e.g. by Scheduler
    Smp::IEntryPoint* MyEntryPoint;
};
```

5.2 Interface-based Design

An interface-based design adds interfaces as the standard mechanisms for inter-model communication. This isolates the definition of an interface (the "contract") from its implementation. In an interface-based design, a model can provide any number of interfaces.

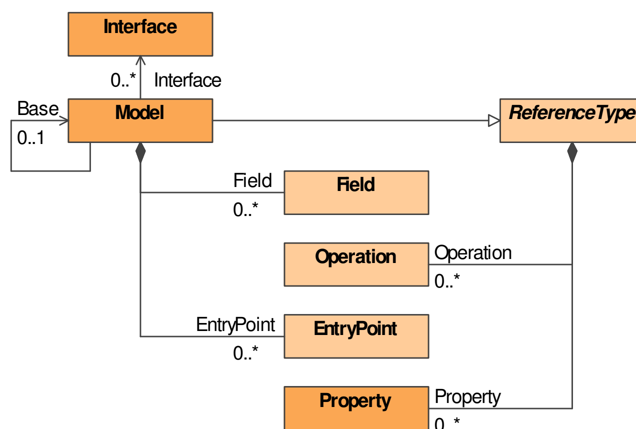

Figure 5-2: UML Class Diagram: Interface-based Model

5.2.1 Design elements

The additional design elements needed for an interface-based design are explained below.

5.2.1.1 Interfaces

An `Interface` defines a contract between models. Every model implementing the interface has to provide all the functionality of the interface, so that every model, which consumes this interface can rely on a complete implementation. As interfaces are a mechanism to de-couple models, they do not give access to fields, but only to operations. With special operations that read or write a single value, access to fields can be added.

An interface can inherit functionality from any number of other interfaces. This mechanism is called interface inheritance. As an interface is only a contract with no implementation, multiple inheritance of interfaces can be mapped to most languages not supporting multiple inheritance (e.g. Java or C#, which both natively support interfaces).

5.2.1.2 Operations

Similar to a class-based design, an interface defines operations. As an interface does not have an implementation, a model has to provide an implementation for each operation of the interfaces it provides. This implementation may be inherited from its base class.

5.2.1.3 Properties

As access to fields via interfaces is not possible, it is common to provide a pair of operations giving read and write access to a field via an interface. To ease the specification of such a pair, the `Property` element has been introduced. It typically maps to a parameter-less operation to read a value (called the getter), and a void operation with a single parameter to write the same value (called the setter). However, properties may restrict access to read-only or write-only simply by providing only one of these two methods.

 © EUROPEAN SPACE AGENCY, 2005

5.2.2 Mapping to C++

As C++ does not natively support interface-based design, its mechanisms of pure virtual functions together with multiple inheritance of classes are used to achieve the same functionality.

5.2.2.1 Mapping of Interfaces

The ISO/ANSI C++ language does not natively support interfaces, nor does it separate between interface inheritance and implementation inheritance. Therefore, an `Interface` is mapped as an abstract C++ `class`, which means that every C++ member function for an `Operation` or `Property` of the `Interface` is declared pure virtual.

Example:

```
class IReceiver
{
};

class MyModel : public virtual Smp::IModel,
    public virtual IReceiver
{
public:
    [Skip implementation of IModel here]
};
```

Multiple interface inheritance is supported because C++ supports multiple inheritance.

5.2.2.2 Mapping of Operations

An `Operation` of an `Interface` is mapped as a pure virtual ISO/ANSI C++ member function. Every non-abstract class that inherits from this interface has to provide an implementation for each operation; otherwise, it would have abstract methods, which are only allowed for abstract classes.

Example:

```
class IReceiver
{
    /// Sample method that is pure virtual.
    virtual void SampleMethod(void) = 0;
};

class MyModel : public virtual Smp::IModel,
    public virtual IReceiver
{
public:
    [Skip implementation of IModel here]

    /// Provide implementation of methods of all interfaces
    void SampleMethod(void);
};
```

5.2.2.3 Mapping of Properties

The ISO/ANSI C++ language does not natively support properties. As a property translates into one or two methods to read and/or write a value, both the getter and the setter of a `Property` are mapped as pure virtual ISO/ANSI C++ member functions of the interface.

Example:

```
class IReceiver
{
    /// Sample method that is pure virtual.
    virtual void SampleMethod(void) = 0;

    /// Property with getter and setter
    virtual Smp::Float64 get_Mass() = 0;
    virtual void set_Mass(const Smp::Float64 mass) = 0;
};


class MyModel : public virtual Smp::IModel,
    public virtual IReceiver
{
public:
    [Skip implementation of IModel here]

    /// Provide implementation of methods of all interfaces
    void SampleMethod(void);

    /// Property with getter and setter
    Smp::Float64 get_Mass();
    void set_Mass(const Smp::Float64 value);
};
```

5.3 Component-based Design

In addition to the interface for inter-model communication, a component-based design provides mechanisms to describe aggregation and composition. These mechanisms describe dependencies between models, either via references, or via containment.

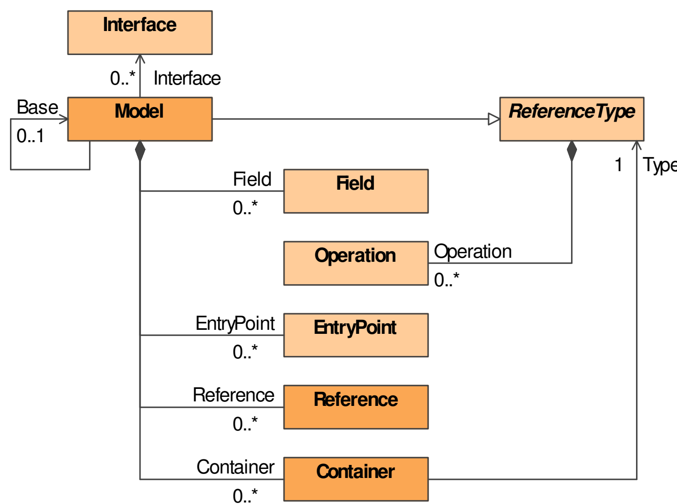

Figure 5-3: UML Class Diagram: Component-based Model

The additional design elements needed for a component-based design are explained below.

 © EUROPEAN SPACE AGENCY, 2005

5.3.1 Design elements

5.3.1.1 Aggregation

Aggregation of models means to have references to other models, but not to own these models (i.e. not be the parent model of these models). A reference to another model can either exist because a model requires another model to function properly (e.g. a payload which requires access to a specific interface of the satellite), or because a model consumes an interface (e.g. a power distribution model which controls all models that consume power).

A managed reference is a reference that allows adding new components to it. A managed reference to other models specifies lower and upper bounds for the number of models it can reference. Typical examples are:

Table 5-1: Typical bounds for a Reference

Lower bound	Upper bound	UML	Typical application
0	1	0..1	Optional reference: The model can use another model, but does not rely on it.
1	1	1	Mandatory reference: The model relies on another model providing certain functionality.
0	-	0..*	Unlimited reference: The model can consume functionality from any number of other models.

The default is a mandatory reference, i.e. lower and upper bound equal to 1.

5.3.1.2 Composition

Composition of models means to own these models (i.e. be the parent model of these models). A container of other models typically relates to (a) subsystem(s) of a system, e.g. receivers of a ground station.

A managed container is a container that allows adding new components to it. A managed container of other models specifies lower and upper bounds for the number of models it can contain. Typical examples are:

Table 5-2: Typical bounds for a Container

Lower bound	Upper bound	UML	Typical application
0	1	0..1	Optional child: The model can contain another model, but does not rely on it.
1	1	1	Mandatory child: The model relies on a child model providing certain functionality.
0	-	0..*	Unlimited children: The model can have any number of other child models of a specific type.

The default is a mandatory child, i.e. lower and upper bound equal to 1.

© EUROPEAN SPACE AGENCY, 2005

5.3.2 Mapping to C++

In addition to the interface for inter-model communication, a component-based design provides mechanisms to describe aggregation and composition.

5.3.2.1 Mapping of Aggregation

Aggregation is expressed in SMDL using references. Every model with at least one reference has to implement the `IAggregate` interface to give access to the references.

```
/// The aggregate interface is implemented by components with references
class IAggregate : public virtual IComponent
{
public:
      // Returns all references that the aggregate component requires.
      virtual const ReferenceCollection* GetReferences() = 0;

      // Returns a specific reference that the aggregate requires.
      virtual IReference* GetReference(String8 name) = 0;
};
```

For each reference, the model has to return a reference implementing the `IReference` interface.

```
/// The reference interface given access to components in a reference
class IReference : public virtual IObject
{
public:
      // Returns all referenced components of the reference.
      virtual const ComponentCollection* GetComponents() = 0;

      // Returns a named referenced component of the reference.
      virtual IComponent* GetComponent(String8 name) = 0;
};

/// Collection of references.
typedef std::vector<IReference*> ReferenceCollection;
```

The `IReference` interface gives access to the components aggregated by the model.

Each individual reference of a model is represented by a named instance of `IReference`.

Example:

```
class MyModel : public virtual Smp::IModel
{
public:
    [Skip implementation of IModel here]

    /// Reference to power loads of power distribution
    Smp::IReference* Powerloads;
};
```

Note: This definition only defines how other components can access a reference. It does not mandate how such a reference is implemented, as the specification is pure abstract. A supporting C++ MDK could provide an implementation that can be used to implement typed, managed references, i.e. references that allow adding new components via `AddComponent()`, and that check multiplicity and container types.

 © EUROPEAN SPACE AGENCY, 2005

5.3.2.2 Mapping of Composition

Composition is expressed in SMDL using containers. Every model with at least one container has to implement the `IComposite` interface to give access to its containers.

```
/// The composite interface is implemented by components with containers
class IComposite : public virtual IComponent
{
public:
      /// Returns all containers that the composite provides.
      virtual const ContainerCollection* GetContainers() = 0;

      /// Returns a specific container that the composite provides.
      virtual IContainer* GetContainer(String8 name) = 0;
};
```

For each container, the model has to return a container implementing the `IContainer` interface.

```
/// The container interface gives access to components in a container
class IContainer : public virtual IObject
{
public:
      /// Returns all contained components of the container.
      virtual const ComponentCollection* GetComponents() = 0;

      /// Returns a named contained component of the container.
      virtual IComponent* GetComponent(String8 name) = 0;
};

/// Collection of containers.
typedef std::vector<IContainer*> ContainerCollection;
```

The `IContainer` interface gives access to the components composed by the model.

Each individual container of a model is represented by a named instance of `IContainer`.

Example:

```
class MyModel : public virtual Smp::IModel
{
public:
    [Skip implementation of IModel here]

    /// Container of payloads of a satellite
    Smp::IContainer* Payloads;
};
```

Note: This definition only defines how other components can access a container. It does not mandate how such a container is implemented, as the specification is pure abstract. A supporting C++ MDK could provide an implementation that can be used to implement typed, managed containers, i.e. containers that allow adding new components via `AddComponent()`, and that check multiplicity and container types.

5.4 Event-based Design

An event-based design can be used in combination with a class-based design, but as well on top of an interface-based or component-based design. In an interface-based design, a consumer of an interface actively calls a provider of the same interface. Before it can do this, it has to get hold of the provider. If it does not know about the existence of a provider, it will neither get hold of it (via the interface), nor call its operations. In an event-based design, roles change: The provider of the event (the event source) calls all consumers that are subscribed to the event (the event sinks). While interfaces typically define a set of operations, an event is always mapped to a single function.

A model that is part of an event-based design may provide events (event sources) as well as consume events (event sinks). As an event defines a function with a typed parameter (called the event argument), an event type is needed for both event sources and event sinks. Only sources and sinks of matching event types can get connected.

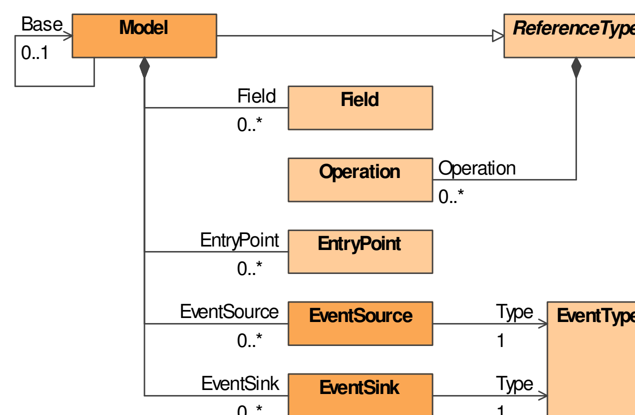

Figure 5-4: UML Class Diagram: Event-based model

5.4.1 Design elements

5.4.1.1 Event Types

An event type defines name and event argument of an event. The event argument needs to be a primitive type, i.e. a type that is fully defined by its value. Only these types can be passed as arguments between event source and event sinks without passing memory addresses.

5.4.1.2 Event Sources

An event source is a named feature of a model defined by an event type. In addition to this type, it can be specified whether the event source supports multiple sinks connected to it (multi-cast event, which is the default), or only a single event sink. An event source has a function to trigger it, i.e. to emit an event, passing an event argument as specified by the event type. In turn, the event source will call all event sinks currently connected to it, passing them a reference to the sending component as well as the event argument.

5.4.1.3 Event Sinks

An event sink is a named feature of a model defined by an event type. It can be connected to any number of event sources of the same event type. Whenever one of these event sources is triggered, it will call the event sink, passing it a reference to the model (the sender of the event) as well as an event argument.

 © EUROPEAN SPACE AGENCY, 2005

5.4.2 Mapping to C++

The mapping of events to C++ makes use of interfaces, as C++ has no native concept for events.

5.4.2.1 Mapping of Event Types

An event type is mapped to the type of the event argument of the event sink method in C++.

5.4.2.2 Mapping of Event Sources

An event source maps to an interface with `Subscribe()` and `Unsubscribe()` methods.

