**ECSS-E-TM-40-07 Volume 4A**
25 January 2011


# Space engineering

## Simulation modelling platform - Volume 4: C++ Mapping

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

## Foreword

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

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

## Disclaimer



Published by: ESA Requirements and Standards Division
ESTEC, P.O. Box 299,
2200 AG Noordwijk
The Netherlands

# Change log

| | |
|---|---|
| ECSS-E-TM-40-07 Volume 4A<br>25 January 2011 | First issue |

# Table of contents

**Change log** ..... **3**

**Introduction** ..... **12**

**1 Scope** ..... **14**

**2 Base Types** ..... **15**

2.1 Primitive Types ..... 15
2.1.1 Char8 ..... 15
2.1.2 Bool ..... 15
2.1.3 Int8 ..... 16
2.1.4 UInt8 ..... 16
2.1.5 Int16 ..... 16
2.1.6 UInt16 ..... 17
2.1.7 Int32 ..... 17
2.1.8 UInt32 ..... 17
2.1.9 Int64 ..... 18
2.1.10 UInt64 ..... 18
2.1.11 Float32 ..... 18
2.1.12 Float64 ..... 19
2.1.13 Duration ..... 19
2.1.14 Date Time ..... 20
2.1.15 String8 ..... 21
2.2 Simple Type Union ..... 21
2.2.1 Primitive Type Kind ..... 21
2.2.2 Any Simple Array ..... 23
2.2.3 Any Simple ..... 23
2.2.4 Primitive Type Value ..... 24
2.3 Universally Unique Identifiers ..... 25
2.3.1 Uuid ..... 25
2.3.2 Uuid Bytes ..... 26

**3 Component Model ..... 27**

3.1 Exceptions ..... 27

3.1.1 Exception ..... 27

3.1.2 Duplicate Name ..... 28

3.1.3 Invalid Any Type ..... 29

3.1.4 Invalid Event Id ..... 30

3.1.5 Invalid Object Type ..... 31

3.2 Objects and Components ..... 32

3.2.1 Objects ..... 32

3.2.1.1 IObject ..... 32

3.2.2 Components ..... 34

3.2.2.1 IComponent ..... 34

3.2.2.2 Component Collection ..... 35

3.2.2.3 Model State Kind ..... 36

3.2.2.4 IModel ..... 37

3.2.2.5 Model Collection ..... 41

3.2.2.6 IService ..... 42

3.2.2.7 Service Collection ..... 42

3.3 Component Mechanisms ..... 43

3.3.1 Aggregation ..... 43

3.3.1.1 IAggregate ..... 43

3.3.1.2 IReference ..... 45

3.3.1.3 Reference Collection ..... 46

3.3.2 Composition ..... 47

3.3.2.1 IComposite ..... 47

3.3.2.2 IContainer ..... 48

3.3.2.3 Container Collection ..... 50

3.3.3 Events ..... 50

3.3.3.1 IEvent Sink ..... 50

3.3.3.2 Event Sink Collection ..... 52

3.3.3.3 IEvent Source ..... 52

3.3.3.4 Event Source Collection ..... 57

3.3.4 Entry Points ..... 57

3.3.4.1 ITask ..... 57

3.3.4.2 IEntry Point ..... 59

3.3.4.3 Entry Point Collection ..... 60

3.3.5 Dynamic Invocation ..... 61

3.3.5.1 IDynamic Invocation ..... 62

3.3.5.2 IRequest ..... 68

3.3.6 Persistence ..... 75

3.3.6.1 IPersist ..... 75

3.3.6.2 IStorage Reader ..... 78

3.3.6.3 IStorage Writer ..... 79

3.4 Model Mechanisms ..... 80

3.4.1 Fallible Models ..... 80

3.4.1.1 IFailure ..... 81

3.4.1.2 IFallible Model ..... 82

3.5 Management Interfaces ..... 84

3.5.1 Managed Components ..... 84
3.5.1.1 IManaged Object ..... 84
3.5.1.2 IManaged Component ..... 87
3.5.2 Managed Component Mechanisms ..... 88
3.5.2.1 IComponent Collection ..... 89
3.5.2.2 IManaged Reference ..... 91
3.5.2.3 IManaged Container ..... 95
3.5.2.4 IEvent Consumer ..... 97
3.5.2.5 IEvent Provider ..... 98
3.5.2.6 IEntry Point Publisher ..... 100
3.5.3 Managed Model Mechanisms ..... 102
3.5.3.1 IManaged Model ..... 102
3.5.3.2 IField ..... 105
3.5.3.3 IArray Field ..... 108
3.5.3.4 ISimple Field ..... 113
3.5.3.5 IForcible Field ..... 115
3.5.3.6 View Kind ..... 116
3.6 Simulation Environments ..... 118
3.6.1 Simulators ..... 118
3.6.1.1 Simulator State Kind ..... 119
3.6.1.2 ISimulator ..... 123
3.6.1.3 IManaged Simulator ..... 134
3.6.1.4 IFactory ..... 138
3.6.1.5 Factory Collection ..... 140
3.6.2 Publication ..... 140
3.6.3 Service Acquisition ..... 140
**4 Simulation Services ..... 142**
4.1 Logger ..... 142
4.1.1 ILogger ..... 142
4.1.1.1 Log ..... 144
4.1.1.2 Query Log Message Kind ..... 144
4.1.2 Log Message Kind ..... 144
4.1.3 Predefined Log Message Kinds ..... 145
4.1.4 User defined Log Message Kinds ..... 146
4.2 Time Keeper ..... 146
4.2.1 ITime Keeper ..... 146
4.2.1.1 Get Epoch Time ..... 148
4.2.1.2 Get Mission Time ..... 148
4.2.1.3 Get Simulation Time ..... 148
4.2.1.4 Get Zulu Time ..... 149
4.2.1.5 Set Epoch Time ..... 149
4.2.1.6 Set Mission Start ..... 149
4.2.1.7 Set Mission Time ..... 150
4.2.2 Time Kind ..... 150
4.3 Scheduler ..... 154
4.3.1 IScheduler ..... 154
4.3.1.1 Add Epoch Time Event ..... 160
4.3.1.2 Add Immediate Event ..... 160
4.3.1.3 Add Mission Time Event ..... 161

4.3.1.4 Add Simulation Time Event ..... 161
4.3.1.5 Add Zulu Time Event ..... 162
4.3.1.6 Remove Event ..... 163
4.3.1.7 Set Event Count ..... 163
4.3.1.8 Set Event Cycle Time ..... 164
4.3.1.9 Set Event Epoch Time ..... 164
4.3.1.10 Set Event Mission Time ..... 164
4.3.1.11 Set Event Simulation Time ..... 165
4.3.1.12 Set Event Zulu Time ..... 165
4.3.1.13 Invalid Cycle Time ..... 166
4.3.1.14 Invalid Event Time ..... 166

4.4 Event Manager ..... 167

4.4.1 IEvent Manager ..... 167
4.4.1.1 Emit ..... 174
4.4.1.2 Query Event Id ..... 174
4.4.1.3 Subscribe ..... 175
4.4.1.4 Unsubscribe ..... 175
4.4.1.5 Already Subscribed ..... 175
4.4.1.6 Not Subscribed ..... 176

4.4.2 Predefined Event Types ..... 178

4.5 Resolver ..... 180

4.5.1 IResolver ..... 180
4.5.1.1 Resolve Absolute ..... 181
4.5.1.2 Resolve Relative ..... 181

4.5.2 Component Paths ..... 182

4.6 Link Registry ..... 182

4.6.1 ILink Registry ..... 182
4.6.1.1 Add Link ..... 184
4.6.1.2 Can Remove ..... 184
4.6.1.3 Get Links ..... 185
4.6.1.4 Has Link ..... 185
4.6.1.5 Remove Link ..... 186
4.6.1.6 Remove Links ..... 186

**5 Publication ..... 187**

5.1 Type Registry ..... 187

5.1.1 IType Registry ..... 188
5.1.1.1 Add Array Type ..... 191
5.1.1.2 Add Class Type ..... 192
5.1.1.3 Add Enumeration Type ..... 192
5.1.1.4 Add Float Type ..... 192
5.1.1.5 Add Integer Type ..... 193
5.1.1.6 Add String Type ..... 193
5.1.1.7 Add Structure Type ..... 194
5.1.1.8 Get Type ..... 194
5.1.1.9 Get Type ..... 194
5.1.1.10 Already Registered ..... 195

5.1.2 IType ..... 195
5.1.2.1 Get Primitive Type ..... 196
5.1.2.2 Get Uuid ..... 197
5.1.2.3 Publish ..... 197

5.1.3 IEnumeration Type ..... 197
5.1.3.1 Add Literal ..... 198

5.1.4 IStructure Type ..... 198
5.1.4.1 Add Field ..... 199
5.1.5 IClass Type ..... 200
5.1.6 Not Registered ..... 200
5.2 Publication of Fields, Operations and Properties ..... 201
5.2.1 IPublish Operation ..... 201
5.2.1.1 Publish Parameter ..... 202
5.2.2 IPublication ..... 202
5.2.2.1 Create Request ..... 209
5.2.2.2 Delete Request ..... 209
5.2.2.3 Get Array Field ..... 209
5.2.2.4 Get Simple Field ..... 210
5.2.2.5 Get Type Registry ..... 210
5.2.2.6 Publish Array ..... 210
5.2.2.7 Publish Array ..... 211
5.2.2.8 Publish Field ..... 211
5.2.2.9 Publish Field ..... 212
5.2.2.10 Publish Field ..... 212
5.2.2.11 Publish Field ..... 212
5.2.2.12 Publish Field ..... 213
5.2.2.13 Publish Field ..... 213
5.2.2.14 Publish Field ..... 214
5.2.2.15 Publish Field ..... 214
5.2.2.16 Publish Field ..... 214
5.2.2.17 Publish Field ..... 215
5.2.2.18 Publish Field ..... 215
5.2.2.19 Publish Field ..... 215
5.2.2.20 Publish Field ..... 216
5.2.2.21 Publish Operation ..... 216
5.2.2.22 Publish Property ..... 217
5.2.2.23 Publish Structure ..... 217
5.2.2.24 Invalid Field Type ..... 217
5.2.3 Access Kind ..... 218
5.2.4 Parameter Direction Kind ..... 218
**6 Metamodel ..... 220**
6.1 Overview ..... 220
6.1.1 Placeholders ..... 220
6.1.2 Coloring and Font Schema ..... 220
6.1.3 Generation of Type Identification ..... 220
6.1.4 Alternative Code ..... 221
6.1.5 Optional Code ..... 221
6.2 C++ Specific Attributes ..... 221
6.2.1 C++ Attributes ..... 221
6.2.1.1 Abstract ..... 221
6.2.1.2 Base Class ..... 222
6.2.1.3 By Reference ..... 222
6.2.1.4 Const ..... 223
6.2.1.5 Constructor ..... 223
6.2.1.6 Operator ..... 224
6.2.1.7 Operator Kind ..... 224

6.2.1.8 Static....226
6.2.1.9 Virtual....227
6.3 Core Elements....227
6.3.1 Basic Types....227
6.3.1.1 Identifier....227
6.3.1.2 Name....227
6.3.1.3 Description....227
6.3.1.4 UUID....227
6.3.2 Elements....227
6.3.2.1 Named Element....227
6.3.2.2 Document....228
6.3.3 Metadata....228
6.3.3.1 Metadata....228
6.3.3.2 Comment....228
6.3.3.3 Documentation....228
6.4 Core Types....228
6.4.1 Types....228
6.4.1.1 Visibility Element....228
6.4.1.2 Visibility Kind....228
6.4.1.3 Type....229
6.4.1.4 Language Type....229
6.4.1.5 Value Type....229
6.4.1.6 Value Reference....229
6.4.1.7 Native Type and Platform Mapping....229
6.4.2 Value Types....231
6.4.2.1 Simple Type....231
6.4.2.2 Primitive Type....231
6.4.2.3 Enumeration....231
6.4.2.4 Enumeration Literal....231
6.4.2.5 Integer....231
6.4.2.6 Float....232
6.4.2.7 String....232
6.4.2.8 Array....233
6.4.2.9 Structure....233
6.4.2.10 Class....234
6.4.2.11 Exception....235
6.4.3 Features....236
6.4.3.1 Constant....236
6.4.3.2 Field....236
6.4.3.3 Property....237
6.4.3.4 Association....238
6.4.3.5 Operation....238
6.4.3.6 Parameter....240
6.4.4 Values....240
6.4.4.1 Value....240
6.4.4.2 Simple Value....240
6.4.4.3 Simple Array Value....241
6.4.4.4 Array Value....242
6.4.4.5 Structure Value....242
6.4.5 Attributes....242
6.5 Smdl Catalogue....243
6.5.1 Catalogue....243
6.5.1.1 Catalogue....243