```
class IEventSource : public virtual IObject
{
public:
    /// Subscribe to the event source, i.e. request notifications.
    virtual void Subscribe(IEventSink* eventSink) = 0;

    /// Event unsubscription.
    virtual void Unsubscribe(IEventSink* eventSink) = 0;
};

/// Collection of event sources.
typedef std::vector<IEventSource*> EventSourceCollection;
```

Each individual event source of a model is represented by a named instance of `IEventSource`.

Example:

```
class MyModel : public virtual Smp::IModel
{
public:
    [Skip implementation of IModel here]

    /// Event source for changes in epoch time
    Smp::IEventSource* EpochChanged;
};
```

Note: This definition only defines how other components can access an event source. It does not mandate how such an event source is implemented, as the specification is pure abstract. A supporting C++ MDK could provide an implementation that can be used to implement typed event sources, i.e. event sources that allow subscribing event sinks via `Subscribe()`, unsubscribing via `Unsubscribe()`, and that check the event types of the event sinks. Further, the implementation could provide an `Emit` method that allows the parent model (i.e. the model owning the event source) to notify all event sinks.

5.4.2.3 Mapping of Event Sinks

An event sink maps to an interface with a `Notify` method.

```
class IEventSink : public virtual IObject
{
public:
    /// Event notification.
    virtual void Notify(IComponent* sender, AnySimple arg) = 0;
};

/// Collection of event sinks.
typedef std::vector<IEventSink*> EventSinkCollection;
```

Each individual event sink of a model is represented by a named instance of `IEventSink`.

Example:

```
class MyModel : public virtual Smp::IModel
{
public:
    [Skip implementation of IModel here]

    /// Event sink for changes in epoch time
    Smp::IEventSink* OnEpochChanged;
};
```

Note: This definition only defines how other components can access an event sink. It does not mandate how such an event sink is implemented, as the specification is pure abstract. A supporting C++ MDK could provide an implementation that can be used to implement typed event sinks, i.e. event sinks that can be connected to event sources via `Subscribe()`, and disconnected via `Unsubscribe()`. In the constructor, a private method could be passed to the event sink that is called on `Notify`.

5.5 Dataflow-based Design

Dataflow-based design can be implemented on top of a class-based design. It adds two attributes to the fields of a model, namely the input flag and the output flag. These flags identify input and output fields of a model.

In contradiction to the other design approaches introduced here, a model does not actively perform the flow of data in a dataflow-based design. Rather, data is transferred by (an) external mediator(s) based on information about output fields, input fields, and field links between them.

SMDL provides the mechanisms to tag fields of models in a catalogue as input and/or output fields, and to define dataflow links between the model instances in an assembly. However, this information is not actively evaluated by the model implementation, as the model relies on the fact that it is fed with input data.

5.5.1 Additional Design elements

In addition to the elements introduced for the different design approaches, SMDL defines some basic concepts applicable to all of the design approaches: Namespaces as a grouping mechanism, and user-defined types for float, integers, enumerations, arrays, strings, structures, and classes.

5.5.2 Design elements

5.5.2.1 Namespace

A Namespace is a grouping mechanism. It allows putting some structure into a model library.

5.5.2.2 Types

Fields, operations and properties are typed features within a catalogue. While SMDL defines some base types, if provides mechanisms to define new value types.

5.5.2.2.1 Float

Float is a mechanism to define double-precision floating-point types with additional range and unit information.

 © EUROPEAN SPACE AGENCY, 2005

5.5.2.2.2 Integer

Integer is a mechanism to define 32 bit signed integer types with additional range information.

5.5.2.2.3 Enumeration

An Enumeration allows expressing an integer number using string literals. It consists of a collection of name-value pairs, called the enumeration literals.

5.5.2.2.4 Array

An Array is a fixed-size vector of elements of one type.

5.5.2.2.5 String

A String is a fixed-size array of characters, with a convenient way of handling string values.

5.5.2.2.6 Structure

A Structure is a structured, user-defined type with named fields.

5.5.2.2.7 Class

A Class is a structured, user-defined type with named fields, operations, and an optional base class.

5.5.3 Mapping to C++

5.5.3.1 Mapping of Namespaces

A `Namespace` is mapped to an ISO/ANSI C++ `namespace`. As namespaces may contain nested namespaces, a C++ `namespace` may contain further namespaces.

Example:

```
namespace MyNamespace
{
	// Define nested namespaces and types
}
```

5.5.3.2 Mapping of Types

The user-defined types are mapped to C++ types as well.

5.5.3.2.1 Mapping of Float

A `Float` is mapped to either `Smp::Float32` or `Smp::Float64`. No range checking or unit conversion is performed at run-time.

Example:

```
typedef Smp::Float64 MyFloat;
```

5.5.3.2.2 Mapping of Integer

An `Integer` is mapped to any of the available integer types (except for `Smp::UInt64`), by default to `Smp::Int32`. No range checking is performed at run-time.

© EUROPEAN SPACE AGENCY, 2005

Example:

```
typedef Smp::Int32 MyInteger;
```

5.5.3.2.3 Mapping of Enumeration

An `Enumeration` is mapped as an ISO/ANSI C++ `enum` type. Each `EnumerationLiteral` is mapped as C++ `enum` literal with value assignment.

Example:

```
enum MyEnumeration
{
	MyLiteral1 = 0,
	MyLiteral2 = 1,
	...
};
```

5.5.3.2.4 Mapping of Array

An `Array` type is mapped as an ISO/ANSI C++ `struct` with an internal array. This is needed in order to allow using array types as return types of operations and properties.

Example:

```
struct MyArray
{
	Smp::Float64 internalArray[3];
};
```

5.5.3.2.5 Mapping of String

A `String` type is mapped as an ISO/ANSI C++ `struct` with an internal array. This is needed in order to allow using string types as return types of operations and properties. Further, the size of the array is by one bigger than the maximum number of characters, to allow for a null-terminated string.

Example:

```
struct MyString
{
	Smp::Char8 internalString[21];
};
```

5.5.3.2.6 Mapping of Structure

A `Structure` is mapped as an ISO/ANSI C++ `struct` with fields.

Example:

```
struct MyStructure
{
	Smp::Float64 MyFirstField;
	Smp::Int32   MySecondField;
};
```

 © EUROPEAN SPACE AGENCY, 2005

5.5.3.2.7 Mapping of Class

A `Class` type is mapped to a `class` in ISO/ANSI C++. Although a `Model` is mapped to a `class` as well, its semantics is very different: Models are components in the SMP2 component model, they may have entry points, and they can be enhanced by various elements for component-based or event-based design. Classes do exist as well, but they are not formally exposed as nodes of the model hierarchy (though their fields are typically published to the simulation environment).

Example:

```
class MyClass
{
	Smp::Float64 MyFirstField;
	Smp::Int32   MySecondField;
public:
	Smp::Bool MyOperation(Smp::Float64 myParameter);
};
```

This Page is Intentionally left Blank

© EUROPEAN SPACE AGENCY, 2005

6. MODEL INTEGRATION

When a library of model implementations is available, a configuration of model instances has to be created. This process is called model integration, and explained in this section.

Two different approaches exist for model integration. The first one uses source code to build and initialise a model hierarchy, the second one makes use of external information (typically an SMDL assembly document) to build and initialise the models. Both are explained in this section.

6.1 Model Integration using Source Code

This approach needs very little support from models, i.e. does (almost) not rely on optional interfaces. An example of a model supporting manual integration is the `Counter` model introduced in section 3. Instances of it can be added into the hierarchy with a given name and parent. As the counter model does not use any of the optional component mechanisms (aggregation, i.e. references to other models, composition, i.e. containers with child components, or events), it does not need to implement any additional interfaces.

6.1.1 Adding a Root Model to the Simulator

The following example code snippets show how a simulation with an instance of this counter model can be created.

Error handling is omitted for clarity.

```
#include "Counter.h"
#include "Smp/ISimulator.h"

// Create Simulation using Counter
// ===========================================================================
void CreateCounter(Smp::ISimulator* simulator)
{
    // 1. Create instance of model with simulator as parent
    Counter* counterModel = new Counter("Counter", simulator);

    // 2. Add counter to models container
    simulator->AddModel(counterModel);

    // 3. Publish and Configure model
    simulator->Publish();
    simulator->Configure();

    // 4. Switch from Building to Connecting state
    simulator->Connect();
}
```

All that is done in this code is creating a named instance of the model and adding it as a root model of the simulator. The simulation environment is responsible for calling the `Publish()`, `Configure()`, and `Connect()` methods of the model. The `Counter` model will add its entry points to the scheduler and event manager itself.

© EUROPEAN SPACE AGENCY, 2005

6.1.2 Building a Hierarchy of Model Instances

The next example demonstrates how a hierarchy of model instances can be created. The code shows a thermal subsystem with two heater child models.

```
// Create Simulation using Thermal and Heater
// ================================================================
void CreateThermal(Smp::ISimulator* simulator)
{
    // 1. Create thermal subsystem
    Thermal* thermal = new Thermal("Thermal", "Thermal subsystem",
        simulator);

    // 2. Create two heaters of thermal subsystem
    Heater* heater1 = new Heater("Heater1", "First heater", thermal);
    Heater* heater2 = new Heater("Heater2", "Second heater", thermal);

    // 3. Add heaters to their container
    thermal->Heaters->AddComponent(heater1);
    thermal->Heaters->AddComponent(heater2);

    // 4. Publish and Configure models
    simulator->Publish();
    simulator->Configure();

    // 5. Switch from Building to Connecting state
    simulator->Connect();
}
```

Note: A two-way link is created between the `thermal` subsystem and the `heater1` and `heater2` components: Firstly, the `thermal` container is passed as parent model to the constructor of the heater models:

```
heater1 = new Heater("Heater1", "First heater", thermal);
```

Secondly, the `heater1` model is explicitly added to the `Heaters` container:

```
thermal->Heaters->AddComponent(heater1);
```

These two lines correspond to the `GetParent()` mechanism of the `heater1` model, and the `GetComponents()` mechanism of the `Heaters` container. They allow navigating upwards and downwards in the model hierarchy.

Note: In order to be able to create this two-way link, the `thermal` model has to allow adding a heater to it. The implementation shown here uses the recommended SMP2 mechanism of a managed container that supports `AddComponent()`. However, for this type of model integration, any other mechanism is valid as well: The power system could e.g. support a method `AddHeater()` which adds the model to its `Heaters` container using another mechanism.

6.1.3 Creating Interface Links between Model Instances

The third example enhances the second one by a power distribution model that is added as root model to the simulator. As the heater components consume power, they are connected to a reference of power loads of the power distribution model.

```
// Create Simulation using Thermal, Heater and PowerDistribution
// ================================================================
void CreateSimulator(Smp::ISimulator* simulator)
```


© EUROPEAN SPACE AGENCY, 2005

```
{
    // 1. Create thermal subsystem
    Thermal* thermal = new Thermal("Thermal", "Thermal subsystem",
        simulator);

    // 2. Create two heaters of thermal subsystem
    Heater* heater1 = new Heater("Heater1", "First heater", thermal);
    Heater* heater2 = new Heater("Heater2", "Second heater", thermal);

    // 3. Add heaters to their container
    thermal->Heaters->AddComponent(heater1);
    thermal->Heaters->AddComponent(heater2);

    // 4. Create instance of power distribution
    PowerDistribution* powerDistribution = new PowerDistribution(
        "PowerDistribution", "Power distribution", simulator);

    // 5. Include heaters as power loads
    powerDistribution->PowerLoads->AddComponent(heater1);
    powerDistribution->PowerLoads->AddComponent(heater2);

    // 6. Publish and Configure models
    simulator->Publish();
    simulator->Configure();

    // 7. Switch from Building to Connecting state
    simulator->Connect();
}
```

Each of the two heater instances is connected to the `PowerLoads` reference of the power distribution model, so the power distribution model can take their power consumption into account.

```
powerDistribution->Powerloads->AddComponent(heater1);
```

Note: In order to be able to create this interface link, the `powerDistribution` model has to allow adding a reference to a power load. The implementation shown here uses the recommended SMP2 mechanism of a managed reference that supports `AddComponent()`. However, for this type of model integration, any other mechanism is valid as well: The power distribution subsystem could e.g. support a method `LinkPowerLoad()`, which adds the model to its `PowerLoads` reference using another mechanism.

6.1.4 Creating Event Links between Model Instances

This final example demonstrates how links between event sources and event sinks of model instances can be established. It shows an on-board reference time model which emits a time synchronisation event, and a payload model which subscribes an event source to it.

```
// Create Simulation of On-Board Reference Time and Payload
// ========================================================================
void CreateSimulation(Smp::ISimulator* simulator)
{
    // 1. Create instances of models and add to simulator
    OBRT* obrt = new OBRT("OBRT", "On Board Reference Time", simulator);
    Payload* payload = new Payload("Payload", "Any Payload", simulator);

    // 2. Link instances together via event link
    obrt->TimeSync->Subscribe(payload->OnTimeSync);

    // 3. Publish and Configure models
```

```
    simulator->Publish();
    simulator->Configure();

    // 4. Switch from Building to Connecting state
    simulator->Connect();
}
```

The `OnTimeSync` event sink of the `payload` model is connected to the `TimeSync` event source of the `obrt` model.

```
obrt->TimeSync->Subscribe(payload->OnTimeSync)
```

Whenever the `obrt` model emits a `TimeSync` event, the `Notify` method of the payload's `OnTimeSync` event sink will be called.

All four examples in this section explicitly access fields of the model instances. Therefore, they need to be compiled including the header files of the involved models. While this is fully valid for scenarios with fairly static configuration, this may be a limitation when simulations need to be configured dynamically, i.e. when new models are frequently added, and the model hierarchy is often changed.

6.2 Model Integration using Assemblies

In addition to the default model integration technique explained above, SMP2 defines an optional dynamic model integration mechanism. The simulation environment and the models used within such a scenario need to support all interfaces and features as specified in section 2.9.2. Typically, such a dynamically configured simulation uses the metamodel concepts of catalogues, assemblies and schedules.

The sequence of steps during the `Creating` phase can be summarised as follows:

1. All available binary model implementations register their factories with the dynamic simulator.
2. An Assembly document is read into memory. Its information is used to do the following:
    a. Create model instances as specified in the Assembly document
    b. Create a hierarchy of these model instances as specified in the Assembly document
    c. Create interface links between model instances as specified in the Assembly document
    d. Create event links between model instances as specified in the Assembly document
    e. Define initial values of the fields of the model instances.
3. A Schedule document is read into memory. Its information is used to do the following:
    a. Create tasks in the scheduler service as specified in the Schedule document
    b. Create events in the scheduler service as specified in the Schedule document.
4. The `Publish()` method of the dynamic simulator is called to start model initialisation.

Each of these steps is explained in more detail below.

6.2.1 Model Factories

When loading information about model instances from an assembly document, the information available for each model instance include its reference to its model type in a catalogue (called its specification), and its model implementation. While the specification identifier is important to know when creating and editing

 © EUROPEAN SPACE AGENCY, 2005

an assembly (as the model types define the structure), only the implementation identifier is needed when creating a run-time configuration of run-time model instances.

An implementation identifier is a Universally Unique Identifier (**UUID**) that uniquely identifies an implementation. However, it does neither say where this implementation "lives" (e.g. in which library or shared object), nor does it say how an instance of the implementation can be created. The second question is answered by a component implementing the `IDynamicSimulator` interface (i.e. the dynamic simulator), as its `CreateInstance()` method creates a run-time model instance for a given model implementation identifier:

```
IComponent* model = dynSim->CreateInstance(implUuid);
```

However, the dynamic simulator first needs to get told how it can create such an instance. This is done using its `RegisterFactory()` method, passing it a factory for the model implementation:

```
dynSim->RegisterFactory(modelFactory);
```

When asked for an instance of the model implementation, the dynamic simulator simply forwards this call to the corresponding factory:

```
IComponent* model = modelFactory->CreateInstance();
```

While all these methods are standardised by SMP2, it is not standardised how the factories are created before the `RegisterFactory()` method of the dynamic simulator is called. Furthermore, packaging of models and factories is not standardised. This is on purpose, as different implementations may have different requirements and limitations.

The SMP 2.0 C++ Mapping [AD-3, Section 5.5] provides a mapping of a `Package` for the C++ platform, which uses a well-defined entry point `Initialise()` in a DLL/DSO that is called to register all value types and factories within the DLL/DSO.

6.2.2 Building a Hierarchy of Model Instances

The assembly document specifies the following information for each model instance:

- Model instance name and description
- Parent component and container name
- Implementation identifier

To add a new model instance to the hierarchy using only this information and the dynamic simulator (i.e. a component implementing `IDynamicSimulator`), the following code could be used:

```
// Create model instance and insert it into the model hierarchy
// ========================================================================
void AddToContainer(Smp::String8 modelName,
                    Smp::String8 modelDescription,
                    Smp::IComposite* parent,
                    Smp::String8 containerName,
                    Smp::Uuid    implUuid,
                    Smp::IDynamicSimulator* simulator)
{
    using namespace Smp::Management;

    IManagedComponent* managedComponent;
    IManagedContainer* managedContainer;

    // 1. Create new instance using component factory via simulator
```

```
    Smp::IComponent* modelInstance= simulator->CreateInstance(implUuid);

    // 2. Set name, description, parent
    managedComponent = dynamic_cast<IManagedComponent*>(modelInstance);
    managedComponent->SetName(modelName);
    managedComponent->SetDescription(modelDescription);
    managedComponent->SetParent(parent);

    // 3. Get container of parent
    managedContainer = dynamic_cast<IManagedContainer*>
                              (parent->GetContainer(containerName));

    // 4. Add new instance to container of parent
    managedContainer->AddComponent(modelInstance);
}
```

This implementation only uses information available from XML, and does not need additional information about a specific model (i.e. does not include a header file of a model implementation). It relies on a model implementing `IManagedComponent` to set name, description, and parent, on the `IComposite` implementation of the `parent`, and on an `IManagedContainer` to add the component to.

6.2.3 Creating Interface Links between Model Instances

The assembly document specifies the following information for each interface link:

- Consumer model instance of the link (parent of link)
- Provider model instance
- Name of Reference to add provider to

To add an interface link to the named Reference using only this information, the following code could be used:

```
// Create interface link between provider and reference of consumer
// ======================================================================
void AddToReference(Smp::IComponent* provider,
                    Smp::IComponent* consumer,
                    Smp::String8 referenceName)
{
    using namespace Smp::Management;

    Smp::IAggregate* aggregate;
    Smp::IReference* reference;
    IManagedReference* managedReference;

    // 1. Get reference
    aggregate = dynamic_cast<Smp::IAggregate*>(consumer);
    reference = aggregate->GetReference(referenceName);

    // 2. Add provider to reference
    managedReference = dynamic_cast<IManagedReference*>(reference);
    managedReference->AddComponent(provider);
}
```

This implementation only uses information available from XML, and does not need additional information about a specific model (i.e. does not include a header file of a model implementation). It relies on the `IAggregate` implementation of the `consumer` to get the named reference, and on the implementation of the `IManagedReference` interface of this reference to add the component to it.

 © EUROPEAN SPACE AGENCY, 2005

6.2.4 Creating Event Links between Model Instances

The assembly document specifies the following information for each event link:

- Consumer model instance of the link (parent of link)
- Provider model instance
- Name of Event Source to add event sink to
- Name of Event Sink to add to event source

To add an event link between the two model instances using only this information, the following code could be used:

```
// Create event link between provider source and consumer sink
// ==================================================================
void AddToEventSource(Smp::IComponent* provider,
                      Smp::String8     eventSourceName,
                      Smp::IComponent* consumer,
                      Smp::String8     eventSinkName)
{
    using namespace Smp;
    using namespace Smp::Management;

    IEventProvider*  eventProvider;
    IEventConsumer*  eventConsumer;
    IEventSource*    eventSource;
    IEventSink*      eventSink;

    // 1. Get event source
    eventProvider = dynamic_cast<IEventProvider*>(provider);
    eventSource = eventProvider->GetEventSource(eventSourceName);

    // 2. Get event sink
    eventConsumer = dynamic_cast<IEventConsumer*>(consumer);
    eventSink = eventConsumer->GetEventSink(eventSinkName);

    // 3. Subscribe event sink to event source
    eventSource->Subscribe(eventSink);
}
```

This implementation only uses information available from XML, and does not need additional information about a specific model (i.e. does not include a header file of a model implementation). It relies on the `IEventProvider` implementation of the `provider`, and on the `IEventConsumer` implementation of the `consumer`.

6.2.5 Scheduling Entry Points of the Model Instances

When a complete model hierarchy has been created and linked together based on the information in an assembly document, a schedule document can be loaded to define its scheduling.

6.2.5.1 Create a Task of Entry Points

The schedule document specifies the following information for each task:

- Name and description of task
- Collection of entry points of the task, with model instance and entry point name for each

To create a new task using only this information, an implementation of the `ITask` interface is required. This is e.g. provided by the C++ Model Development Kit.

To add an entry point to this new task using only this information, the following code could be used:

```
// Add entry point to task
// ==========================================================================
void AddEntryPoint(Smp::Services::ITask* task,
                   Smp::IComponent* publisher,
                   Smp::String8 name)
{
    using namespace Smp;
    using namespace Smp::Management;

    IEntryPointPublisher* epPublisher;

    // 1. Get entry point
    epPublisher = dynamic_cast<IEntryPointPublisher*>(publisher);
    const IEntryPoint* entryPoint = epPublisher->GetEntryPoint(name);

    // 2. Add entry point to task
    task->AddEntryPoint(entryPoint);
}
```

This implementation only uses information available from XML, and does not need additional information about a specific model (i.e. does not include a header file of a model implementation). It relies on the `IEntryPointPublisher` implementation of the `publisher` to get the entry point by name.

6.2.5.2 Schedule a Task

The schedule document specifies the following information for each event:

- Event name and description
- Delta time, cycle time and count
- Task to schedule

This translates into a direct call into the scheduler service, e.g. for a `SimulationTimeEvent`:

```
scheduler->AddSimulationTimeEvent(task, deltaTime, cycleTime, count);
```


© EUROPEAN SPACE AGENCY, 2005

7. MODEL EXAMPLES

This section contains model examples for the different design approaches. For each example, a UML design, the Catalogue Elements, the Catalogue file and the C++ model definition files (header files) are shown. Where applicable, Assembly Elements and the Assembly file are shown as well.

7.1 Class-based Example

This first example continues the Counter example started in section 3 (Getting Started). The complete source code has been given already in section 3.7 (The complete model), but here, the model is expressed using a Catalogue.

The class-based example model provides all elements of a class-based model: It has a field `counter` for its state, with an initial value of 0, a read-only property `Counter` to gain read-only access to the state, and two entry points `Count` and `Reset`.

7.1.1 UML Design

In UML, such a class could be defined as shown in Figure 7-1. Note that UML neither supports properties nor entry points, so both are expressed as operations.

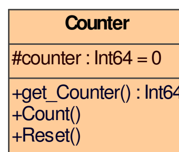

Figure 7-1: UML Class Diagram: Design for Class-based Example

7.1.2 Catalogue Elements

In a Catalogue editor, the features of the Counter model may be displayed as shown in Figure 7-2.

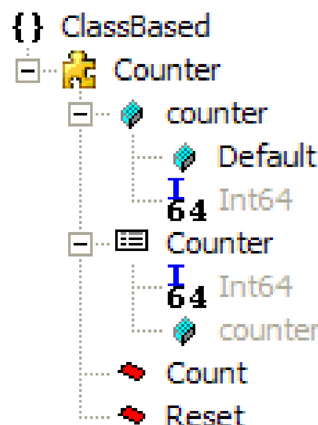

Figure 7-2: Catalogue Elements for Class-based Example

The `Counter` model is defined within the namespace `ClassBased` in a catalogue called `Handbook`.

The `counter` field as well as the `Counter` property link to their type `Int64`. In addition, the `Counter` property links to its attached field `counter`, while the `counter` field specifies its `Default` value.

Note that such a display does only show some information. It is e.g. not visible that the field is protected, or that the Counter property is read-only. Such additional information needs to be shown in additional dialogs.

7.1.3 Catalogue File

The corresponding XML specification for the namespace and the model is shown in Figure 7-3.