6.5.1.2 Namespace ....243
6.5.2 Reference Types ....243
6.5.2.1 Reference Type ....243
6.5.2.2 Component ....243
6.5.2.3 Interface ....243
6.5.2.4 Model ....244
6.5.2.5 Service ....245
6.5.3 Modelling Mechanisms ....247
6.5.3.1 Class based Design ....247
6.5.3.2 Component based Design ....247
6.5.3.3 Event based Design ....247
6.5.4 Catalogue Attributes ....248
6.5.4.1 View and ViewKind ....248
6.6 Smdl Package ....249
6.6.1 Package ....249
6.6.1.1 Package ....249
6.6.1.2 Implementation ....250
6.6.2 Bundle Format ....250
6.6.3 Binary Distribution ....253

## Figures

Figure 1 - Sequence of calls for dynamic invocation ....61
Figure 2 - Simulation Environment State Diagram with State Transition Methods ....118
Figure 3 - Model State Diagram with State Transition Methods ....119
Figure 4 - Sequence of calls for service acquisition ....141
Figure 5 - Predefined Event Types ....178
Figure 6 - Abstract ....222
Figure 7 - Base Class ....222
Figure 8 - By Reference ....222
Figure 9 - Const ....223
Figure 10 - Constructor ....224
Figure 11 - Operator ....224
Figure 12 - Operator Kind ....225
Figure 13 - Static ....226
Figure 14 - Virtual ....227

## Tables

Table 1 - Enumeration Literals of PrimitiveTypeKind ....22
Table 2 - Specification of UuidBytes ....26
Table 3 - Enumeration Literals of ModelStateKind ....37
Table 4 - Enumeration Literals of ViewKind ....117
Table 5 - Enumeration Literals of SimulatorStateKind ....121

Table 6 - Specification of LogMessageKind ..... 145
Table 7 - Enumeration Literals of TimeKind ..... 151
Table 8 - Enumeration Literals of AccessKind ..... 218
Table 9 - Enumeration Literals of ParameterDirectionKind ..... 219
Table 10 - Enumeration Literals of OperatorKind ..... 225
Table 11 – Property Type Modifier depending on type and attribute ..... 237
Table 12 – Association Type Modifier depending on type and attribute ..... 238
Table 13 - Parameter Modifier depending on type and direction ..... 240

# Introduction

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

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

This Technical Memorandum comprises of a number of volumes.

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

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

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

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

- **Volume 1 – Principles and requirements**

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

- **Volume 2 - Metamodel**

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

- **Volume 3 - Component Model**

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

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

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

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

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

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

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

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

- **Volume 5 – SMP usage**

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

# 1 Scope

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

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

This document provides a mapping of both parts of the PIM to the ISO/ANSI C++ Platform as defined by the International Organization for Standardization (**ISO**) and the American National Standards Institute (**ANSI**). Therefore, this document provides a Platform Specific Model (**PSM**). The mapping of these two parts is different in nature:

- The SMP Component Model is mapped to a number of C++ types, which are available in source code format as well (see annex).
- The SMP Metamodel is mapped to a process of how to turn Metaclasses into C++ types which make use of the Component Model C++ types.

Due to the dependency of the Metamodel mapping on the Component Model mapping, the latter is introduced first in this document.

# 2 Base Types

## 2.1 Primitive Types

The type system is based on a number of pre-defined primitive types. For each platform, a mapping of these types to native platform types has to be provided. The semantics of these types is defined in the Component Model, so that they have a consisting semantics on all platforms.

As various C++ compilers on various operating systems differ in their definition especially of the integer types, the SMP C++ type system is bootstrapped with the definition of Native Types for all Primitive Types. This definition is included from a file called `Platform.h`, which has to be provided for each compiler and operating system. The file delivered with the standard already provides a number of common compilers and operating systems.

### 2.1.1 Char8

8 bit character type.

**File**

#include "Smp/PrimitiveTypes.h"

**Namespace**

Smp

**Declaration of Char8**

```
/// Unique Identifier of type Char8.
extern const Uuid Uuid_Char8;

/// 8 bit character type.
typedef NT_Char8 Char8;
```

### 2.1.2 Bool

Boolean with true and false.

**File**

#include "Smp/PrimitiveTypes.h"

**Namespace**

Smp

**Declaration of Bool**

```
/// Unique Identifier of type Bool.
extern const Uuid Uuid_Bool;

/// Boolean with true and false.
typedef NT_Bool Bool;
```

### 2.1.3 Int8

8 bit signed integer type.

**File**

#include "Smp/PrimitiveTypes.h"

**Namespace**

Smp

**Declaration of Int8**

```
/// Unique Identifier of type Int8.
extern const Uuid Uuid_Int8;

/// 8 bit signed integer type.
typedef NT_Int8 Int8;
```

### 2.1.4 UInt8

8 bit unsigned integer type.

**File**

#include "Smp/PrimitiveTypes.h"

**Namespace**

Smp

**Declaration of UInt8**

```
/// Unique Identifier of type UInt8.
extern const Uuid Uuid_UInt8;

/// 8 bit unsigned integer type.
typedef NT_UInt8 UInt8;
```

### 2.1.5 Int16

16 bit signed integer type.

**File**

#include "Smp/PrimitiveTypes.h"

**Namespace**

Smp

**Declaration of Int16**

```
/// Unique Identifier of type Int16.
extern const Uuid Uuid_Int16;

/// 16 bit signed integer type.
typedef NT_Int16 Int16;
```

### 2.1.6 UInt16

16 bit unsigned integer type.

**File**

#include "Smp/PrimitiveTypes.h"

**Namespace**

Smp

**Declaration of UInt16**

```
/// Unique Identifier of type UInt16.
extern const Uuid Uuid_UInt16;

/// 16 bit unsigned integer type.
typedef NT_UInt16 UInt16;
```

### 2.1.7 Int32

32 bit signed integer type.

**File**

#include "Smp/PrimitiveTypes.h"

**Namespace**

Smp

**Declaration of Int32**

```
/// Unique Identifier of type Int32.
extern const Uuid Uuid_Int32;

/// 32 bit signed integer type.
typedef NT_Int32 Int32;
```

### 2.1.8 UInt32

32 bit unsigned integer type.

**File**

#include "Smp/PrimitiveTypes.h"

**Namespace**

Smp

**Declaration of UInt32**

```
/// Unique Identifier of type UInt32.
extern const Uuid Uuid_UInt32;

/// 32 bit unsigned integer type.
typedef NT_UInt32 UInt32;
```

### 2.1.9 Int64

64 bit signed integer type.

**File**

#include "Smp/PrimitiveTypes.h"

**Namespace**

Smp

**Declaration of Int64**

```
/// Unique Identifier of type Int64.
extern const Uuid Uuid_Int64;

/// 64 bit signed integer type.
typedef NT_Int64 Int64;
```

### 2.1.10 UInt64

64 bit unsigned integer type.

**File**

#include "Smp/PrimitiveTypes.h"

**Namespace**

Smp

**Declaration of UInt64**

```
/// Unique Identifier of type UInt64.
extern const Uuid Uuid_UInt64;

/// 64 bit unsigned integer type.
typedef NT_UInt64 UInt64;
```

### 2.1.11 Float32

32-bit single-precision float type.

**File**

#include "Smp/PrimitiveTypes.h"

**Namespace**

Smp

**Declaration of Float32**

```
/// Unique Identifier of type Float32.
extern const Uuid Uuid_Float32;

/// 32-bit single-precision float type.
typedef NT_Float32 Float32;
```

### 2.1.12 Float64

64-bit double-precision float type.

**File**

#include "Smp/PrimitiveTypes.h"

**Namespace**

Smp

**Declaration of Float64**

```
/// Unique Identifier of type Float64.
extern const Uuid Uuid_Float64;

/// 64-bit double-precision float type.
typedef NT_Float64 Float64;
```

### 2.1.13 Duration

Duration in nanoseconds.

This type is used for relative time values. It specifies a duration in nanoseconds. The following holds for Duration:

- Duration is a value measured in nanoseconds, which is the lowest level of granularity supported for time in SMP.
- Duration is stored as a signed 64-bit integer value.
- Positive values correspond to positive durations, and negative values correspond to negative durations.

This allows specifying duration values roughly between -290 years and 290 years. With this definition, the Duration type is compatible with the DateTime type.

**File**

#include "Smp/PrimitiveTypes.h"

**Namespace**

Smp

**Declaration of Duration**

```
/// Unique Identifier of type Duration.
extern const Uuid Uuid_Duration;

/// Duration in nanoseconds.
/// This type is used for relative time values. It specifies a duration in
/// nanoseconds. The following holds for Duration:
/// <ul>
///      - Duration is a value measured in nanoseconds, which is the lowest
/// level of granularity supported for time in SMP.
```

```
    ///      - Duration is stored as a signed 64-bit integer value.
    ///      - Positive values correspond to positive durations, and negative
    /// values correspond to negative durations.
    /// </ul>
    /// This allows specifying duration values roughly between -290 years and
    /// 290 years. With this definition, the Duration type is compatible with
    /// the DateTime type.
    typedef NT_Duration Duration;
```

### 2.1.14 Date Time

Absolute time in nanoseconds.

This type is used for absolute time values. It specifies a time in nanoseconds, relative to a reference time.

The following holds for DateTime:

- Time is a value measured in nanoseconds, which is the lowest level of granularity supported for time in SMP.
- Time is stored as a signed 64-bit integer value, relative to the reference time (01.01.2000, 12:00, Modified Julian Date (MJD) 2000+0.5).
- Positive values correspond to times after the reference time, and negative values correspond to time values before the reference time.

This allows specifying time values roughly between 1710 and 2290. With this definition, the DateTime type is compatible with the Duration type.

**File**

#include "Smp/PrimitiveTypes.h"

**Namespace**

Smp

**Declaration of DateTime**

```
    /// Unique Identifier of type DateTime.
    extern const Uuid Uuid_DateTime;

    /// Absolute time in nanoseconds.
    /// This type is used for absolute time values. It specifies a time in
    /// nanoseconds, relative to a reference time.
    /// The following holds for DateTime:
    /// <ul>
    ///      - Time is a value measured in nanoseconds, which is the lowest
    /// level of granularity supported for time in SMP.
    ///      - Time is stored as a signed 64-bit integer value, relative to the
    /// reference time (01.01.2000, 12:00, Modified Julian Date (MJD)
    /// 2000+0.5).
    ///      - Positive values correspond to times after the reference time,
    /// and negative values correspond to time values before the reference
    /// time.
    /// </ul>
    /// This allows specifying time values roughly between 1710 and 2290. With
    /// this definition, the DateTime type is compatible with the Duration
    /// type.
    typedef NT_DateTime DateTime;
```

### 2.1.15 String8

8 bit character string.

**File**

#include "Smp/PrimitiveTypes.h"

**Namespace**

Smp

**Declaration of String8**

```
/// Unique Identifier of type String8.
extern const Uuid Uuid_String8;

/// 8 bit character string.
typedef NT_String8 String8;
```

The C++ language has limited support for strings. Many applications use null-terminated arrays of characters to store string values, others use the string class from the Standard Template Library (**STL**). In the PSM, all strings of the PIM are mapped to constant character pointers (`const Char8*`).

## 2.2 Simple Type Union

Some of the interfaces defined in the Component Model make use of the `AnySimple` type. This data type represents a type that can hold any of the simple types defined in SMP, which includes all primitive types defined above. As for the primitive types, a native platform mapping has to be provided for each platform.