```
<Namespace Id="ClassBased" Name="ClassBased">
  <Type xsi:type="Catalogue:Model" Id="Counter" Name="Counter">
    <Uuid>6bf35670-29f4-4a1e-9a5e-deb3638c206f</Uuid>
    <Property Id="Counter.Counter" Name="Counter">
      <Type xlink:title="Int64" xlink:href="http://www.esa.int/2005/10/Smp#Int64" />
      <AttachedField xlink:title="counter" xlink:href="#Counter.counter" />
    </Property>
    <EntryPoint Id="Counter.Count" Name="Count" />
    <EntryPoint Id="Counter.Reset" Name="Reset" />
    <Field Id="Counter.counter" Name="counter" Visibility="protected">
      <Type xlink:title="Int64" xlink:href="http://www.esa.int/2005/10/Smp#Int64" />
      <Default xsi:type="Types:SimpleValue">
        <Value xsi:type="xsd:long">0</Value>
      </Default>
    </Field>
  </Type>
</Namespace>
```

Figure 7-3: Catalogue File for Class-based Example

7.1.4 Model Definition File

A compatible C++ definition of the `Counter` model is shown below.

```
// Definition of Counter
namespace ClassBased
{
    class Counter : public virtual Smp::IModel
    {
    private:
        char* name;                          ///< Name of model.
        char* description;                   ///< Description of model.
        Smp::IComposite* parent;             ///< Parent component.

    protected:
        Smp::Int64 counter;                  ///< Counter value.

    public:
        virtual void Count(void);            ///< Increment counter.
        virtual void Reset(void);            ///< Reset counter.
        virtual Smp::Int64 get_Counter(void); ///< Get counter.

        /// Public constructor with name, description and parent.
        Counter(Smp::String8 name, Smp::String8 description, Smp::IComposite* parent);

        // IObject, IComponent and IModel methods
        virtual Smp::String8 GetName()        const;
        virtual Smp::String8 GetDescription() const;
        virtual Smp::IComposite* GetParent()  const;
        virtual Smp::ModelStateKind GetState() const;
        virtual void Publish(Smp::IPublication* receiver)
        throw (Smp::IModel::InvalidModelState);
        virtual void Configure(Smp::Services::ILogger* logger)
        throw (Smp::IModel::InvalidModelState);
        virtual void Connect(Smp::ISimulator* simulator)
        throw (Smp::IModel::InvalidModelState);
    };
}
```


© EUROPEAN SPACE AGENCY, 2005

7.2 Interface-based Example

This second example shows how an interface can be used to break an explicit dependency between models. Introduced are two models (part of a complete satellite system) that communicate via an interface. The two models involved are a power distribution model that takes care of all power loads, and a heater component that is one specific power load. However, the heater component is not a sub-component of the power system, but rather part of the thermal system. For simplicity, this example defines a power subsystem that holds power distribution and power loads in two containers.

7.2.1 UML Design

In UML, such a relationship is called an Aggregation. It is shown in Figure 7-4.

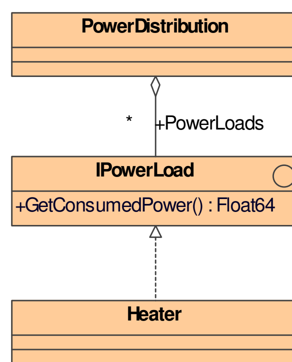

Figure 7-4: UML Class Diagram: Design for Interface-based Example

7.2.2 Catalogue Elements

In a Catalogue editor, the features of the involved models and interfaces may be displayed as shown in Figure 7-5.

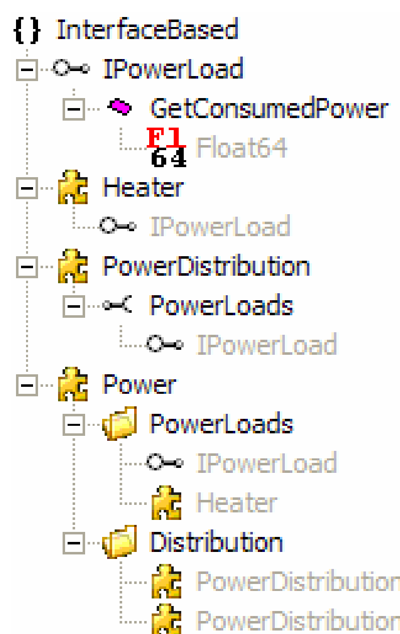

Figure 7-5: Catalogue Elements for Interface-based Example

The Heater model **provides** the IPowerLoad interface.

The PowerDistribution model **consumes** the interface via its reference PowerLoads. For the reference, a multiplicity of "*" is specified, which means that any number of components can be referenced. This is often expressed as "0..*" as well.

The IPowerLoad interface is defined as an interface with an operation GetConsumedPower to get the power consumption of a component.

Finally, the Power model provides two containers, one that can hold the PowerDistribution model, and one that can hold models implementing IPowerLoad (although in a real spacecraft design, the Heater would most likely not be part of the Power subsystem). For demonstration purposes, the PowerLoads container has the Heater model as its default model.

7.2.3 Catalogue File

The corresponding XML specification for the namespace and the models is shown in Figure 7-6.

```
<Namespace Id="InterfaceBased" Name="InterfaceBased">
  <Type xsi:type="Catalogue:Interface" Id="IPowerLoad" Name="IPowerLoad">
    <Uuid>ee7686d0-dd5b-435f-bd39-af9ef7f2b0fe</Uuid>
    <Operation Id="IPowerLoad.GetConsumedPower" Name="GetConsumedPower">
    <Type xlink:title="Float64" xlink:href="http://www.esa.int/2005/10/Smp#Float64" />
    </Operation>
  </Type>
  <Type xsi:type="Catalogue:Model" Id="Heater" Name="Heater">
    <Uuid>9559b5e9-aaea-42c7-8179-01a88d24c9bf</Uuid>
    <Interface xlink:title="IPowerLoad" xlink:href="#IPowerLoad" />
  </Type>
  <Type xsi:type="Catalogue:Model" Id="PowerDistribution" Name="PowerDistribution">
    <Uuid>e208d553-d30c-4cc7-b624-9b7cb6770440</Uuid>
    <Reference Id="PowerDistribution.PowerLoads" Name="PowerLoads" Lower="0"
                                                                    Upper="-1">
      <Interface xlink:title="IPowerLoad" xlink:href="#IPowerLoad" />
    </Reference>
  </Type>
  <Type xsi:type="Catalogue:Model" Id="Power" Name="Power">
    <Uuid>e208d553-d30c-4cc7-b624-9b7cb677043f</Uuid>
    <Container Id="Power.PowerLoads" Name="PowerLoads" Lower="0" Upper="-1">
      <Type xlink:title="IPowerLoad" xlink:href="#IPowerLoad" />
      <DefaultModel xlink:title="Heater" xlink:href="#Heater" />
    </Container>
    <Container Id="Power.Distribution" Name="Distribution">
      <Type xlink:title="PowerDistribution" xlink:href="#PowerDistribution" />
      <DefaultModel xlink:title="PowerDistribution" xlink:href="#PowerDistribution" />
    </Container>
  </Type>
</Namespace>
```

Figure 7-6: Catalogue File for Interface-based Example

7.2.4 Model Definition Files

A compatible C++ definition of the IPowerLoad interface is shown below.

```
#include "Smp/SimpleTypes.h"

class IPowerLoad
{
public:
    virtual ~IPowerLoad() { }
    virtual Smp::Float64 GetConsumedPower() = 0;
};
```

 © EUROPEAN SPACE AGENCY, 2005

A compatible C++ definition of the `Heater` model is shown below.

```
#include "IPowerLoad.h"
#include "Smp/IModel.h"

namespace InterfaceBased
{
    class Heater : public virtual Smp::IModel, public virtual IPowerLoad
    {
    private:
        char* name;                         ///< Name of model.
        char* description;                  ///< Description of model.
        Smp::IComposite* parent;            ///< Parent component.

    public:
        // IPowerLoad methods
        Smp::Float64 GetConsumedPower();

        /// Public constructor with name, description and parent.
        Heater(Smp::String8 name, Smp::String8 description,
               Smp::IComposite* parent);

        // IObject, IComponent and IModel methods
        virtual Smp::String8 GetName()        const;
        virtual Smp::String8 GetDescription() const;
        virtual Smp::IComposite* GetParent()  const;
        virtual Smp::ModelStateKind GetState() const;
        virtual void Publish(Smp::IPublication* receiver)
        throw (Smp::IModel::InvalidModelState);
        virtual void Configure(Smp::Services::ILogger* logger)
        throw (Smp::IModel::InvalidModelState);
        virtual void Connect(Smp::ISimulator* simulator)
        throw (Smp::IModel::InvalidModelState);
    };
}
```

A compatible C++ definition of the `PowerDistribution` model is shown below.

```
#include "Smp/IModel.h"
#include "Smp/IReference.h"

namespace InterfaceBased
{
    class PowerDistribution : public virtual Smp::IModel
    {
    private:
        char* name;                         ///< Name of model.
        char* description;                  ///< Description of model.
        Smp::IComposite* parent;            ///< Parent component.

    public:
        // PowerLoads reference
        Smp::IReference* PowerLoads;

        /// Public constructor with name, description and parent.
        PowerDistribution(Smp::String8 name, Smp::String8 description,
                          Smp::IComposite* parent);

        // IObject, IComponent and IModel methods
        virtual Smp::String8 GetName()        const;
        virtual Smp::String8 GetDescription() const;
        virtual Smp::IComposite* GetParent()  const;
        virtual Smp::ModelStateKind GetState() const;
        virtual void Publish(Smp::IPublication* receiver)
        throw (Smp::IModel::InvalidModelState);
        virtual void Configure(Smp::Services::ILogger* logger)
        throw (Smp::IModel::InvalidModelState);
        virtual void Connect(Smp::ISimulator* simulator)
        throw (Smp::IModel::InvalidModelState);
    };
}
```

7.2.5 Assembly Elements

When using such a design in an Assembly, several instances of the Heater model can be created and connected to the same instance of the power distribution model, as indicated in Figure 7-8. An Assembly editor may show the same information as a hierarchy as indicated in Figure 7-7. It can be seen that both `Heater1` and `Heater2` link to the `Heater` container, while `PowerDistribution` links to the `Distribution` container. Every model instance links back to its type in the catalogue. The two interface links both link to the same reference (namely `PowerLoads`), but to different providers (`Heater1` and `Heater2`).

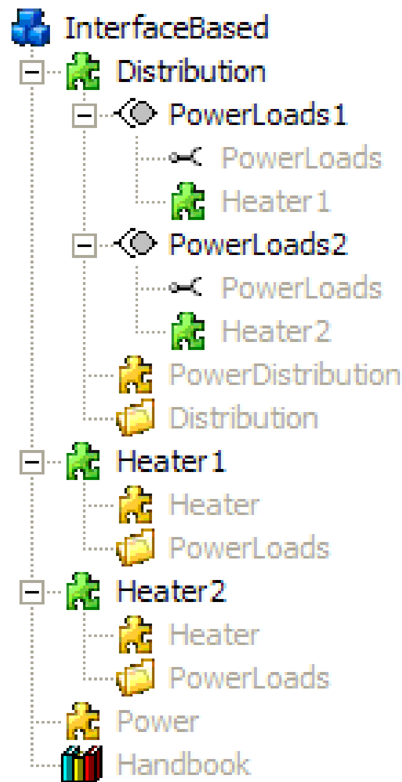

Figure 7-7: Assembly Elements for Interface-based Example

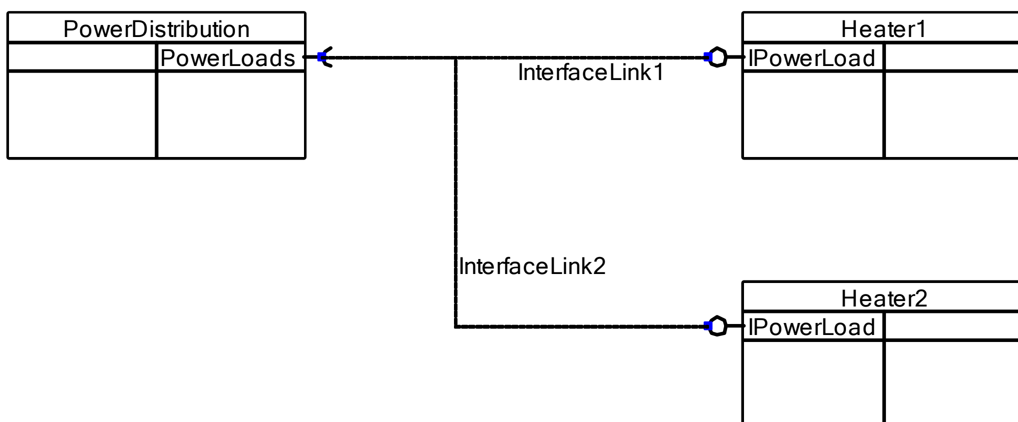

Figure 7-8: Example Configuration using Interface Links

© EUROPEAN SPACE AGENCY, 2005

Note that this diagram does not show the model types of the catalogue ("class diagram"), but the model instances in an assembly ("implementation diagram").

7.2.6 Assembly File

The corresponding XML specification for the model integration is shown in Figure 7-9.