### 2.2.1 Primitive Type Kind

This is an enumeration of the available primitive types.

**File**

#include "Smp/PrimitiveTypes.h"

**Namespace**

Smp

**Declaration of PrimitiveTypeKind**

```
/// Unique Identifier of type PrimitiveTypeKind.
extern const Uuid Uuid_PrimitiveTypeKind;

/// This is an enumeration of the available primitive types.
enum PrimitiveTypeKind
{
    /// No type, e.g. for void.
    PTK_None,

    /// 8 bit character type.
    PTK_Char8,
```

```
        /// Boolean with true and false.
        PTK_Bool,

        /// 8 bit signed integer type.
        PTK_Int8,

        /// 8 bit unsigned integer type.
        PTK_UInt8,

        /// 16 bit signed integer type.
        PTK_Int16,

        /// 16 bit unsigned integer type.
        PTK_UInt16,

        /// 32 bit signed integer type.
        PTK_Int32,

        /// 32 bit unsigned integer type.
        PTK_UInt32,

        /// 64 bit signed integer type.
        PTK_Int64,

        /// 64 bit unsigned integer type.
        PTK_UInt64,

        /// 32 bit single-precision floating-point type.
        PTK_Float32,

        /// 64 bit double-precision floating-point type.
        PTK_Float64,

        /// Duration in nanoseconds.
        PTK_Duration,

        /// Absolute time in nanoseconds.
        PTK_DateTime,

        /// 8 bit character string.
        PTK_String8
    };
```

**Table 1 - Enumeration Literals of PrimitiveTypeKind**

| Name | Description |
|---|---|
| PTK_None | No type, e.g. for void. |
| PTK_Char8 | 8 bit character type. |
| PTK_Bool | Boolean with true and false. |
| PTK_Int8 | 8 bit signed integer type. |
| PTK_UInt8 | 8 bit unsigned integer type. |
| PTK_Int16 | 16 bit signed integer type. |
| PTK_UInt16 | 16 bit unsigned integer type. |
| PTK_Int32 | 32 bit signed integer type. |
| PTK_UInt32 | 32 bit unsigned integer type. |
| PTK_Int64 | 64 bit signed integer type. |
| PTK_UInt64 | 64 bit unsigned integer type. |

| Name | Description |
|---|---|
| PTK_Float32 | 32 bit single-precision floating-point type. |
| PTK_Float64 | 64 bit double-precision floating-point type. |
| PTK_Duration | Duration in nanoseconds. |
| PTK_DateTime | Absolute time in nanoseconds. |
| PTK_String8 | 8 bit character string. |

### 2.2.2 Any Simple Array

Array of AnySimple values.

**Namespace**

Smp

### 2.2.3 Any Simple

Variant that can store a value of any of the simple types. The type attribute defines the type used to represent the value, while the value attribute contains the actual value.

**File**

#include "Smp/AnySimple.h"

**Namespace**

Smp

**Declaration of AnySimple**

```
/// Unique Identifier of type AnySimple.
extern const Uuid Uuid_AnySimple;

/// Variant that can store a value of any of the simple types. The type
/// attribute defines the type used to represent the value, while the value
/// attribute contains the actual value.
struct AnySimple
{
    /// Type of the contained value.
    Smp::PrimitiveTypeKind type;

    /// Value stored in the AnySimple instance.
    Smp::PrimitiveTypeValue value;

    /// Default constructor.
    AnySimple();

    /// Copy constructor from another AnySimple instance.
    /// @param   that Instance to copy from.
    AnySimple(const Smp::AnySimple& that);

    /// Destructor that releases memory.
    virtual ~AnySimple();

    /// Assignment operator from another AnySimple instance.
    /// @param   that Instance to assign from.
    /// @return  Reference to instance.
    virtual Smp::AnySimple& operator=(const Smp::AnySimple& that);
};
```

### 2.2.4 Primitive Type Value

Union of primitive type values, which is used for the value field of AnySimple.

**File**

#include "Smp/PrimitiveTypes.h"

**Namespace**

Smp

**Declaration of PrimitiveTypeValue**

```
/// Unique Identifier of type PrimitiveTypeValue.
extern const Uuid Uuid_PrimitiveTypeValue;

/// Union of primitive type values, which is used for the value field of
/// AnySimple.
union PrimitiveTypeValue
{
    /// 8 bit character value.
    Smp::Char8 char8Value;

    /// Boolean with true and false.
    Smp::Bool boolValue;

    /// 8 bit signed integer value.
    Smp::Int8 int8Value;

    /// 8 bit unsigned integer value.
    Smp::UInt8 uInt8Value;

    /// 16 bit signed integer value.
    Smp::Int16 int16Value;

    /// 16 bit unsigned integer value.
    Smp::UInt16 uInt16Value;

    /// 32 bit signed integer value.
    Smp::Int32 int32Value;

    /// 32 bit unsigned integer value.
    Smp::UInt32 uInt32Value;

    /// 64 bit signed integer value.
    Smp::Int64 int64Value;

    /// 64 bit unsigned integer value.
    Smp::UInt64 uInt64Value;

    /// 32 bit single-precision floating-point value.
    Smp::Float32 float32Value;

    /// 64 bit double-precision floating-point value.
    Smp::Float64 float64Value;

    /// Duration in nanoseconds.
    Smp::Duration durationValue;

    /// Absolute time in nanoseconds.
    Smp::DateTime dateTimeValue;

    /// 8 bit character string value.
    Smp::String8 string8Value;
};
```

**Alternatives**

| Name | Type | Description |
|---|---|---|
| char8Value | Char8 | 8 bit character value. |
| boolValue | Bool | Boolean with true and false. |
| int8Value | Int8 | 8 bit signed integer value. |
| uInt8Value | UInt8 | 8 bit unsigned integer value. |
| int16Value | Int16 | 16 bit signed integer value. |
| uInt16Value | UInt16 | 16 bit unsigned integer value. |
| int32Value | Int32 | 32 bit signed integer value. |
| uInt32Value | UInt32 | 32 bit unsigned integer value. |
| int64Value | Int64 | 64 bit signed integer value. |
| uInt64Value | UInt64 | 64 bit unsigned integer value. |
| float32Value | Float32 | 32 bit single-precision floating-point value. |
| float64Value | Float64 | 64 bit double-precision floating-point value. |
| durationValue | Duration | Duration in nanoseconds. |
| dateTimeValue | DateTime | Absolute time in nanoseconds. |
| string8Value | String8 | 8 bit character string value. |

## 2.3 Universally Unique Identifiers

For a unique identification of types (and hence models), SMP uses Universally Unique Identifiers with the format specified by the Open Group (http://www.opengroup.org).

### 2.3.1 Uuid

Universally Unique Identifier.

A Universally Unique Identifier is 128 bit long, separated into 32 hex nibbles (where each hex nibble stands for 4 bits).

**File**

#include "Smp/Platform.h"

**Namespace**

Smp

**Declaration of Uuid**

```
/// Universally Unique Identifier.
/// For a unique identification of types (and hence models), SMP uses
/// Universally Unique Identifiers with the format specified by the Open
/// Group (http://www.opengroup.org).
struct Uuid
{
    /// 8 hex nibbles.
    Smp::NT_UInt32 Data1;
```

```
        /// 4 hex nibbles.
        Smp::NT_UInt16 Data2;

        /// 4 hex nibbles.
        Smp::NT_UInt16 Data3;

        /// 4+12 hex nibbles.
        Smp::UuidBytes Data4;
    };
```

**Fields**

| Name | Type | Description |
|---|---|---|
| Data1 | UInt32 | 8 hex nibbles. |
| Data2 | UInt16 | 4 hex nibbles. |
| Data3 | UInt16 | 4 hex nibbles. |
| Data4 | UuidBytes | 4+12 hex nibbles. |

### 2.3.2 Uuid Bytes

Final 8 bytes of Uuid.

**File**

#include "Smp/Platform.h"

**Namespace**

Smp

**Declaration of UuidBytes**

```
    /// Final 8 bytes of Uuid.
    typedef Smp::NT_UInt8 UuidBytes[8];
```

**Table 2 - Specification of UuidBytes**

| Item type | Size |
|---|---|
| UInt8 | 8 |

# 3
# Component Model

## 3.1 Exceptions

SMP defines some basic exceptions which are used in several interfaces, and which are therefore defined outside of an individual interface. For each exception, see the detailed specification of the interfaces to find out which methods actually may raise this exception.

### 3.1.1 Exception

This is the base class for all SMP exceptions.

This exception is the base class for all other SMP exceptions. It provides Name, Description and Message.

**File**

#include "Smp/Exceptions.h"

**Namespace**

Smp

**Declaration of Exception**

```
/// This is the base class for all SMP exceptions.
/// This exception is the base class for all other SMP exceptions. It
/// provides Name, Description and Message.
class Exception
{
protected:
    /// Constructor for new exception.
    /// @param   name Name of the exception that is returned by GetName.
    /// @param   description Description of the exception that is returned
    ///          by GetDescription.
    Exception(
        Smp::String8 name,
        Smp::String8 description) throw();

    /// Copy constructor.
    Exception(
        Exception& ex) throw();

    /// Virtual destructor to release memory.
    virtual ~Exception();

    /// Name of the exception that is returned by GetName.
    Smp::String8 name;

    /// Description of the exception that is returned by GetDescription.
    Smp::String8 description;

    /// Description of the problem that is returned by GetMessage.
    Smp::String8 message;
```

```
public:
    /// Returns the name of the exception class. This name can be used e.g.
    /// for debugging purposes.
    /// @return  Name of the exception class.
    virtual Smp::String8 GetName() const;

    /// Returns a textual description of the exception class. This
    /// description can be used e.g. for debugging purposes.
    /// @return  Textual description of the exception class.
    virtual Smp::String8 GetDescription() const;

    /// Returns the description of the problem encountered. This message
    /// can be used e.g. for debugging purposes.
    /// @return  Textual description of the problem encountered.
    virtual Smp::String8 GetMessage() const;

protected:
    /// Defines the description of the problem encountered.
    /// This message can be used e.g.
    /// for debugging purposes.
    /// @param   message Textual description of the problem encountered.
    virtual void SetMessage(Smp::String8 message);

    /// Assignment operator.
    Exception& operator =(const Exception& ex);
};
```

**Base Exceptions**

None

**Fields**

| Name | Type | Description |
|---|---|---|
| description | String8 | Description of the exception that is returned by GetDescription. |
| message | String8 | Description of the problem that is returned by GetMessage. |
| name | String8 | Name of the exception that is returned by GetName. |

### 3.1.2 Duplicate Name

This exception is raised when trying to add an object to a collection of objects, which have to have unique names, but another object with the same name does exist already in this collection. This would lead to duplicate names.

**File**

#include "Smp/Exceptions.h"

**Namespace**

Smp

**Declaration of DuplicateName**

```
/// This exception is raised when trying to add an object to a collection
/// of objects, which have to have unique names, but another object with
/// the same name does exist already in this collection. This would lead to
/// duplicate names.
```

```
class DuplicateName : public Smp::Exception
{
public:
    /// Constructor for new exception.
    /// @param   duplicateName Name that already exists in the collection.
    DuplicateName(
        Smp::String8 duplicateName) throw();

    /// Copy constructor.
    DuplicateName(
        DuplicateName& ex) throw();

    /// Virtual destructor to release memory.
    virtual ~DuplicateName();

    /// Name that already exists in the collection.
    Smp::String8 duplicateName;
};
```

**Base Exceptions**

Smp::Exception

**Fields**

| Name | Type | Description |
|---|---|---|
| duplicateName | String8 | Name that already exists in the collection. |

### 3.1.3 Invalid Any Type

This exception is raised when trying to use an AnySimple argument of wrong type.