```
<?xml version="1.0" encoding="utf-8"?>
<Assembly:Assembly Name="InterfaceBased" Date="2005-02-28T12:00:00+01:00"
                     Id="InterfaceBased" Creator="pfritzen" Version="1.1"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  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:Assembly="http://www.esa.int/2005/10/Smdl/Assembly">
  <Model xlink:title="Power" xlink:href="Handbook.cat#Power" />
  <Implementation>00000000-0000-0000-0000-000000000000</Implementation>
  <ModelInstance Id="Distribution" Name="Distribution">
  <Model xlink:title="PowerDistribution" xlink:href="Handbook.cat#PowerDistribution"/>
    <Implementation>00000000-0000-0000-0000-000000000000</Implementation>
    <Link xsi:type="Assembly:InterfaceLink" Id="Distribution.PowerLoads1"
                                      Name="PowerLoads1">
      <Reference xlink:title="PowerLoads"
                 xlink:href="Handbook.cat#PowerDistribution.PowerLoads" />
      <Provider xlink:title="Heater1" xlink:href="#Heater1" />
    </Link>
    <Link xsi:type="Assembly:InterfaceLink" Id="Distribution.PowerLoads2"
                                      Name="PowerLoads2">
      <Reference xlink:title="PowerLoads"
                 xlink:href="Handbook.cat#PowerDistribution.PowerLoads" />
      <Provider xlink:title="Heater2" xlink:href="#Heater2" />
    </Link>
    <Container xlink:title="Distribution"
               xlink:href="Handbook.cat#Power.Distribution" />
  </ModelInstance>
  <ModelInstance Id="Heater1" Name="Heater1">
    <Model xlink:title="Heater" xlink:href="Handbook.cat#Heater" />
    <Implementation>00000000-0000-0000-0000-000000000000</Implementation>
    <Container xlink:title="PowerLoads" xlink:href="Handbook.cat#Power.PowerLoads" />
  </ModelInstance>
  <ModelInstance Id="Heater2" Name="Heater2">
    <Model xlink:title="Heater" xlink:href="Handbook.cat#Heater" />
    <Implementation>00000000-0000-0000-0000-000000000000</Implementation>
    <Container xlink:title="PowerLoads" xlink:href="Handbook.cat#Power.PowerLoads" />
  </ModelInstance>
</Assembly:Assembly>
```

Figure 7-9: Assembly File for Interface-based Example

Note that no implementation has been specified for the model instances.

7.3 Component-based Example

Taking interfaces further leads to a component-based design. Here, the interface-based example is expanded to include a thermal system that contains heaters. Such containment could be expressed via interfaces as well, but this example creates a direct dependency between the thermal model and the heater model. Although this is less flexible, such a design is sufficient when the Thermal model will either never be replaced, or only be replaced by models derived from it via implementation inheritance.

7.3.1 UML Design

In UML, a containment relationship is called Composition. It is shown in Figure 7-10.

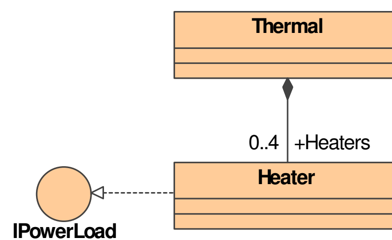

Figure 7-10: UML Class Diagram: Design for Component-based Example

Here, the number of heater components has been limited to at most 4 heaters. Note that it is shown that the Heater implements the `IPowerLoad` interface (which is irrelevant for this diagram), but using another UML shape for an interface.

7.3.2 Catalogue Elements

In a Catalogue editor, the features of the Thermal model may be displayed as shown in Figure 7-11. The `Heater` model from the Interface-based example is used both as type and as default model of the `Heaters` container.

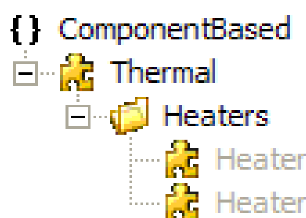

Figure 7-11: Catalogue Elements for Component-based Example

© EUROPEAN SPACE AGENCY, 2005

7.3.3 Catalogue File

The corresponding XML specification for the namespace and the model is shown in Figure 7-12.

```
<Namespace Id="ComponentBased" Name="ComponentBased">
  <Type xsi:type="Catalogue:Model" Id="Thermal" Name="Thermal">
    <Uuid>64f1ad7e-c745-473d-8d48-ad79daaf29af</Uuid>
    <Container Id="Thermal.Heaters" Name="Heaters" Lower="0" Upper="4">
      <Type xlink:title="Heater" xlink:href="#Heater" />
      <DefaultModel xlink:title="Heater" xlink:href="#Heater" />
    </Container>
  </Type>
</Namespace>
```

Figure 7-12: Catalogue File for Component-based Example

7.3.4 Model Definition Files

A compatible C++ definition of the `Thermal` model is shown below.

```
#include "Smp/IModel.h"
#include "Smp/IComposite.h"
#include "Smp/Management/IManagedContainer.h"

namespace InterfaceBased
{
    class Thermal : public virtual Smp::IModel,
        public virtual Smp::IComposite
    {
    private:
        char* name;                              ///< Name of model.
        char* description;                       ///< Description of model.
        Smp::IComposite* parent;                 ///< Parent component.
        Smp::ContainerCollection* containers;    ///< Container collection.

    public:
        // PowerLoads reference
        Smp::Management::IManagedContainer* Heaters;

        /// Public constructor with name, description and parent.
        Thermal(Smp::String8 name, Smp::String8 description,
                Smp::IComposite* parent);

        // IObject, IComponent and IModel methods
        virtual Smp::String8 GetName()        const;
        virtual Smp::String8 GetDescription() const;
        virtual Smp::IComposite* GetParent()  const;
        virtual Smp::ModelStateKind GetState() const;
        virtual void Publish(Smp::IPublication* receiver)
        throw (Smp::IModel::InvalidModelState);
        virtual void Configure(Smp::Services::ILogger* logger)
        throw (Smp::IModel::InvalidModelState);
        virtual void Connect(Smp::ISimulator* simulator)
        throw (Smp::IModel::InvalidModelState);

        // IComposite methods
        const Smp::ContainerCollection* GetContainers() const;
        Smp::IContainer* GetContainer(Smp::String8 name) const;
    };
}
```

7.3.5 Assembly Elements

An Assembly integrating a thermal model with two heaters may be shown as a hierarchy by an Assembly editor as indicated in Figure 7-13. Both model instances reference the `Heaters` container and the `Heater` model.

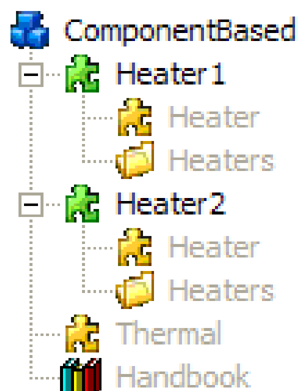

Figure 7-13: Assembly Elements for Component-based Example

7.3.6 Assembly File

The corresponding XML specification for the model integration is shown in Figure 7-14.

```
<?xml version="1.0" encoding="utf-8"?>
<Assembly:Assembly Name="ComponentBased" Date="2005-02-28T12:00:00+01:00"
                     Id="ComponentBased" Creator="pfritzen" Version="1.1"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  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:Assembly="http://www.esa.int/2005/10/Smdl/Assembly">
  <Model xlink:title="Thermal" xlink:href="Handbook.cat#Thermal" />
  <Implementation>00000000-0000-0000-0000-000000000000</Implementation>
  <ModelInstance Id="Heater1" Name="Heater1">
    <Model xlink:title="Heater" xlink:href="Handbook.cat#Heater" />
    <Implementation>00000000-0000-0000-0000-000000000000</Implementation>
    <Container xlink:title="Heaters" xlink:href="Handbook.cat#Thermal.Heaters" />
  </ModelInstance>
  <ModelInstance Id="Heater2" Name="Heater2">
    <Model xlink:title="Heater" xlink:href="Handbook.cat#Heater" />
    <Implementation>00000000-0000-0000-0000-000000000000</Implementation>
    <Container xlink:title="Heaters" xlink:href="Handbook.cat#Thermal.Heaters" />
  </ModelInstance>
</Assembly:Assembly>
```

Figure 7-14: Assembly File for Component-based Example

Note that no implementation has been specified for the model instances.

7.4 Event-based Example

Finally, an event-based example is presented. It defines an event type `TimeSyncEvent` that passes the current time (as a `DateTime`). The On-Board Reference Time (OBRT) Model provides an event source that can notify an unlimited number of event sinks of changes of the reference time. As an example, a `Payload` model is implemented with an event sink of the same event type.

 © EUROPEAN SPACE AGENCY, 2005

7.4.1 UML Design

UML does not provide native mechanisms to express event-based designs. However, by defining the event type as an interface stereotyped «event», a usage dependency stereotyped «emit» can express an event source, while a realization stereotyped «receive» can express an event sink[13].

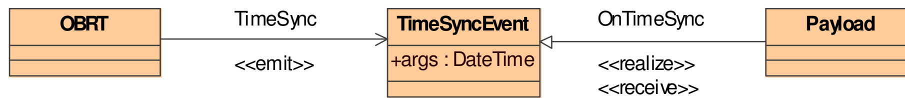

Figure 7-15: UML Class Diagram: Design for Event-based Example

7.4.2 Catalogue Elements

In a Catalogue editor, the features of the event type and both models may be displayed as shown in Figure 7-16.

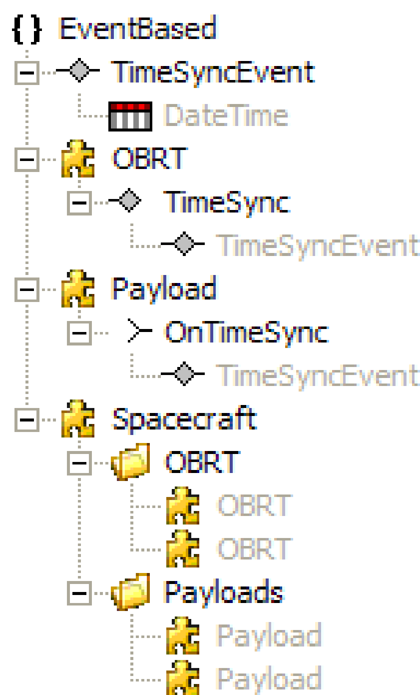

Figure 7-16: Catalogue Elements for Event-based Example

The `TimeSyncEvent` is defined as an event type with an event argument of type `DateTime`. The `OBRT` Model provides an event source of this event type, and the `Payload` Model consumes the event type via its event sink `OnTimeSync`. The `Spacecraft` model has been defined only to allow putting both models into a container.

---

[13] In UML 2.0, multiple event sources or event sinks of the same event type may be represented via ports attached to the event emitter or receiver model, respectively.

7.4.3 Catalogue File

The corresponding XML specification for the namespace, the event type and the models is shown in Figure 7-17.