**File**

#include "Smp/Exceptions.h"

**Namespace**

Smp

**Declaration of InvalidAnyType**

```
/// This exception is raised when trying to use an AnySimple argument of
/// wrong type.
/// @remarks This can happen when assigning a value to an AnySimple
///          instance, but as well when e.g. registering an event sink with
///          an event source of another event argument type.
class InvalidAnyType : public Smp::Exception
{
public:
    /// Constructor for new exception.
    /// @param   invalidType Type that is not valid.
    /// @param   expectedType Type that was expected.
    InvalidAnyType(
        Smp::PrimitiveTypeKind invalidType,
        Smp::PrimitiveTypeKind expectedType) throw();

    /// Copy constructor.
    InvalidAnyType(
        InvalidAnyType& ex) throw();

    /// Virtual destructor to release memory.
    virtual ~InvalidAnyType();
```

```
        /// Type that is not valid.
        Smp::PrimitiveTypeKind invalidType;

        /// Type that was expected.
        Smp::PrimitiveTypeKind expectedType;
    };
```

***Remark:*** This can happen when assigning a value to an AnySimple instance, but as well when e.g. registering an event sink with an event source of another event argument type.

**Base Exceptions**

Smp::Exception

**Fields**

| Name | Type | Description |
|---|---|---|
| expectedType | PrimitiveTypeKind | Type that was expected. |
| invalidType | PrimitiveTypeKind | Type that is not valid. |

### 3.1.4 Invalid Event Id

This exception is raised when an invalid event id is provided, e.g. when calling Subscribe(), Unsubscribe() or Emit() of the Event Manager (using an invalid global event id), or when calling SetEventSimulationTime(), SetEventMissionTime(), SetEventEpochTime(), SetEventZuluTime(), SetEventCycleTime(), SetEventCount() or RemoveEvent() of the Scheduler (using an invalid scheduler event id).

**File**

#include "Smp/Exceptions.h"

**Namespace**

Smp::Services

**Declaration of InvalidEventId**

```
        /// This exception is raised when an invalid event id is provided, e.g.
        /// when calling Subscribe(), Unsubscribe() or Emit() of the Event
        /// Manager (using an invalid global event id), or when calling
        /// SetEventSimulationTime(), SetEventMissionTime(),
        /// SetEventEpochTime(), SetEventZuluTime(), SetEventCycleTime(),
        /// SetEventCount() or RemoveEvent() of the Scheduler (using an invalid
        /// scheduler event id).
        class InvalidEventId : public Smp::Exception
        {
        public:
            /// Constructor for new exception.
            /// @param   eventId Invalid event identifier.
            InvalidEventId(
                Smp::Services::EventId eventId) throw();

            /// Copy constructor.
            InvalidEventId(
                InvalidEventId& ex) throw();

            /// Virtual destructor to release memory.
            virtual ~InvalidEventId();
```

```
            /// Invalid event identifier.
            Smp::Services::EventId eventId;
        };
```

**Base Exceptions**

Smp::Exception

**Fields**

| Name | Type | Description |
|---|---|---|
| eventId | EventId | Invalid event identifier. |

### 3.1.5 Invalid Object Type

This exception is raised when trying to pass an object of wrong type.

**File**

#include "Smp/Exceptions.h"

**Namespace**

Smp

**Declaration of InvalidObjectType**

```
    /// This exception is raised when trying to pass an object of wrong type.
    /// @remarks This can happen when adding a component to a container or
    ///          reference which is semantically typed by a specific type
    ///          implementing IComponent.
    class InvalidObjectType : public Smp::Exception
    {
    public:
        /// Constructor for new exception.
        /// @param   invalidObject Object that is not of valid type.
        InvalidObjectType(
            Smp::IObject* invalidObject) throw();

        /// Copy constructor.
        InvalidObjectType(
            InvalidObjectType& ex) throw();

        /// Virtual destructor to release memory.
        virtual ~InvalidObjectType();

        /// Object that is not of valid type.
        Smp::IObject* invalidObject;
    };
```

***Remark:*** This can happen when adding a component to a container or reference which is semantically typed by a specific type implementing IComponent.

**Base Exceptions**

Smp::Exception

**Fields**

| Name | Type | Description |
|---|---|---|
| invalidObject | IObject | Object that is not of valid type. |

## 3.2 Objects and Components

In SMP, a simulation is composed of components, where models, services, and the simulation environment all implement a common base interface. Other elements in SMP are not components, but only objects.

### 3.2.1 Objects

Objects are elements of the simulation which provide name and description.

#### 3.2.1.1 IObject

This interface is the base interface for almost all other SMP interfaces. While most interfaces derive from IComponent, which itself is derived from IObject, some objects (including IField, IFailure, IEntryPoint, IEventSink, IEventSource, IContainer and IReference) are directly derived from IObject.

**File**

#include "Smp/IObject.h"

**Namespace**

Smp

**Declaration of IObject**

```
/// Unique Identifier of type IObject.
extern const Uuid Uuid_IObject;

/// This interface is the base interface for almost all other SMP
/// interfaces. While most interfaces derive from IComponent, which itself
/// is derived from IObject, some objects (including IField, IFailure,
/// IEntryPoint, IEventSink, IEventSource, IContainer and IReference) are
/// directly derived from IObject.
/// @remarks The two methods of this interface ensure that all SMP objects
///          can be shown with a name, and with an optional description.
class IObject
{
public:

    /// Virtual destructor to release memory.
    virtual ~IObject() {}

    /// Return the name of the object ("property getter").
    /// Names must
    ///      - be unique within their context,
    ///      - not be empty,
    ///      - start with a letter, and
    ///      - only contain letters, digits, the underscore ("_") and
    /// brackets ("[" and "]").
    ///
    /// @remarks Applications may display the name as user readable object
    ///          identification.
    /// @remarks It is recommended that names do not exceed 32 characters
    ///          in size.
    /// @return  Name of object.
    virtual Smp::String8 GetName() const = 0;

    /// Return the description of the object ("property getter").
    /// Descriptions are optional and may be empty.
```

```
        /// @remarks Applications may display the description as additional
        ///          information about the object.
        /// @return  Description of object.
        virtual Smp::String8 GetDescription() const = 0;
    };
```

***Remark:*** The two methods of this interface ensure that all SMP objects can be shown with a name, and with an optional description.

**Base Interfaces**

None

**Operations**

| Name | Description |
|---|---|
| GetDescription | Return the description of the object ("property getter"). |
| GetName | Return the name of the object ("property getter"). |

#### 3.2.1.1.1 Get Description

Return the description of the object ("property getter").

Descriptions are optional and may be empty.

***Remark:*** Applications may display the description as additional information about the object.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | String8 | Description of object. |

**Exceptions**

None

#### 3.2.1.1.2 Get Name

Return the name of the object ("property getter").

Names must

- be unique within their context,
- not be empty,
- start with a letter, and
- only contain letters, digits, the underscore ("_") and brackets ("[" and "]").

***Remark:*** Applications may display the name as user readable object identification.

***Remark:*** It is recommended that names do not exceed 32 characters in size.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | String8 | Name of object. |

**Exceptions**

None

### 3.2.2 Components

Most elements in SMP are components, which implement the IComponent interface.

The three most important component types are models, services, and the simulator. The first two of these interfaces are introduced in this section, while the ISimulator interface is explained in section 3.6.1.2.

#### 3.2.2.1 IComponent

This is the base interface for all SMP components.

**File**

#include "Smp/IComponent.h"

**Namespace**

Smp

**Declaration of IComponent**

```
    /// Unique Identifier of type IComponent.
    extern const Uuid Uuid_IComponent;

    /// This is the base interface for all SMP components.
    /// @remarks SMP components are typically models and services.
    class IComponent :
        public virtual Smp::IObject
    {
    public:

        /// Virtual destructor to release memory.
        virtual ~IComponent() {}

        /// Return the parent component of the component ("property getter").
        /// Components link to their parent to allow traversing the tree of
        /// components upwards to the root component of a composition.
        /// @remarks Typically, only the simulator itself is a root component,
        ///          so all other components should have a parent component.
        /// @return  Parent component of component or null if component has no
        ///          parent.
        virtual Smp::IComposite* GetParent() const = 0;

        /// Query for an interface specified by its Uuid.
        /// @param   interfaceId Uuid of interface to query for.
        /// @param   result Interface to component or NULL if component does
        ///          not implement the requested interface.
        /// @return  True if the interface is implemented, false otherwise.
        virtual Smp::Bool QueryInterface(Smp::Uuid interfaceId, void* result) const
= 0;
    };
```

***Remark:*** SMP components are typically models and services.

**Base Interfaces**

Smp::IObject

**Operations**

| Name | Description |
|---|---|
| GetParent | Return the parent component of the component ("property getter"). |
| QueryInterface | Query for an interface specified by its Uuid. |

#### 3.2.2.1.1 Get Parent

Return the parent component of the component ("property getter").

Components link to their parent to allow traversing the tree of components upwards to the root component of a composition.

***Remark:*** Typically, only the simulator itself is a root component, so all other components should have a parent component.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | IComposite | Parent component of component or null if component has no parent. |

**Exceptions**

None

#### 3.2.2.1.2 Query Interface

Query for an interface specified by its Uuid.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| interfaceId | in | Uuid | Uuid of interface to query for. |
| result | inout | void* | Interface to component or NULL if component does not implement the requested interface. |
| | return | Bool | True if the interface is implemented, false otherwise. |

**Exceptions**

None

### 3.2.2.2 Component Collection

A component collection is an ordered collection of components, which allows iterating all members.

This type is platform specific. For details see the SMP Platform Mappings.

**File**

#include "Smp/IComponent.h"

**Namespace**

Smp

**Declaration of ComponentCollection**

```
/// Unique Identifier of type ComponentCollection.
extern const Uuid Uuid_ComponentCollection;

/// A component collection is an ordered collection of components, which
/// allows iterating all members.
/// This type is platform specific. For details see the SMP Platform
/// Mappings.
typedef std::vector<IComponent*> ComponentCollection;
```

#### 3.2.2.3 Model State Kind

This is an enumeration of the available states of a model. Each model is always in one of these four model states.

**File**

#include "Smp/IModel.h"

**Namespace**

Smp

**Declaration of ModelStateKind**

```
/// Unique Identifier of type ModelStateKind.
extern const Uuid Uuid_ModelStateKind;

/// This is an enumeration of the available states of a model. Each model
/// is always in one of these four model states.
enum ModelStateKind
{
    /// The Created state is the initial state of a model. Model creation
    /// is done by an external mechanism, e.g. by factories.
    /// This state is entered automatically after the model has been
    /// created.
    /// This state is left via the Publish() state transition.
    MSK_Created,

    /// In Publishing state, the model is allowed to publish features. This
    /// includes publication of fields, operations and properties. In
    /// addition, the model is allowed to create other models.
    /// This state is entered via the Publish() state transition.
    /// This state is left via the Configure() state transition.
    MSK_Publishing,

    /// In Configured state, the model has been fully configured. This
    /// configuration may be done by external components, or internally by
    /// the model itself, e.g. by reading data from an external source.
    /// This state is entered via the Configure() state transition.
    /// This state is left via the Connect() state transition.
    MSK_Configured,

    /// In Connected state, the model is connected to the simulator. In
    /// this state, neither publication nor creation of other models is
    /// allowed anymore.
    /// This state is entered via the Connect() state transition.
    /// This is the final state of a model, and only left on termination.
    MSK_Connected
};
```

**Table 3 - Enumeration Literals of ModelStateKind**

| Name | Description |
|---|---|
| MSK_Created | The Created state is the initial state of a model. Model creation is done by an external mechanism, e.g. by factories.<br>This state is entered automatically after the model has been created.<br>This state is left via the Publish() state transition. |
| MSK_Publishing | In Publishing state, the model is allowed to publish features. This includes publication of fields, operations and properties. In addition, the model is allowed to create other models.<br>This state is entered via the Publish() state transition.<br>This state is left via the Configure() state transition. |
| MSK_Configured | In Configured state, the model has been fully configured. This configuration may be done by external components, or internally by the model itself, e.g. by reading data from an external source.<br>This state is entered via the Configure() state transition.<br>This state is left via the Connect() state transition. |
| MSK_Connected | In Connected state, the model is connected to the simulator. In this state, neither publication nor creation of other models is allowed anymore.<br>This state is entered via the Connect() state transition.<br>This is the final state of a model, and only left on termination. |

#### 3.2.2.4 IModel

Interface for a model.