```
<Namespace Id="EventBased" Name="EventBased">
  <Type xsi:type="Catalogue:EventType" Id="TimeSyncEvent" Name="TimeSyncEvent">
    <Uuid>8c878a63-8c0d-4768-bfff-a3b4da231819</Uuid>
    <EventArgs xlink:title="DateTime"
               xlink:href="http://www.esa.int/2005/10/Smp#DateTime" />
  </Type>
  <Type xsi:type="Catalogue:Model" Id="OBRT" Name="OBRT">
    <Uuid>1e7f60ec-faef-48aa-b0d6-494617d31f0e</Uuid>
    <EventSource Id="OBRT.TimeSync" Name="TimeSync">
      <Type xlink:title="TimeSyncEvent" xlink:href="#TimeSyncEvent" />
    </EventSource>
  </Type>
  <Type xsi:type="Catalogue:Model" Id="Payload" Name="Payload">
    <Uuid>aea95f10-bbaf-4628-aaba-2c397df36c56</Uuid>
    <EventSink Id="Payload.OnTimeSync" Name="OnTimeSync">
      <Type xlink:title="TimeSyncEvent" xlink:href="#TimeSyncEvent" />
    </EventSink>
  </Type>
  <Type xsi:type="Catalogue:Model" Id="Spacecraft" Name="Spacecraft">
    <Uuid>8c878a63-8c0d-4768-bfff-a3b4da231818</Uuid>
    <Container Id="Spacecraft.OBRT" Name="OBRT" Lower="0">
      <Type xlink:title="OBRT" xlink:href="#OBRT" />
      <DefaultModel xlink:title="OBRT" xlink:href="#OBRT" />
    </Container>
    <Container Id="Spacecraft.Payloads" Name="Payloads" Lower="0" Upper="-1">
      <Type xlink:title="Payload" xlink:href="#Payload" />
      <DefaultModel xlink:title="Payload" xlink:href="#Payload" />
    </Container>
  </Type>
</Namespace>
```

Figure 7-17: Catalogue File for Event-based Example

7.4.4 Model Definition Files

A compatible C++ definition of the OBRT model is shown below.

```
namespace EventBased
{
    class OBRT : public virtual Smp::IModel
    {
    private:
        char* name;                             ///< Name of model.
        char* description;                      ///< Description of model.
        Smp::IComposite* parent;                ///< Parent component.
    public:
        /// Public constructor with name, description and parent.
        OBRT(Smp::String8 name, Smp::String8 description,
             Smp::IComposite* parent);

        // TimeSync EventSource
        Smp::IEventSource* TimeSync;

        // IObject, IComponent and IModel methods
        virtual Smp::String8 GetName()        const;
        virtual Smp::String8 GetDescription() const;
        virtual Smp::IComposite* GetParent()  const;
        virtual Smp::ModelStateKind GetState() const;
        virtual void Publish(Smp::IPublication* receiver)
        throw (Smp::IModel::InvalidModelState);
        virtual void Configure(Smp::Services::ILogger* logger)
        throw (Smp::IModel::InvalidModelState);
        virtual void Connect(Smp::ISimulator* simulator)
        throw (Smp::IModel::InvalidModelState);
    };
```

 © EUROPEAN SPACE AGENCY, 2005

```
}
```

A compatible C++ definition of the `Payload` model is shown below.

```
namespace EventBased
{
    class Payload : public virtual Smp::IModel
    {
    private:
        char* name;                         ///< Name of model.
        char* description;                  ///< Description of model.
        Smp::IComposite* parent;            ///< Parent component.
    public:
        /// Public constructor with name, description and parent.
        Payload(Smp::String8 name, Smp::String8 description,
                Smp::IComposite* parent);

        // OnTimeSync EventSink
        Smp::IEventSink* OnTimeSync;

        // IObject, IComponent and IModel methods
        virtual Smp::String8 GetName()        const;
        virtual Smp::String8 GetDescription() const;
        virtual Smp::IComposite* GetParent()  const;
        virtual Smp::ModelStateKind GetState() const;
        virtual void Publish(Smp::IPublication* receiver)
        throw (Smp::IModel::InvalidModelState);
        virtual void Configure(Smp::Services::ILogger* logger)
        throw (Smp::IModel::InvalidModelState);
        virtual void Connect(Smp::ISimulator* simulator)
        throw (Smp::IModel::InvalidModelState);
    };
}
```

7.4.5 Assembly Elements

When using such a design in an Assembly, several instances of the `Payload` model can be created and their event sinks connected to the event source of the `OBRT` model, as indicated in Figure 7-19. The same information may be shown as a hierarchy by an Assembly editor as indicated in Figure 7-18.

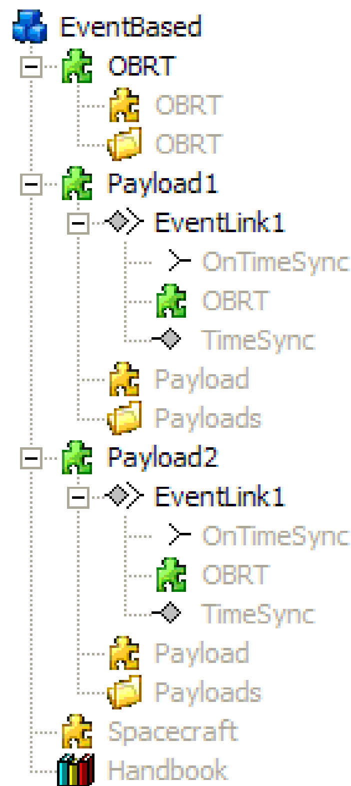

Figure 7-18: Assembly Elements for Event-based Example

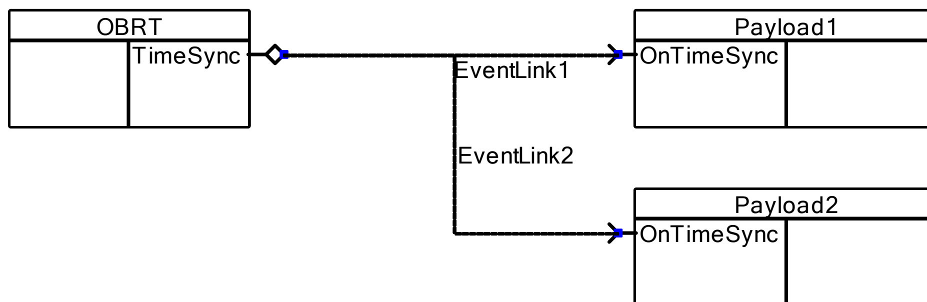

Figure 7-19: Example configuration using Event Links

Note that this diagram does not show the model types, but the model instances.

 © EUROPEAN SPACE AGENCY, 2005

7.4.6 Assembly File

The corresponding XML specification for the model integration is shown in Figure 7-20.