All SMP models implement this interface. As models interface to the simulation environment, they have a dependency to it via the two interfaces IPublication and ISimulator.

This is the only mandatory interface models have to implement. All other interfaces (component and model mechanisms and managed interfaces) are optional.

**File**

#include "Smp/IModel.h"

**Namespace**

Smp

**Declaration of IModel**

```
/// Unique Identifier of type IModel.
extern const Uuid Uuid_IModel;

/// Interface for a model.
/// All SMP models implement this interface. As models interface to the
/// simulation environment, they have a dependency to it via the two
/// interfaces IPublication and ISimulator.
/// This is the only mandatory interface models have to implement. All
/// other interfaces (component and model mechanisms and managed
/// interfaces) are optional.
class IModel :
    public virtual Smp::IComponent
{
public:

    /// Virtual destructor to release memory.
    virtual ~IModel() {}

    /// Returns the state the model is currently in.
    /// The model state can be changed using the Publish(), Configure() and
    /// Connect() state transition methods.
    /// @return  Current model state.
    virtual Smp::ModelStateKind GetState() const = 0;

    /// Request the model to publish its fields, properties and operations
    /// against the provided publication receiver.
    /// This method can only be called once for each model, and only when
    /// the model is in the Created state. The method raises an
    /// InvalidModelState exception if the model is not in Created state.
    /// When this operation is called, the model immediately enters the
    /// Publishing state, before it publishes any of its features.
    /// @remarks The simulation environment typically calls this method in
    ///          the Building state.
    /// @param   receiver Publication receiver.
    /// @throws  Smp::IModel::InvalidModelState
    virtual void Publish(Smp::IPublication* receiver) throw (
        Smp::IModel::InvalidModelState) = 0;

    /// Request the model to perform any custom configuration. The model
    /// can create and configure other models using the field values of its
    /// published fields.
    /// This method can only be called once for each model, and only when
    /// the model is in Publishing state. The method raises an
    /// InvalidModelState exception if the model is not in Publishing
    /// state.
    /// The model can still publish further features in this call, and can
    /// even create other models, but at the end of this call, it needs to
    /// enter the Configured state.
    /// @remarks The simulation environment typically calls this method in
    ///          the Building state.
    /// @param   logger Logger service for logging of error messages during
    ///          configuration.
    /// @throws  Smp::IModel::InvalidModelState
    virtual void Configure(Smp::Services::ILogger* logger) throw (
        Smp::IModel::InvalidModelState) = 0;

    /// Allow the model to connect to the simulator.
    /// This method can only be called once for each model, and only when
    /// the model is in the Configured state. The method raises an
    /// InvalidModelState exception if the model is not in Configured
    /// state.
    /// When this operation is called, the model immediately enters the
    /// Connected state, before it uses any of the simulator methods and
    /// services.
    /// In this method, the model may query for and use any of the
    /// available simulation services, as they are all guaranteed to be
    /// fully functional at that time. It may as well connect to other
```

```
        /// models' functionality (e.g. to event sources), as it is guaranteed
        /// that all models have been created and configured before the
        /// Connect() method of any model is called.
        /// @remarks The simulation environment typically calls this method in
        ///          the Connecting state.
        /// @param   simulator Simulation Environment that hosts the model.
        /// @throws  Smp::IModel::InvalidModelState
        virtual void Connect(Smp::ISimulator* simulator) throw (
            Smp::IModel::InvalidModelState) = 0;
    };
```

**Base Interfaces**

Smp::IComponent

**Operations**

| Name | Description |
|---|---|
| Configure | Request the model to perform any custom configuration. The model can create and configure other models using the field values of its published fields. |
| Connect | Allow the model to connect to the simulator. |
| GetState | Returns the state the model is currently in. |
| Publish | Request the model to publish its fields, properties and operations against the provided publication receiver. |

#### 3.2.2.4.1 Configure

Request the model to perform any custom configuration. The model can create and configure other models using the field values of its published fields.

This method can only be called once for each model, and only when the model is in Publishing state. The method raises an InvalidModelState exception if the model is not in Publishing state.

The model can still publish further features in this call, and can even create other models, but at the end of this call, it needs to enter the Configured state.

***Remark:*** The simulation environment typically calls this method in the Building state.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| logger | in | ILogger | Logger service for logging of error messages during configuration. |

**Exceptions**

Smp::IModel::InvalidModelState

#### 3.2.2.4.2 Connect

Allow the model to connect to the simulator.

This method can only be called once for each model, and only when the model is in the Configured state. The method raises an InvalidModelState exception if the model is not in Configured state.

When this operation is called, the model immediately enters the Connected state, before it uses any of the simulator methods and services.

In this method, the model may query for and use any of the available simulation services, as they are all guaranteed to be fully functional at that time. It may as well connect to other models' functionality (e.g. to event sources), as it is guaranteed that all models have been created and configured before the Connect() method of any model is called.

***Remark:*** The simulation environment typically calls this method in the Connecting state.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| simulator | in | ISimulator | Simulation Environment that hosts the model. |

**Exceptions**

Smp::IModel::InvalidModelState

#### 3.2.2.4.3 Get State

Returns the state the model is currently in.

The model state can be changed using the Publish(), Configure() and Connect() state transition methods.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | ModelStateKind | Current model state. |

**Exceptions**

None

#### 3.2.2.4.4 Publish

Request the model to publish its fields, properties and operations against the provided publication receiver.

This method can only be called once for each model, and only when the model is in the Created state. The method raises an InvalidModelState exception if the model is not in Created state.

When this operation is called, the model immediately enters the Publishing state, before it publishes any of its features.

***Remark:*** The simulation environment typically calls this method in the Building state.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| receiver | in | IPublication | Publication receiver. |

**Exceptions**

Smp::IModel::InvalidModelState

#### 3.2.2.4.5 Invalid Model State

This exception is raised by a model when one of the state transition commands is called in an invalid state.

**File**

#include "Smp/IModel.h"

**Namespace**

Smp::IModel

**Declaration of InvalidModelState**

```
/// This exception is raised by a model when one of the state
/// transition commands is called in an invalid state.
class InvalidModelState : public Smp::Exception
{
public:
    /// Constructor for new exception.
    /// @param   invalidState State that is not valid.
    /// @param   expectedState State that was expected.
    InvalidModelState(
        Smp::ModelStateKind invalidState,
        Smp::ModelStateKind expectedState) throw();

    /// Copy constructor.
    InvalidModelState(
        InvalidModelState& ex) throw();

    /// Virtual destructor to release memory.
    virtual ~InvalidModelState();

    /// State that is not valid.
    Smp::ModelStateKind invalidState;

    /// State that was expected.
    Smp::ModelStateKind expectedState;
};
```

**Fields**

| Name | Type | Description |
|---|---|---|
| expectedState | ModelStateKind | State that was expected. |
| invalidState | ModelStateKind | State that is not valid. |

### 3.2.2.5 Model Collection

A model collection is an ordered collection of models, which allows iterating all members.

This type is platform specific. For details see the SMP Platform Mappings.

**File**

#include "Smp/IModel.h"

**Namespace**

Smp

**Declaration of ModelCollection**

```
/// Unique Identifier of type ModelCollection.
extern const Uuid Uuid_ModelCollection;

/// A model collection is an ordered collection of models, which allows
/// iterating all members.
/// This type is platform specific. For details see the SMP Platform
/// Mappings.
typedef std::vector<IModel*> ModelCollection;
```

#### 3.2.2.6 IService

Base interface for all SMP services.

**File**

#include "Smp/IService.h"

**Namespace**

Smp

**Declaration of IService**

```
/// Unique Identifier of type IService.
extern const Uuid Uuid_IService;

/// Base interface for all SMP services.
/// @remarks Currently, this interface does not add any functionality.
class IService :
    public virtual Smp::IComponent
{
};
```

***Remark:*** Currently, this interface does not add any functionality.

**Base Interfaces**

Smp::IComponent

**Operations**

None

#### 3.2.2.7 Service Collection

A service collection is an ordered collection of services, which allows iterating all members.

This type is platform specific. For details see the SMP Platform Mappings.

**File**

#include "Smp/IService.h"

**Namespace**

Smp

**Declaration of ServiceCollection**

```
/// Unique Identifier of type ServiceCollection.
extern const Uuid Uuid_ServiceCollection;

/// A service collection is an ordered collection of services, which allows
/// iterating all members.
/// This type is platform specific. For details see the SMP Platform
/// Mappings.
typedef std::vector<IService*> ServiceCollection;
```

## 3.3 Component Mechanisms

While the IComponent base interface provides mechanisms to get name, description, and parent, it does not allow specifying further relations between components. The mechanisms supported by SMP are aggregation, composition, inter-component events via event sources and event sinks, dynamic invocation and persistence.

### 3.3.1 Aggregation

Via aggregation, a component can **reference** other components in the component hierarchy to use their methods. As opposed to composition, an aggregated component is not owned, but only referenced.

#### 3.3.1.1 IAggregate

Interface for an aggregate component.

A component with references to other components implements this interface. Referenced components are held in named references.

**File**

#include "Smp/IAggregate.h"

**Namespace**

Smp

**Declaration of IAggregate**

```
/// Unique Identifier of type IAggregate.
extern const Uuid Uuid_IAggregate;

/// Interface for an aggregate component.
/// A component with references to other components implements this
/// interface. Referenced components are held in named references.
/// @remarks This interface represents the Aggregation mechanism in the SMP
///          Metamodel (via References). In UML 2.0, this is represented by
///          a required interface.
class IAggregate :
    public virtual Smp::IComponent
{
public:

    /// Virtual destructor to release memory.
    virtual ~IAggregate() {}

    /// Query for the collection of all references of the aggregate
    /// component.
    /// The returned collection may be empty if no references exist for the
```

```
        /// aggregate.
        /// @return  Collection of references.
        virtual const Smp::ReferenceCollection* GetReferences() const = 0;

        /// Query for a reference of this aggregate component by its name.
        /// The returned reference may be null if no reference with the given
        /// name could be found. If more than one reference with this name
        /// exists, it is not defined which one is returned.
        /// @param   name Reference name.
        /// @return  Reference queried for by name, or null if no reference
        ///          with this name exists.
        virtual Smp::IReference* GetReference(Smp::String8 name) const = 0;
    };
```

***Remark:*** This interface represents the Aggregation mechanism in the SMP Metamodel (via References). In UML 2.0, this is represented by a required interface.

**Base Interfaces**

Smp::IComponent

**Operations**

| Name | Description |
|---|---|
| GetReference | Query for a reference of this aggregate component by its name. |
| GetReferences | Query for the collection of all references of the aggregate component. |

#### 3.3.1.1.1 Get Reference

Query for a reference of this aggregate component by its name.

The returned reference may be null if no reference with the given name could be found. If more than one reference with this name exists, it is not defined which one is returned.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | IReference | Reference queried for by name, or null if no reference with this name exists. |
| name | in | String8 | Reference name. |

**Exceptions**

None

#### 3.3.1.1.2 Get References

Query for the collection of all references of the aggregate component.

The returned collection may be empty if no references exist for the aggregate.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | ReferenceCollection | Collection of references. |

**Exceptions**

None

#### 3.3.1.2 IReference

Interface for a reference.

A reference allows querying for the referenced components.

**File**

#include "Smp/IReference.h"

**Namespace**

Smp

**Declaration of IReference**

```
/// Unique Identifier of type IReference.
extern const Uuid Uuid_IReference;

/// Interface for a reference.
/// A reference allows querying for the referenced components.
/// @remarks References are used together with the IAggregate interface for
///          aggregation.
class IReference :
    public virtual Smp::IObject
{
public:

    /// Virtual destructor to release memory.
    virtual ~IReference() {}

    /// Query for the collection of all referenced components.
    /// The returned collection may be empty if no components are
    /// referenced.
    /// @return  Collection of referenced components.
    virtual const Smp::ComponentCollection* GetComponents() const = 0;

    /// Query for a referenced component by its name.
    /// The returned component may be null if no component with the given
    /// name could be found.
    /// @param   name Component name.
    /// @return  Referenced component with the given name, or null if no
    ///          referenced component with the given name could be found.
    virtual Smp::IComponent* GetComponent(Smp::String8 name) const = 0;
};
```

***Remark:*** References are used together with the IAggregate interface for aggregation.

**Base Interfaces**

Smp::IObject

**Operations**

| Name | Description |
|---|---|
| GetComponent | Query for a referenced component by its name. |
| GetComponents | Query for the collection of all referenced components. |

#### 3.3.1.2.1 Get Component

Query for a referenced component by its name.