```
<?xml version="1.0" encoding="utf-8"?>
<Assembly:Assembly Name="EventBased" Date="2005-02-28T12:00:00+01:00"
                     Id="EventBased" Creator="pfritzen" Version="1.1"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  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:Assembly="http://www.esa.int/2005/10/Smdl/Assembly">
  <Model xlink:title="Spacecraft" xlink:href="Handbook.cat#Spacecraft" />
  <Implementation>00000000-0000-0000-0000-000000000000</Implementation>
  <ModelInstance Id="OBRT" Name="OBRT">
    <Model xlink:title="OBRT" xlink:href="Handbook.cat#OBRT" />
    <Implementation>00000000-0000-0000-0000-000000000000</Implementation>
    <Container xlink:title="OBRT" xlink:href="Handbook.cat#Spacecraft.OBRT" />
  </ModelInstance>
  <ModelInstance Id="Payload1" Name="Payload1">
    <Model xlink:title="Payload" xlink:href="Handbook.cat#Payload" />
    <Implementation>00000000-0000-0000-0000-000000000000</Implementation>
    <Link xsi:type="Assembly:EventLink" Id="Payload1.EventLink1" Name="EventLink1">
      <EventSink xlink:title="OnTimeSync" xlink:href="Handbook.cat#Payload.OnTimeSync" />
      <Provider xlink:title="OBRT" xlink:href="#OBRT" />
      <EventSource xlink:title="TimeSync" xlink:href="Handbook.cat#OBRT.TimeSync" />
    </Link>
    <Container xlink:title="Payloads" xlink:href="Handbook.cat#Spacecraft.Payloads" />
  </ModelInstance>
  <ModelInstance Id="Payload2" Name="Payload2">
    <Model xlink:title="Payload" xlink:href="Handbook.cat#Payload" />
    <Implementation>00000000-0000-0000-0000-000000000000</Implementation>
    <Link xsi:type="Assembly:EventLink" Id="Payload2.EventLink1" Name="EventLink1">
      <EventSink xlink:title="OnTimeSync" xlink:href="Handbook.cat#Payload.OnTimeSync" />
      <Provider xlink:title="OBRT" xlink:href="#OBRT" />
      <EventSource xlink:title="TimeSync" xlink:href="Handbook.cat#OBRT.TimeSync" />
    </Link>
    <Container xlink:title="Payloads" xlink:href="Handbook.cat#Spacecraft.Payloads" />
  </ModelInstance>
</Assembly:Assembly>
```

Figure 7-20: Assembly File for Event-based Example

This Page is Intentionally left Blank

© EUROPEAN SPACE AGENCY, 2005

8. APPENDIX A: GLOSSARY

8.1 Introduction

This section provides a glossary of essential terms and definitions used throughout the SMP2 specification.

As SMP2 is related to the Unified Modelling Language (**UML**) and the Model Driven Architecture (**MDA**) initiative of the Object Management Group (**OMG**), many terms from these areas are used within the SMP2 specification. The general intent is to be conformant with the UML and MDA terms. However, some of the terms defined below may be used in a more specific meaning within the SPM2 specification. Hence, if SMP2 uses a term from another standard (which is encouraged), then the term will be included in this section, together with a reference to the definition of the term and possibly a refined definition of the semantics used in SMP2.

8.2 Conventions

The following conventions are used within the glossary:

- When one or more words in a multi-word term are enclosed in brackets, it indicates that those words are optional when referring to the term. For example, use case [class] may be referred to as simply use case.
- A phrase of the form “Contrast: <term>” refers to a term that has an opposed or substantively different meaning.
- A phrase of the form “See: <term>” refers to a related term that has a similar, but not synonymous meaning.
- A phrase of the form “Synonym: <term>” refers to a related term that has a synonymous meaning.
- A phrase of the form “UML: <term>” or “MDA: <term>” refers to a related UML or MDA term.

8.3 Models

The term *model* is frequently used to denominate different artefacts at the same time – for example the model specification, the model implementation, and the model instance – which often leads to confusion. Therefore, SMP2 tries to be strict about terms and uses *model* in its different manifestations, where

- a *model [specification]* in a catalogue specifies a contract that a model implementation must fulfil,
- a *model implementation* in a library is a platform specific implementation of a model,
- a *model instance* in an assembly defines how a model implementation is used, and
- a *run-time [model] instance* is an in-memory object of a model implementation at run-time.

The details about these terms are described below.

8.4 SMP2 Terms and Definitions

Aggregation, Aggregate

A component may aggregate functionality of other components via interface references. The component then acts as a consumer of functionality that other components provide via interfaces.

Contrast: *composition, composite*

See: *component, provider, consumer, interface, reference*

UML: *aggregation, component, provided interface, required interface*

Assembly

An assembly *describes* the configuration of a set of model instances, specifying initial values for all fields, actual child instances, and type-based bindings of interface, events, and data fields. An assembly may be independent of any model implementations that fulfil the specified behaviour of the involved model types, i.e. an assembly may specify a scenario of model types without specifying their implementation. On the other hand, each model instance in an assembly is bound to a specific model in the catalogue for its specification.

Note: An assembly is independent from the actual model implementations. Therefore, the same assembly may be used, for example, to instantiate a run-time configuration of ANSI/ISO C++ model implementations or a run-time configuration of CORBA model implementations, provided that both sets of model implementations do exist. The actual binding of the model instances of the assembly to the platform specific model implementations is done via the *implementation* attribute of the model instances.

Contrast: *[run-time] configuration*

See: *model instance, dynamically configured simulation*

Catalogue

A catalogue contains namespaces as a primary ordering mechanism, where each namespace may contain types. These types can be language types, which include model types as well as other types like structures, classes, and interfaces. Additional types that can be defined are event types for inter-model events, and attribute types for typed metadata. The language for describing entities in the catalogue is the SMP2 Metamodel, or SMDL.

See: *model type*

Component

A component is a unit of functionality that can be deployed, and that has a well-defined interface to its environment (also called a contract). In SMP2, typical components are models and services.

See: *model, service, interface, contract*

UML: *component, provided interface, required interface*

Composition, Composite

A composite component may have other components as children that are held in containers.

Contrast: *aggregation, aggregate*

See: *component, container*

UML: *component, composition*

Consumer [component]

A consumer is a component that either

- invokes operations or properties on another component (called the provider) via a reference to one of the interfaces that the other component provides, or that
- receives event notifications sent from an event source of another component (called the provider) to one of its event sinks, or that

 © EUROPEAN SPACE AGENCY, 2005

- receives data from an output field of another component (called the provider or [data] source) to one of its input fields.

Contrast: *provider*

See: *event sink, input field, reference*

Container

A composite component may have other components as children that are held in containers. A container is typed by either an interface or a model type, specifying which types of children may be held in the container. Furthermore, it may specify the minimum and maximum number of allowed children.

Contrast: *reference*

See: *component, composition*

UML: *component, composition*

Contract

A contract defines how a component interacts with its environment. Typically, a contract consists of the interfaces provided by the component as well as the references that the component has to other components' interfaces. Furthermore, the contract may also include event sources and sinks.

In another context, a model type may also be seen as a contract for the corresponding model implementation.

See: *interface, reference, event source, event sink, model type, model implementation*

UML: *component, provided interface, required interface*

Dynamically configured simulation, dynamic configuration

A dynamically configured simulation is configured from an external file (typically an SMDL assembly file), i.e. model instances are created, configured and connected according to the information therein. In a dynamically configured simulation, a change of the model hierarchy, of initial values, of model links, or of model scheduling can be done without any recompilation by changing the assembly file and reloading the simulation.

Contrast: *statically configured simulation*

Entry point

An entry point is an operation without parameters that does not return a value (so-called void-void operation) that can be added to the Scheduler or Event Manager service, or to a Task that then may be scheduled. In SMP2, entry points are executed via an interface rather than via a direct function call as in SMP1/SMI.

In the case of a dynamically configured simulation, a component holding entry points is called a publisher.

See: *component, publisher, service*

© EUROPEAN SPACE AGENCY, 2005

Event sink

A component may use an event sink in order to receive a notification when a certain condition in another component is met. The binding to the event source of the other component (called the provider) is done via a subscription mechanism.

Contrast: *event source*

See: *consumer*

Event source

A component may provide an event source in order to allow other components to subscribe. When a certain condition in the component is met, the component may emit this event and notify all subscribed event sinks held in other components (called the consumers).

Contrast: *event sink*

See: *provider*

Exception

An exception is a special kind of signal, typically used to signal fault situations. The sender of the exception aborts execution and execution resumes with the receiver of the exception, which may be the sender itself. The receiver of an exception is determined implicitly by the interaction sequence during execution; it is not explicitly specified.

UML: *exception*

Field

A field is a valued feature of a structure, class, or model that is typed by a value type (called data item in SMP1/SMI). Fields typically hold the internal state of a model. They may be published to the simulation environment for the purpose of storing/restoring the state vector (`State=true`) or for visualisation or data monitoring purposes (`View=true`).

See: *input field, output field, property*

UML: *attribute*

Implementation inheritance

Implementation inheritance denotes the inheritance of the implementation of a more general element. Includes inheritance of the interface.

Contrast: *interface inheritance*

UML: *implementation inheritance*

 © EUROPEAN SPACE AGENCY, 2005

Input field

In a dataflow-based design, a model may expose input and output fields that are linked to entry points. Exposing an input field indicates that the model requires this field to be updated prior to the execution of the associated entry point.

Note that SMP2 only standardises this meta-information (in the catalogue) together with the possibility to specify field links (in the assembly). It does not, however, standardise the way in which this information is used in an implementation in order to actually perform the data transfer.

Contrast: *output field*

See: *consumer, entry point, target*

Interface

An interface is a named set of properties and operations that characterize the behaviour of a component or parts of it. A component may provide one or more interfaces that may be consumed by other components.

See: *aggregation, component, consumer, provider*

UML: *interface, provided interface*

Interface inheritance

Interface inheritance denotes the inheritance of the interface of a more general element. Does not include inheritance of the implementation.

Contrast: *implementation inheritance*

UML: *interface inheritance*

Model [type]

A model [type] *specifies* a contract for a set of model implementations. It does not depend on a specific target platform. Each model implementation has to implement all artefacts in the contract given by the model type, but it may also extend the contract by adding specific elements like operations, properties, and fields.

Contrast: *model implementation, model instance*

Model implementation

A model implementation implements the functionality of a model in a catalogue. The model implementation depends on the target platform (e.g. ANSI/ISO C++, CORBA, J2EE, .NET) and is intended for reuse by applications or other model implementations.

Note: Currently, we do not envisage cross-platform interoperability (i.e. interoperability of model implementations between different platforms). However, in the future, cross-platform bridges may be specified for this purpose.

Contrast: *model instance, model type, run-time instance*

Model instance

A model instance is part of an assembly. It is linked to a model type (specifying the contract) and may optionally specify a model implementation. It serves as a placeholder for the run-time model instance that is created when the assembly is used to instantiate a configuration.

See: *assembly*

Contrast: *model implementation, model type, run-time instance*

Model library

A model library contains model implementations, usually in binary or compiled form. In terms of implementation in ANSI/ISO C++, the analogous of a model library is a class library.

Note: On Windows platforms, binary class libraries are usually held in Dynamic Link Libraries (DLL), and on UNIX platforms they are usually held in Dynamic Shared Objects (DSO).

Contrast: *catalogue*

UML: *model library*

Output field

In a dataflow-based design, a model may expose input and output fields that are linked to entry points. Exposing an output field indicates that the model updates this field within the execution of the associated entry point – based on the values of the associated input fields (and possibly some internal state).

Note: SMP2 only standardises this meta-information (in the catalogue) together with the possibility to specify field links (in the assembly). It does not, however, standardise the way in which this information is used in an implementation in order to actually perform the data transfer.

Contrast: *input field*

See: *entry point, provider, source*

Platform

A platform is a set of subsystems/technologies that provide a coherent set of functionality through interfaces and specified usage patterns that any subsystem that depends on the platform can use without concern for the details of how the functionality provided by the platform is implemented.

In SMP2, typical platforms are: ANSI/ISO C++ with the appropriate component model specified by SMP2, CORBA with an appropriate component model like the CORBA Component Model (**CCM**), Java 2 Enterprise Edition (**J2EE**) or Java Beans, or the Microsoft .NET platform.

MDA: *platform*

Property

A property is a valued feature of a class, an interface or a model that can be accessed by two operations, the setter and the getter. A read-only property only has a getter, while a write-only property only has a setter. Properties may be used for controlled access to fields.

See: *field*

© EUROPEAN SPACE AGENCY, 2005

Provider [component]

A provider is a component that

- implements an interface (provided interface) whose operations or properties may be invoked by other components (called consumers), or that
- sends event notifications from one of its event sources to other components (called consumers) that have subscribed to this event source with one of their event sinks, or that
- exposes output fields that may be transferred to input fields of other components (called the consumers). The components holding output and input fields are also called *source* and *target*.

Contrast: *consumer*

See: *interface, event source, output field, source*

Publisher [component]

A publisher is a component that

- publishes entry points that may be added to the Scheduler or Event Manager service.

Contrast: *consumer*

See: *entry point, service*

Operation

An operation is a feature of a class, interface, or model, which declares a service that can be performed by instances. It may have parameters and it may return a value.

An operation may be static, in which case it is also called a *class method* in OO languages.

Synonym: *[instance] method, member function*

UML: *operation*

Reference

A component may reference interfaces of other components via aggregation. It consumes functionality from the other components (called providers) via its references. A reference is typed by an interface, specifying which types of components may be referenced. Furthermore, it may specify the minimum and maximum number of allowed references.

See: *aggregation, component, consumer*

UML: *interface, required interface*

© EUROPEAN SPACE AGENCY, 2005

Run-time [model] instance

A run-time model instance is a run-time instance of a model implementation, and is part of a run-time configuration.

In ANSI/ISO C++, this is usually an in-memory instance (i.e. an object) of a class, created, for example, by the `new` directive and initialised by some constructor of the class.

Contrast: *model implementation*

See: *[run-time] configuration*

[Run-time] Configuration

A run-time configuration is a run-time network of run-time model instances, which may be created according to the information in an associated assembly, taking the implementation attributes of the model instances into account. The run-time model instances are

- created according to platform bindings in the implementation attribute, and
- configured according to initial values, interface links and event links in the assembly.

Contrast: *assembly*

See: *run-time [model] instance*

Run-time environment

Synonym: *simulation environment, simulator*

Schedule

A schedule defines how entry points of model instances in an assembly are scheduled. Tasks may be used to group entry points. Typically, a schedule is used together with an assembly in a dynamically configured simulation.