The returned component may be null if no component with the given name could be found.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | IComponent | Referenced component with the given name, or null if no referenced component with the given name could be found. |
| name | in | String8 | Component name. |

**Exceptions**

None

#### 3.3.1.2.2 Get Components

Query for the collection of all referenced components.

The returned collection may be empty if no components are referenced.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | ComponentCollection | Collection of referenced components. |

**Exceptions**

None

### 3.3.1.3 Reference Collection

A reference collection is an ordered collection of references, which allows iterating all members.

This type is platform specific. For details see the SMP Platform Mappings.

**File**

#include "Smp/IReference.h"

**Namespace**

Smp

**Declaration of ReferenceCollection**

```
/// Unique Identifier of type ReferenceCollection.
extern const Uuid Uuid_ReferenceCollection;

/// A reference collection is an ordered collection of references, which
/// allows iterating all members.
/// This type is platform specific. For details see the SMP Platform
/// Mappings.
typedef std::vector<IReference*> ReferenceCollection;
```

### 3.3.2 Composition

Via composition, a component can **contain** other components in the component hierarchy. As opposed to aggregation, a component is owned, and its life-time coincides with its parent component. Composition is the counter-part to the GetParent() method of the IComponent interface and allows traversing the tree of components in downward direction.

#### 3.3.2.1 IComposite

Interface for a composite component.

A component with children implements this interface. Child components are held in named containers.

**File**

#include "Smp/IComposite.h"

**Namespace**

Smp

**Declaration of IComposite**

```
/// Unique Identifier of type IComposite.
extern const Uuid Uuid_IComposite;

/// Interface for a composite component.
/// A component with children implements this interface. Child components
/// are held in named containers.
/// @remarks This interface represents the Composition mechanism in the SMP
///          Metamodel (via Containers). In UML 2.0, this is represented by
///          composite aggregation.
class IComposite :
    public virtual Smp::IComponent
{
public:

    /// Virtual destructor to release memory.
    virtual ~IComposite() {}

    /// Query for the collection of all containers of the composite
    /// component.
    /// The returned collection may be empty if no containers exist for the
    /// composite.
    /// @return  Collection of containers.
    virtual const Smp::ContainerCollection* GetContainers() const = 0;

    /// Query for a container of this composite component by its name.
    /// The returned container may be null if no container with the given
    /// name could be found.
    /// @param   name Container name.
    /// @return  Container queried for by name, or null if no container
    ///          with this name exists.
    virtual Smp::IContainer* GetContainer(Smp::String8 name) const = 0;
};
```

***Remark:*** This interface represents the Composition mechanism in the SMP Metamodel (via Containers). In UML 2.0, this is represented by composite aggregation.

**Base Interfaces**

Smp::IComponent

**Operations**

| Name | Description |
|---|---|
| GetContainer | Query for a container of this composite component by its name. |
| GetContainers | Query for the collection of all containers of the composite component. |

#### 3.3.2.1.1 Get Container

Query for a container of this composite component by its name.

The returned container may be null if no container with the given name could be found.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | IContainer | Container queried for by name, or null if no container with this name exists. |
| name | in | String8 | Container name. |

**Exceptions**

None

#### 3.3.2.1.2 Get Containers

Query for the collection of all containers of the composite component.

The returned collection may be empty if no containers exist for the composite.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | ContainerCollection | Collection of containers. |

**Exceptions**

None

### 3.3.2.2 IContainer

Interface for a container.

A container allows querying for its children.

Containers are used together with the IComposite interface for composition.

**File**

#include "Smp/IContainer.h"

**Namespace**

Smp

**Declaration of IContainer**

```
/// Unique Identifier of type IContainer.
extern const Uuid Uuid_IContainer;

/// Interface for a container.
/// A container allows querying for its children.
/// Containers are used together with the IComposite interface for
/// composition.
class IContainer :
    public virtual Smp::IObject
{
public:

    /// Virtual destructor to release memory.
    virtual ~IContainer() {}

    /// Query for the collection of all components in the container.
    /// The returned collection may be empty if no components exist for the
    /// container.
    /// @return  Collection of contained components.
    virtual const Smp::ComponentCollection* GetComponents() const = 0;

    /// Query for a component contained in the container by name.
    /// The returned component may be null if no child with the given name
    /// could be found.
    /// @param   name Child name.
    /// @return  Child component, or null if no child component with the
    ///          given name exists.
    virtual Smp::IComponent* GetComponent(Smp::String8 name) const = 0;
};
```

**Base Interfaces**

Smp::IObject

**Operations**

| Name | Description |
|---|---|
| GetComponent | Query for a component contained in the container by name. |
| GetComponents | Query for the collection of all components in the container. |

#### 3.3.2.2.1 Get Component

Query for a component contained in the container by name.

The returned component may be null if no child with the given name could be found.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | IComponent | Child component, or null if no child component with the given name exists. |
| name | in | String8 | Child name. |

**Exceptions**

None

#### 3.3.2.2.2 Get Components

Query for the collection of all components in the container.

The returned collection may be empty if no components exist for the container.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | ComponentCollection | Collection of contained components. |

**Exceptions**

None

#### 3.3.2.3 Container Collection

A container collection is an ordered collection of containers, which allows iterating all members.

This type is platform specific. For details see the SMP Platform Mappings.

**File**

#include "Smp/IContainer.h"

**Namespace**

Smp

**Declaration of ContainerCollection**

```
/// Unique Identifier of type ContainerCollection.
extern const Uuid Uuid_ContainerCollection;

/// A container collection is an ordered collection of containers, which
/// allows iterating all members.
/// This type is platform specific. For details see the SMP Platform
/// Mappings.
typedef std::vector<IContainer*> ContainerCollection;
```

### 3.3.3 Events

Events are used in event-based programming. Event-based programming works via event sources and event sinks that can be registered to and unregistered from event sources. When an event source emits an event, it notifies all subscribed event sinks.

#### 3.3.3.1 IEvent Sink

Interface of an event sink that can be subscribed to an event source (IEventSource).

This interface provides a notification method (event handler) that can be called by event sources when an event is emitted.

**File**

#include "Smp/IEventSink.h"

**Namespace**

Smp

**Declaration of IEventSink**

```
/// Unique Identifier of type IEventSink.
extern const Uuid Uuid_IEventSink;

/// Interface of an event sink that can be subscribed to an event source
/// (IEventSource).
/// This interface provides a notification method (event handler) that can
/// be called by event sources when an event is emitted.
class IEventSink :
    public virtual Smp::IObject
{
public:

    /// Virtual destructor to release memory.
    virtual ~IEventSink() {}

    /// This method returns the Component that owns the event sink.
    /// @remarks This is required to be able to store and restore event
    ///          sinks.
    /// @return  Owner of event sink.
    virtual Smp::IComponent* GetOwner() const = 0;

    /// This event handler method is called when an event is emitted.
    /// Components providing event sinks must ensure that these event sinks
    /// do not throw exceptions.
    /// @param   sender Object emitting the event.
    /// @param   arg Event argument.
    virtual void Notify(Smp::IObject* sender, Smp::AnySimple arg) = 0;
};
```

**Base Interfaces**

Smp::IObject

**Operations**

| Name | Description |
|---|---|
| GetOwner | This method returns the Component that owns the event sink. |
| Notify | This event handler method is called when an event is emitted. |

#### 3.3.3.1.1 Get Owner

This method returns the Component that owns the event sink.

***Remark:*** This is required to be able to store and restore event sinks.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | IComponent | Owner of event sink. |

**Exceptions**

None

#### 3.3.3.1.2 Notify

This event handler method is called when an event is emitted.

Components providing event sinks must ensure that these event sinks do not throw exceptions.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| sender | in | IObject | Object emitting the event. |
| arg | in | AnySimple | Event argument. |

**Exceptions**

None

### 3.3.3.2 Event Sink Collection

An event sink collection is an ordered collection of event sinks, which allows iterating all members.

This type is platform specific. For details see the SMP Platform Mappings.

**File**

#include "Smp/IEventSink.h"

**Namespace**

Smp

**Declaration of EventSinkCollection**

```
/// Unique Identifier of type EventSinkCollection.
extern const Uuid Uuid_EventSinkCollection;

/// An event sink collection is an ordered collection of event sinks, which
/// allows iterating all members.
/// This type is platform specific. For details see the SMP Platform
/// Mappings.
typedef std::vector<IEventSink*> EventSinkCollection;
```

### 3.3.3.3 IEvent Source

Interface of an event source that event sinks (IEventSink) can subscribe to.

This interface allows event consumers to subscribe to or unsubscribe from an event.

**File**

#include "Smp/IEventSource.h"

**Namespace**

Smp

**Declaration of IEventSource**

```
/// Unique Identifier of type IEventSource.
extern const Uuid Uuid_IEventSource;

/// Interface of an event source that event sinks (IEventSink) can
/// subscribe to.
/// This interface allows event consumers to subscribe to or unsubscribe
/// from an event.
class IEventSource :
    public virtual Smp::IObject
{
public:

    /// Virtual destructor to release memory.
    virtual ~IEventSource() {}

    /// Subscribe to the event source, i.e. request notifications.
    /// This method raises the AlreadySubscribed exception, if the given
    /// event sink is already subscribed to the event source. In addition,
    /// an exception of type InvalidEventSink may be raised when the event
    /// argument of the event sink is not of the type the event source
    /// expects. This exception depends on additional metadata that is not
    /// defined in this component model.
    /// An event sink can only be subscribed once to each event source.
    /// Event sinks will be called in the order they have been subscribed
    /// to the event source.
    /// Models providing event sinks must ensure that these event sinks do
    /// not throw exceptions.
    /// @remarks Implementations may perform type checking on the optional
    ///          event argument of the event source and event sink.
    /// @param   eventSink Event sink to subscribe to event source.
    /// @throws  Smp::IEventSource::AlreadySubscribed
    /// @throws  Smp::IEventSource::InvalidEventSink
    virtual void Subscribe(Smp::IEventSink* eventSink) throw (
        Smp::IEventSource::AlreadySubscribed,
        Smp::IEventSource::InvalidEventSink) = 0;

    /// Unsubscribe from the event source, i.e. cancel notifications.
    /// This method raises the NotSubscribed exception if the given event
    /// sink is not subscribed to the event source.
    /// An event sink can only be unsubscribed if it has been subscribed
    /// before.
    /// @param   eventSink Event sink to unsubscribe from event source.
    /// @throws  Smp::IEventSource::NotSubscribed
    virtual void Unsubscribe(Smp::IEventSink* eventSink) throw (
        Smp::IEventSource::NotSubscribed) = 0;
};
```

**Base Interfaces**

Smp::IObject

**Operations**

| Name | Description |
|---|---|
| Subscribe | Subscribe to the event source, i.e. request notifications. |
| Unsubscribe | Unsubscribe from the event source, i.e. cancel notifications. |

#### 3.3.3.3.1 Subscribe

Subscribe to the event source, i.e. request notifications.

This method raises the AlreadySubscribed exception, if the given event sink is already subscribed to the event source. In addition, an exception of type InvalidEventSink may be raised when the event argument of the event sink is not of the type the event source expects. This exception depends on additional metadata that is not defined in this component model.

An event sink can only be subscribed once to each event source. Event sinks will be called in the order they have been subscribed to the event source.

Models providing event sinks must ensure that these event sinks do not throw exceptions.