See: *dynamically configured simulation*

Service [interface]

A service is a component that is held as a direct child of the simulation environment, and that provides global functionality to the models. A service is typically queried by name and its functionality is specified by a service *interface*. As SMP2 only standardises the interfaces, not their implementation, the terms *service* and *service interface* are sometimes used synonymously.

Synonym: *simulation service*

Simulation environment

A simulation environment is a component or application that provides SMP2 conformant functionality to SMP2 models. SMP2 does not specify how a simulation environment is implemented, but rather how models and services interface to the simulation environment. ESA uses a number of different simulation environments, for example EuroSim and SIMSAT.

Synonym: *run-time environment, simulator*

© EUROPEAN SPACE AGENCY, 2005

Simulator

Synonym: *run-time environment, simulation environment*

Simulation service

A simulation service is a service used within a simulation.

Synonym: *service*

Source [component]

A component that exposes one or more output fields is called a source.

See: *output field, provider*

Statically configured simulation, static configuration

A statically configured simulation is configured from some piece of source code, i.e. model instances are created and connected programmatically. In a statically configured simulation, a change of the model hierarchy, of initial values, of model links, or of model scheduling requires changing some source code.

Contrast: *dynamically configured simulation*

Target [component]

A component that exposes one or more input fields is called a target.

See: *consumer, input field*

© EUROPEAN SPACE AGENCY, 2005

8.5 SMDL Modelling Elements

This section provides an alphabetic list of all modelling elements defined in SMDL. The first column holds the name of the modelling element, the second column provides a short description, and the third column gives a reference to the relevant section in the SMP 2.0 Metamodel [AD-1] document.

Table 8-1: Alphabetical List of SMDL Modelling Elements

Element	Description	Reference
Activity	An `Activity` is the abstract base class for the three different activities that can be contained in a Task: Trigger, Transfer and SubTask.	7.2.1.1
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 (also called Sequence in CORBA, or Collection in .NET) 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).	4.2.8
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.	4.4.3
Assembly	An `Assembly` is a document that is typed by a `Model` of a catalogue. It is similar to a model instance (see below). This is intended, as it will allow replacing child model instances by sub-assemblies later if needed.	6.1.2
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.	6.1.1
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`).	5.2.3

© EUROPEAN SPACE AGENCY, 2005

Element	Description	Reference
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 achieve an automated mapping to C++. 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 does not have a direct equivalent in SMDL, but rather SMDL separates semantics and specifies two Metaclasses that could be modelled by UML attributes: fields and properties.	4.5.2
Attribute Type	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 simple type, and specifies a `Default` value.	4.5.1
Catalogue	A `Catalogue` is a file that defines types. It contains namespaces as a primary ordering mechanism. The names of these namespaces need to be unique within the catalogue.	5.1.1
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. In addition to the fields (`Field` elements) a class can contain because it is derived from structure, it can have arbitrary numbers of properties (`Property` elements) and operations (`Operation` elements).	5.2.1
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`.	3.3.1

© EUROPEAN SPACE AGENCY, 2005

Element	Description	Reference
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[14]. Furthermore, a component collection 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.	5.3.3.2
DateTime	This simple type is used to store date and time information. Internally, 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. - A tick is a nanosecond (10^{-9} s), which is the lowest level of granularity available. - A positive number corresponds to a data after the reference date, while negative numbers store dates before the reference date. - A value is serialized into an xsd:dateTime. With this definition, DateTime is compatible with Duration as defined below.	4.2.4.1
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 elements to allow identification of documents.	3.2.3
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.	3.3.2

[14] 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.*

© EUROPEAN SPACE AGENCY, 2005

Element	Description	Reference
Duration	This simple type is used to store duration information. Internally, 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. - A value is serialized into an xsd:duration. With this definition, Duration is compatible with DateTime as defined above.	4.2.4.2
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.	4.2.5
Enumeration Literal	An EnumerationLiteral assigns a Name (inherited from *NamedElement*) to an integer Value.	4.2.5.1
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. Entry Points are needed in the Models of a Catalogue to allow connecting Scheduling information later.	5.3.3.1
Epoch Event	An EpochEvent is derived from Event and adds an EpochTime attribute.	7.3.3
Event	An Event defines a point in time when to execute a collection of Task elements. Such a scheduler event can either be called only once, or it may be repeated using a given cycle time and count.	7.3.1
Event Link	An EventLink is used in an Assembly to connect an EventSource of an event Provider to an EventSink of an event consumer. Event source and event sink are defined by the corresponding Model definitions referenced from the ModelInstance instances.	6.2.3
Event Sink	An EventSink is used to specify that a Model can consume a specific event using a given name. On Assembly level, an EventSink can be connected to any number of EventSource instances using EventLink instances.	5.4.3

Element	Description	Reference
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 any number of sinks can connect to the source (the default), or only a single sink can connect (Multicast=false).	5.4.2
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.	5.4.1
Field	A Field is a feature that is typed by a value type, and 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. 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.	4.3.1
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).	6.2.4
Field Value	A FieldValue links to the defining Field and contains a value specification holding the actual Value.	4.4.5.1
Float	A Float type represents floating point values of type Float32 or Float64, 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.	4.2.7
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.	8.1.2
Integer	An Integer type represents integer values of an integer type (e.g. Int32), but with a given range of valid values (via the Minimum and Maximum attributes).	4.2.6

 © EUROPEAN SPACE AGENCY, 2005

Element	Description	Reference
Interface	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.	5.3.2
Interface Link	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 uniquely identify the link source, the `InterfaceLink` links to the `Provider` model instance, 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.	6.2.2
Language Type	A `LanguageType` is the abstract base Metaclass for value types (where instances are defined by their value), reference types (where instances are defined by their reference, i.e. their location in memory), and references to value types.	4.1.3
Link	The Link Metaclass does not introduce new attributes, but serves as a base class for derived Metaclasses.	6.2.1
Mission Event	A MissionEvent is derived from Event and adds a MissionTime attribute.	7.3.4

Element	Description	Reference
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 SMP2 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 Event Manager 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 define its own value types, which is represented by the `NestedType` elements.	5.3.3
Model Instance	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. 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.	6.1.3
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.	3.2.2

 © EUROPEAN SPACE AGENCY, 2005

Element	Description	Reference
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.	5.1.2
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**).	4.2.1
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
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.	8.1.1
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.	4.3.2.1
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.	4.2.1.1

Element	Description	Reference
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.	4.2.3
Property	A `Property` mainly has the same syntax as `Field`, but not the same semantics (see below). Additionally, a property can be assigned an `Access` attribute limiting access to one of `readWrite`, `readOnly`, and `writeOnly`. Furthermore, a property can be assigned a `Category` attribute to be used as ordering criteria in applications. 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 two access operations, namely the setter and/or the getter. If a property is read-only, there is no setter, if it is write-only, there is to getter. The actual implementation depends on the target platform and language. Compared to fields, properties have the advantage that there is no direct memory access, but every access is operation based. This allows mapping them to distributed platforms (e.g. CORBA), and ensures that the containing type always has knowledge about changes of its state. However, on implementation level, properties are frequently bound to a specific field. This can be expressed by linking to a field (of the same parent 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.	5.2.2
Reference	A `Reference` defines the rules of aggregation (links to interfaces) for a `Model`. The type of models that can be referenced is specified via the `Interface` link, while the `Lower` and `Upper` attributes specify the multiplicity, i.e. the number of possibly held references to models implementing this interface. Therein the upper bound may be unlimited, which is represented by `Upper=-1`.	5.3.3.3
Reference Type	An instance of a `ReferenceType` 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 may also be abstract, i.e. not instantiable. Reference types include interfaces, classes, components, and component types. A `ReferenceType` may hold any number of properties and operations.	5.3.1
Schedule	A `Schedule` is a `Document` that holds an arbitrary number of tasks (`Task` elements) and events (`TimedEvent` elements) triggering these tasks.	7.1.1

 © EUROPEAN SPACE AGENCY, 2005

Element	Description	Reference
Simple Type	A number of pre-defined types are needed in order to bootstrap the type system. These pre-defined value types are represented by meta-objects of the Metaclass `SimpleType` (in contrast to Metaclasses) and have a pre-defined `Id` attribute to allow unique identification across SMDL compliant applications, as well as the ability to create valid XML links to them. Do not use `SimpleType` to define new types.	4.2.2
Simple Value	The `SimpleValue` Metaclass is used for values of simple types. As each simple type corresponds to a different XML type, the `Value` element is not typed.	4.4.2
Simulation Event	A SimulationEvent is derived from Event and adds a SimulationTime attribute.	7.3.2
String	A `String` type represents fixed `Length` string values base on `Smp.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.	4.2.9
String Value	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.	4.4.4
Structure	A `Structure` type collects an arbitrary number of `Fields` representing the state of the structure. A `Field` of a `Structure` may only reference another `ValueType` or a `Reference` (to a `ValueType`).	4.2.10
Structure Value	A `StructureValue` holds field values for all fields of the corresponding structure type.	4.4.5
SubTask	A SubTask references another Task in a Schedule, to initiate all activities defined in it.	7.2.1.4
Task	A `Task` is a container of activities. The order of activities defines in which order the entry points referenced by the `Activity` elements are called.	7.2.1
Transfer	A Transfer selects a FieldLink defined in an Assembly, to initiate its execution to transfer data from the source to the target.	7.2.1.3
Trigger	A `Trigger` links an `EntryPoint` defined in a `Model` of a `Catalogue` to a specific `Provider` instance defined in an `Assembly`.	7.2.1.2

Element	Description	Reference
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). This is needed such that implementations may reference back to their specification without the need to directly reference an XML element in the catalogue.	4.1.2
Value	The `Value` Metaclass is an abstract base class for specialised value specifications.	4.4.1
Value Reference	A `ValueReference` is a type that references a specific value type. It is the "missing link" between value types and reference types.	4.1.5
Value Type	An instance of a `ValueType` is uniquely determined by its value, and does not have internal state. 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.	4.1.4
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`).	4.1.1
Workspace	A Workspace is a Document that references an arbitrary number of other SMDL documents via its Document links.	9.1.1
Zulu Event	A ZuluEvent is derived from Event and adds a ZuluTime attribute.	7.3.5

© EUROPEAN SPACE AGENCY, 2005

9. APPENDIX B: NOTES ON SUPPORTING TOOLS

This section collects ideas and plans for tools that support the SMP2 standard. As such, tools are not part of the SMP2 standard itself, but rather intended to be built on top of the standard in order to support it.

9.1 XSIM Prototype Tools

While not a formal part of the standard, the XML Based Simulation (**XSIM**) project will provide elements supporting the use of SMP2, which will be released after the specification has been accepted by the CCB. All software and tools provided would be *prototypes* that are built in order to verify and practically test the standard.

Model Development Kit: For the C++ platform mapping, a reference implementation will be provided that can be used as a starting point for implementing SMP2 models. While not a formal part of the standard, this Model Development Kit (**MDK**) will reduce the effort for implementing compliant models, and should therefore enable users to easily upgrade to SMP2.

Catalogue Editor: This application allows creating and editing SMDL Catalogue documents, which define types and models used within an SMP2 simulation.

Catalogue Validator: This tool allows validating a catalogue document. This is not a validation of the syntax against the XML Schema document defined for catalogues (as this can easily be done using existing tools), but a validation of the semantics. This semantics is defined as rules of the metamodel.

Code Generator: This tool generates C++ skeleton code for models defined within a catalogue document. This ensures that a model implementation in a library is synchronized with the corresponding model type in a catalogue. The source code generated by the code generator makes use of the C++ MDK.

Assembly Editor: This application allows creating and editing SMDL Assembly documents, which define model integration of models defined in SMDL Catalogues.

Assembly Validator: This tool allows validating an assembly document. This is not a validation of the syntax against the XML Schema document defined for catalogues (as this can easily be done using existing tools), but a validation of the semantics. This semantics is defined as rules of the metamodel, and by the catalogue(s) defining the models used within the assembly.

9.2 Simulation Development Life-Cycle

Figure 9-1 below shows a possible life cycle of simulation development based on the XSIM prototype tools. This is described in the following sections.

9.2.1 Model Design: The SMDL Catalogue

A *Catalogue* defines models, and simple types used by these models (structures, enumerations, etc.). The *Catalogue Editor* allows creating and editing Catalogues easily. The *Catalogue Validator* allows validating a Catalogue against rules (recommendations, warnings, errors).

9.2.2 Model Development: Implementing Models

A *Code Generator* translates models from the platform independent Catalogue into a target platform (here: C++). The *Model Wrapper Code* has to be completed by *Model Source Code* to provide the required functionality. The Model Development Kit (MDK) supports this task. Finally, a compiler is needed to generate binary models from sources.

© EUROPEAN SPACE AGENCY, 2005

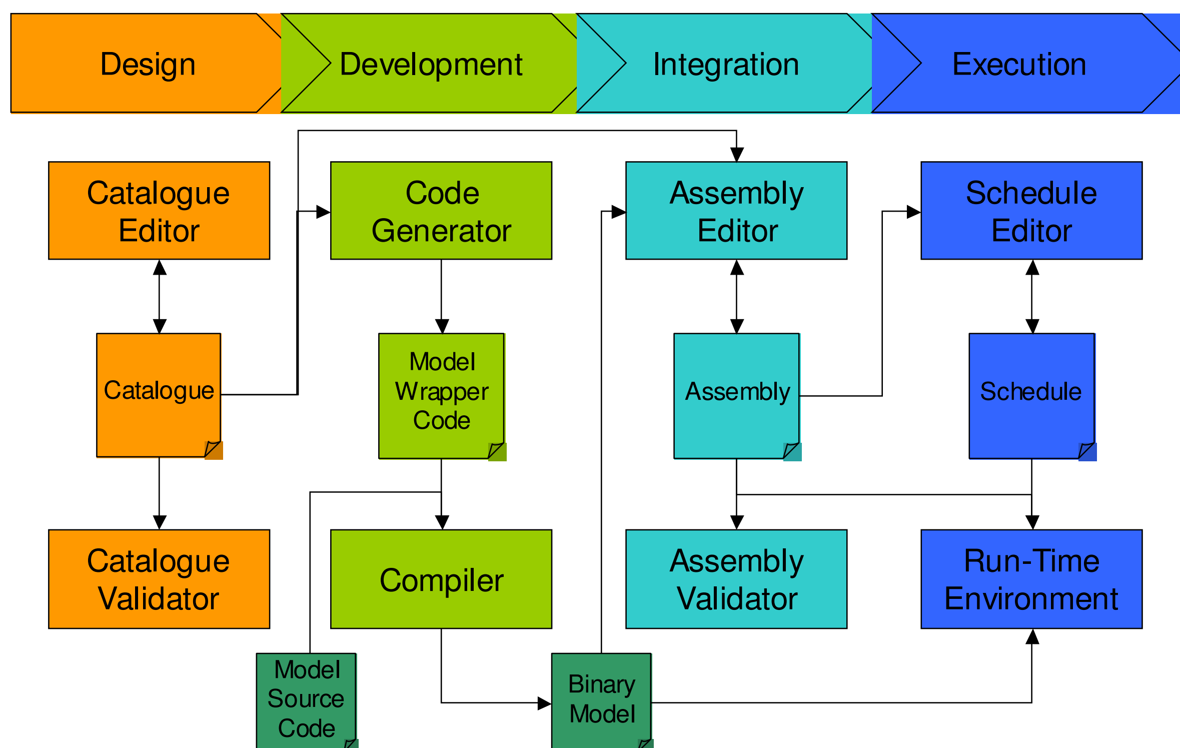

Figure 9-1: Possible Simulation Development Life-Cycle using XSIM Prototype Tools

9.2.3 Model Integration: The SMDL Assembly

An *Assembly* describes how existing models[15] are assembled for a specific purpose (links models together) and how the models are configured (initial values). The *Assembly Editor* allows assembling models easily, taking design constraints into account. The *Assembly Validator* allows validating an Assembly against rules (recommendations, warnings, errors).

9.2.4 Model Execution: The SMDL Schedule

A *Schedule* defines how entry points within the models of an assembly are to be scheduled. The *Schedule Editor* allows creating and editing a Schedule easily. The *Run-Time Environment* loads an *Assembly* and a *Schedule* and executes the simulation.

9.3 Other Support Tools

Apart from the XSIM Prototypes, we envisage a number of other support tools, for example:

- **Converters** to/from UML 2.0 and/or SysML
- **Code Parsers** that generate an SMDL Catalogue from legacy models
- **Wrapper Generators**, for example to produce SMP2 compliant Matlab/Simulink models
- **Run-time Validation**, for example by generating code that allows to validate design-time constraints during run-time (e.g. range, unit)

[15] *Strictly speaking, model instances are assembled, rather than models.*

© EUROPEAN SPACE AGENCY, 2005