***Remark:*** Implementations may perform type checking on the optional event argument of the event source and event sink.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| eventSink | in | IEventSink | Event sink to subscribe to event source. |

**Exceptions**

Smp::IEventSource::AlreadySubscribed, Smp::IEventSource::InvalidEventSink

#### 3.3.3.3.2 Unsubscribe

Unsubscribe from the event source, i.e. cancel notifications.

This method raises the NotSubscribed exception if the given event sink is not subscribed to the event source.

An event sink can only be unsubscribed if it has been subscribed before.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| eventSink | in | IEventSink | Event sink to unsubscribe from event source. |

**Exceptions**

Smp::IEventSource::NotSubscribed

#### 3.3.3.3.3 Already Subscribed

This exception is raised when trying to subscribe an event sink to an event source that is already subscribed.

**File**

#include "Smp/IEventSource.h"

**Namespace**

Smp::IEventSource

**Declaration of AlreadySubscribed**

```
/// This exception is raised when trying to subscribe an event sink to
/// an event source that is already subscribed.
class AlreadySubscribed : public Smp::Exception
{
public:
    /// Constructor for new exception.
    /// @param   eventSource Event source the event sink is subscribed
    ///          to.
    /// @param   eventSink Event sink that is already subscribed.
    AlreadySubscribed(
        const Smp::IEventSource* _eventSource,
        const Smp::IEventSink* eventSink) throw();

    /// Copy constructor.
    AlreadySubscribed(
        AlreadySubscribed& ex) throw();

    /// Virtual destructor to release memory.
    virtual ~AlreadySubscribed();

    /// Event source the event sink is subscribed to.
    Smp::IEventSource* eventSource;

    /// Event sink that is already subscribed.
    Smp::IEventSink* eventSink;
};
```

**Fields**

| Name | Type | Description |
|---|---|---|
| eventSink | IEventSink | Event sink that is already subscribed. |
| eventSource | IEventSource | Event source the event sink is subscribed to. |

#### 3.3.3.3.4 Invalid Event Sink

This exception is raised when trying to subscribe an event sink to an event source that has a different event type.

**File**

#include "Smp/IEventSource.h"

**Namespace**

Smp::IEventSource

**Declaration of InvalidEventSink**

```
/// This exception is raised when trying to subscribe an event sink to
/// an event source that has a different event type.
class InvalidEventSink : public Smp::Exception
{
public:
    /// Constructor for new exception.
    /// @param   eventSource Event source the event sink is subscribed
    ///          to.
    /// @param   eventSink Event sink that is not of valid type.
    InvalidEventSink(
        const Smp::IEventSource* _eventSource,
        const Smp::IEventSink* eventSink) throw();

    /// Copy constructor.
    InvalidEventSink(
```

```
            InvalidEventSink& ex) throw();

        /// Virtual destructor to release memory.
        virtual ~InvalidEventSink();

        /// Event source the event sink is subscribed to.
        Smp::IEventSource* eventSource;

        /// Event sink that is not of valid type.
        Smp::IEventSink* eventSink;
    };
```

**Fields**

| Name | Type | Description |
|---|---|---|
| eventSink | IEventSink | Event sink that is not of valid type. |
| eventSource | IEventSource | Event source the event sink is subscribed to. |

#### 3.3.3.3.5 Not Subscribed

This exception is raised when trying to unsubscribe an event sink from an event source that is not subscribed to it.

**File**

#include "Smp/IEventSource.h"

**Namespace**

Smp::IEventSource

**Declaration of NotSubscribed**

```
    /// This exception is raised when trying to unsubscribe an event sink
    /// from an event source that is not subscribed to it.
    class NotSubscribed : public Smp::Exception
    {
    public:
        /// Constructor for new exception.
        /// @param   eventSource Event source the event sink is not
        ///          subscribed to.
        /// @param   eventSink Event sink that is not subscribed.
        NotSubscribed(
            const Smp::IEventSource* _eventSource,
            const Smp::IEventSink* eventSink) throw();

        /// Copy constructor.
        NotSubscribed(
            NotSubscribed& ex) throw();

        /// Virtual destructor to release memory.
        virtual ~NotSubscribed();

        /// Event source the event sink is not subscribed to.
        Smp::IEventSource* eventSource;

        /// Event sink that is not subscribed.
        Smp::IEventSink* eventSink;
    };
```

**Fields**

| Name | Type | Description |
|---|---|---|
| eventSink | IEventSink | Event sink that is not subscribed. |
| eventSource | IEventSource | Event source the event sink is not subscribed to. |

#### 3.3.3.4 Event Source Collection

An event source collection is an ordered collection of event sources, which allows iterating all members.

This type is platform specific. For details see the SMP Platform Mappings.

**File**

#include "Smp/IEventSource.h"

**Namespace**

Smp

**Declaration of EventSourceCollection**

```
/// Unique Identifier of type EventSourceCollection.
extern const Uuid Uuid_EventSourceCollection;

/// An event source collection is an ordered collection of event sources,
/// which allows iterating all members.
///
/// This type is platform specific. For details see the SMP Platform
/// Mappings.
typedef std::vector<IEventSource*> EventSourceCollection;
```

### 3.3.4 Entry Points

An entry point is an interface that exposes a void function with no return value that can be called by the scheduler or event manager service.

#### 3.3.4.1 ITask

Interface for a task, which is an ordered collection of entry points.

This interface extends IEntryPoint to allow executing a number of entry points in one operation.

**File**

#include "Smp/ITask.h"

**Namespace**

Smp

**Declaration of ITask**

```
/// Unique Identifier of type ITask.
extern const Uuid Uuid_ITask;

/// Interface for a task, which is an ordered collection of entry points.
/// This interface extends IEntryPoint to allow executing a number of entry
/// points in one operation.
class ITask :
    public virtual Smp::IEntryPoint
{
public:

    /// Virtual destructor to release memory.
    virtual ~ITask() {}

    /// Add an entry point to the task.
    /// Entry points in a task will be executed in the order they have been
    /// added.
    /// @param   entryPoint Entry point to add to task.
    virtual void AddEntryPoint(const Smp::IEntryPoint* entryPoint) = 0;

    /// Query for the collection of all entry points. The order of entry
    /// points in the collection is the order in which they have been added
    /// to the task.
    /// The returned collection may be empty if no entry points have been
    /// added to the task.
    /// @return  Collection of entry points.
    virtual const Smp::EntryPointCollection* GetEntryPoints() const = 0;
};
```

**Base Interfaces**

Smp::IEntryPoint

**Operations**

| Name | Description |
|---|---|
| AddEntryPoint | Add an entry point to the task. |
| GetEntryPoints | Query for the collection of all entry points. The order of entry points in the collection is the order in which they have been added to the task. |

#### 3.3.4.1.1 Add Entry Point

Add an entry point to the task.

Entry points in a task will be executed in the order they have been added.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| entryPoint | in | IEntryPoint | Entry point to add to task. |

**Exceptions**

None

#### 3.3.4.1.2 Get Entry Points

Query for the collection of all entry points. The order of entry points in the collection is the order in which they have been added to the task.

The returned collection may be empty if no entry points have been added to the task.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | EntryPointCollection | Collection of entry points. |

**Exceptions**

None

### 3.3.4.2 IEntry Point

Interface of an entry point.

This interface provides a notification method (event handler) that can be called e.g. by the Scheduler or Event Manager when an event is emitted.

**File**

#include "Smp/IEntryPoint.h"

**Namespace**

Smp

**Declaration of IEntryPoint**

```
/// Unique Identifier of type IEntryPoint.
extern const Uuid Uuid_IEntryPoint;

/// Interface of an entry point.
/// This interface provides a notification method (event handler) that can
/// be called e.g. by the Scheduler or Event Manager when an event is
/// emitted.
class IEntryPoint :
    public virtual Smp::IObject
{
public:

    /// Virtual destructor to release memory.
    virtual ~IEntryPoint() {}

    /// This method returns the Component that owns the entry point.
    /// @remarks This is required to be able to store and restore entry
    ///          points.
    /// @return  Owner of entry point.
    virtual Smp::IComponent* GetOwner() const = 0;

    /// This method is called when an associated event is emitted.
    /// Components providing entry points must ensure that these entry
    /// points do not throw exceptions.
    virtual void Execute() const = 0;
};
```

**Base Interfaces**

Smp::IObject

**Operations**

| Name | Description |
|---|---|
| Execute | This method is called when an associated event is emitted. |
| GetOwner | This method returns the Component that owns the entry point. |

#### 3.3.4.2.1 Execute

This method is called when an associated event is emitted.

Components providing entry points must ensure that these entry points do not throw exceptions.

**Parameters**

None

**Exceptions**

None

#### 3.3.4.2.2 Get Owner

This method returns the Component that owns the entry point.

***Remark:*** This is required to be able to store and restore entry points.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | IComponent | Owner of entry point. |

**Exceptions**

None

### 3.3.4.3 Entry Point Collection

An entry point collection is an ordered collection of entry points, which allows iterating all members.

This type is platform specific. For details see the SMP Platform Mappings.

**File**

#include "Smp/IEntryPoint.h"

**Namespace**

Smp

**Declaration of EntryPointCollection**

```
/// Unique Identifier of type EntryPointCollection.
extern const Uuid Uuid_EntryPointCollection;

/// An entry point collection is an ordered collection of entry points,
/// which allows iterating all members.
/// This type is platform specific. For details see the SMP Platform
/// Mappings.
typedef std::vector<const IEntryPoint*> EntryPointCollection;
```

### 3.3.5 Dynamic Invocation

Dynamic invocation is a mechanism that makes the operations of a component available via a standardised interface (as opposed to a custom interface of the component which is not known at compile time of the simulation environment). In order to allow calling a named method with any number of parameters, a request object has to be created which contains all information needed for the method invocation. This request object is as well used to transfer back a return value of the operation.

The dynamic invocation concept presented here standardises the request objects (IRequest interface). In addition, two methods are provided as part of IDynamicInvocation to create and delete request objects. However, it is not mandatory to use these methods, as request objects can well be created and deleted using another implementation. A reason for doing this could be to minimise the number of round-trips between a client (that calls a method) and a component that implements IDynamicInvocation. The sequence diagram in Figure 1 shows all steps involved when using the CreateRequest() and DeleteRequest() methods.

***Remark***: Especially when running distributed components, this implementation is slow. In these cases, the request object should be created by the client, and only passed to the component on Invoke().

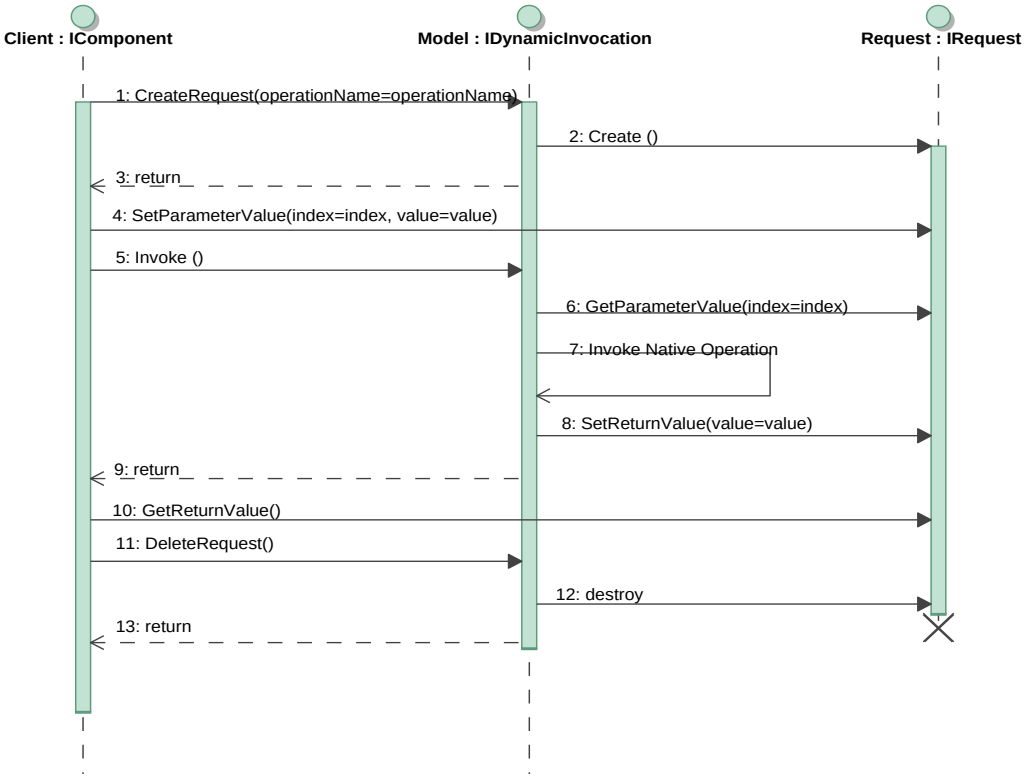


**Figure 1 - Sequence of calls for dynamic invocation**

The sequence diagram in Figure 1, using a Client component and a Model implementing IDynamicInvocation, contains the following steps:

1. The client calls the CreateRequest() operation of the component to create a request object for the operation, passing it the name of the operation.
2. The component creates a request object for the operation, using the default values of all parameters.
3. The component returns the Request object via its IRequest interface to the client.
4. The client calls the SetParameterValue() operation of the Request object to set parameters to non-default values.
5. The client calls the Invoke() operation of the component to invoke the corresponding operation.
6. The component calls the GetParameterValue() operation of the Request object to get parameters.
7. The component calls its internal operation that corresponds to the invoked operation.
8. The component calls the SetReturnValue() operation of the Request object to set the return value.
9. The component returns control to the client.
10. The client calls the GetReturnValue() operation of the Request object to get the return value.
11. The client calls the DeleteRequest() operation of the component to delete the Request object.
12. The component destroys the request object.
13. The component returns control to the client.

Like all mechanisms in this section, dynamic invocation is an optional mechanism.

#### 3.3.5.1 IDynamic Invocation

Interface for a component that supports dynamic invocation of operations.

A component may implement this interface in order to allow dynamic invocation of its operations.

**File**

#include "Smp/IDynamicInvocation.h"

**Namespace**

Smp

**Declaration of IDynamicInvocation**

```
/// Unique Identifier of type IDynamicInvocation.
extern const Uuid Uuid_IDynamicInvocation;

/// Interface for a component that supports dynamic invocation of
/// operations.
/// @remarks Dynamic invocation is typically used for scripting.
/// A component may implement this interface in order to allow dynamic
/// invocation of its operations.
class IDynamicInvocation :
    public virtual Smp::IComponent
{
public:

    /// Virtual destructor to release memory.
    virtual ~IDynamicInvocation() {}

    /// Return a request object for the given operation that describes the
    /// parameters and the return value.
    /// The request object may be null if no operation with the given name
    /// could be found, or if the operation with the given name does not
    /// support dynamic invocation.
    /// @param   operationName Name of operation.
    /// @return  Request object for operation, or null if either no
    ///          operation with the given name could be found, or the
    ///          operation with the given name does not support dynamic
    ///          invocation.
    virtual Smp::IRequest* CreateRequest(Smp::String8 operationName) = 0;

    /// Dynamically invoke an operation using a request object that has
    /// been created and filled with parameter values by the caller.
    /// @remarks The same request object can be used to invoke a method
    ///          several times.
    ///          This method raises the InvalidOperationName exception if
    ///          the request object passed does not name an operation that
    ///          allows dynamic invocation. When calling invoke with a
    ///          wrong number of parameters, the InvalidParameterCount
    ///          exception is raised. When passing a parameter of wrong
    ///          type, the InvalidParameterType exception is raised.
    /// @param   request Request object to invoke.
    /// @throws  Smp::IDynamicInvocation::InvalidOperationName
    /// @throws  Smp::IDynamicInvocation::InvalidParameterCount
    /// @throws  Smp::IDynamicInvocation::InvalidParameterType
    virtual void Invoke(Smp::IRequest* request) throw (
        Smp::IDynamicInvocation::InvalidOperationName,
        Smp::IDynamicInvocation::InvalidParameterCount,
        Smp::IDynamicInvocation::InvalidParameterType) = 0;

    /// Destroy a request object that has been created with the
    /// CreateRequest() method before.
    /// The request object must not be used anymore after DeleteRequest()
    /// has been called for it.
    /// @param   request Request object to destroy.
    virtual void DeleteRequest(Smp::IRequest* request) = 0;
};
```

***Remark:*** Dynamic invocation is typically used for scripting.

**Base Interfaces**

Smp::IComponent

**Operations**

| Name | Description |
|---|---|
| CreateRequest | Return a request object for the given operation that describes the parameters and the return value. |
| DeleteRequest | Destroy a request object that has been created with the CreateRequest() method before. |
| Invoke | Dynamically invoke an operation using a request object that has been created and filled with parameter values by the caller. |

#### 3.3.5.1.1 Create Request

Return a request object for the given operation that describes the parameters and the return value.

The request object may be null if no operation with the given name could be found, or if the operation with the given name does not support dynamic invocation.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | IRequest | Request object for operation, or null if either no operation with the given name could be found, or the operation with the given name does not support dynamic invocation. |
| operationName | in | String8 | Name of operation. |

**Exceptions**

None

#### 3.3.5.1.2 Delete Request

Destroy a request object that has been created with the CreateRequest() method before.

The request object must not be used anymore after DeleteRequest() has been called for it.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| request | in | IRequest | Request object to destroy. |

**Exceptions**

None

#### 3.3.5.1.3 Invoke

Dynamically invoke an operation using a request object that has been created and filled with parameter values by the caller.

This method raises the InvalidOperationName exception if the request object passed does not name an operation that allows dynamic invocation. When calling invoke with a wrong number of parameters, the InvalidParameterCount exception is raised. When passing a parameter of wrong type, the InvalidParameterType exception is raised.

***Remark:*** The same request object can be used to invoke a method several times.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| request | in | IRequest | Request object to invoke. |

**Exceptions**

Smp::IDynamicInvocation::InvalidOperationName,
Smp::IDynamicInvocation::InvalidParameterCount,
Smp::IDynamicInvocation::InvalidParameterType

#### 3.3.5.1.4 Invalid Operation Name

This exception is raised by the Invoke() method when trying to invoke a method that does not exist, or that does not support dynamic invocation.

**File**

#include "Smp/IDynamicInvocation.h"

**Namespace**

Smp::IDynamicInvocation

**Declaration of InvalidOperationName**

```
/// This exception is raised by the Invoke() method when trying to
/// invoke a method that does not exist, or that does not support
/// dynamic invocation.
class InvalidOperationName : public Smp::Exception
{
public:
    /// Constructor for new exception.
    /// @param   operationName Operation name of request passed to the
    ///          Invoke() method.
    InvalidOperationName(
        Smp::String8 operationName) throw();

    /// Copy constructor.
    InvalidOperationName(
        InvalidOperationName& ex) throw();

    /// Virtual destructor to release memory.
    virtual ~InvalidOperationName();

    /// Operation name of request passed to the Invoke() method.
    Smp::String8 operationName;
};
```

**Fields**

| Name | Type | Description |
|---|---|---|
| operationName | String8 | Operation name of request passed to the Invoke() method. |

#### 3.3.5.1.5 Invalid Parameter Count

This exception is raised by the Invoke() method when trying to invoke a method with a wrong number of parameters.

**File**

#include "Smp/IDynamicInvocation.h"

**Namespace**

Smp::IDynamicInvocation

**Declaration of InvalidParameterCount**

```
/// This exception is raised by the Invoke() method when trying to
/// invoke a method with a wrong number of parameters.
class InvalidParameterCount : public Smp::Exception
{
public:
    /// Constructor for new exception.
    /// @param   operationName Operation name of request passed to the
    ///          Invoke() method.
    /// @param   operationParameters Correct number of parameters of
    ///          operation.
    /// @param   requestParameters Wrong number of parameters of
    ///          operation.
    InvalidParameterCount(
        Smp::String8 operationName,
        Smp::Int32 operationParameters,
        Smp::Int32 requestParameters) throw();

    /// Copy constructor.
    InvalidParameterCount(
        InvalidParameterCount& ex) throw();

    /// Virtual destructor to release memory.
    virtual ~InvalidParameterCount();

    /// Operation name of request passed to the Invoke() method.
    Smp::String8 operationName;

    /// Correct number of parameters of operation.
    Smp::Int32 operationParameters;

    /// Wrong number of parameters of operation.
    Smp::Int32 requestParameters;
};
```

**Fields**

| Name | Type | Description |
|---|---|---|
| operationName | String8 | Operation name of request passed to the Invoke() method. |
| operationParameters | Int32 | Correct number of parameters of operation. |
| requestParameters | Int32 | Wrong number of parameters of operation. |

#### 3.3.5.1.6 Invalid Parameter Type

This exception is raised by the Invoke() method when trying to invoke a method passing a parameter of wrong type.

**File**

#include "Smp/IDynamicInvocation.h"

**Namespace**

Smp::IDynamicInvocation

**Declaration of InvalidParameterType**

```
/// This exception is raised by the Invoke() method when trying to
/// invoke a method passing a parameter of wrong type.
/// @remarks The index of the parameter of wrong type can be extracted
///          from the request using the method GetParameterIndex().
class InvalidParameterType : public Smp::Exception
{
public:
    /// Constructor for new exception.
    /// @param   operationName Operation name of request passed to the
    ///          Invoke() method.
    /// @param   parameterName Name of parameter of wrong type.
    /// @param   invalidType Type that is not valid.
    /// @param   expectedType Type that was expected.
    InvalidParameterType(
        Smp::String8 operationName,
        Smp::String8 parameterName,
        Smp::PrimitiveTypeKind invalidType,
        Smp::PrimitiveTypeKind expectedType) throw();

    /// Copy constructor.
    InvalidParameterType(
        InvalidParameterType& ex) throw();

    /// Virtual destructor to release memory.
    virtual ~InvalidParameterType();

    /// Operation name of request passed to the Invoke() method.
    Smp::String8 operationName;

    /// Name of parameter of wrong type.
    Smp::String8 parameterName;

    /// Type that is not valid.
    Smp::PrimitiveTypeKind invalidType;

    /// Type that was expected.
    Smp::PrimitiveTypeKind expectedType;
};
```

***Remark:*** The index of the parameter of wrong type can be extracted from the request using the method GetParameterIndex().

**Fields**

| Name | Type | Description |
|---|---|---|
| expectedType | PrimitiveTypeKind | Type that was expected. |
| invalidType | PrimitiveTypeKind | Type that is not valid. |
| operationName | String8 | Operation name of request passed to the Invoke() method. |
| parameterName | String8 | Name of parameter of wrong type. |

#### 3.3.5.2 IRequest

A request holds information, which is passed between a client invoking an operation via the IDynamicInvocation interface and a component being invoked.

**File**

#include "Smp/IRequest.h"

**Namespace**

Smp

**Declaration of IRequest**

```
/// Unique Identifier of type IRequest.
extern const Uuid Uuid_IRequest;

/// A request holds information, which is passed between a client invoking
/// an operation via the IDynamicInvocation interface and a component being
/// invoked.
class IRequest
{
public:

    /// Virtual destructor to release memory.
    virtual ~IRequest() {}

    /// Return the name of the operation that this request is for.
    /// @remarks A request is typically created using the CreateRequest()
    ///          method to dynamically call a specific method of a
    ///          component implementing the IDynamicInvocation interface.
    ///          This method returns the name passed to it, to allow
    ///          finding out which method is actually called on Invoke().
    /// @return  Name of the operation.
    virtual Smp::String8 GetOperationName() const = 0;

    /// Return the number of parameters stored in the request.
    /// Parameters can be accessed by their 0-based index. This index
    ///      - must not be negative,
    ///      - must be smaller than the parameter count.
    /// @remarks The GetParameterIndex() method may be used to access
    ///          parameters by name.
    /// @return  Number of parameters in request object.
    virtual Smp::Int32 GetParameterCount() const = 0;

    /// Query for a parameter index by parameter name.
    /// The index values are 0-based. An index of -1 indicates a wrong
    /// parameter name.
    /// @param   parameterName Name of parameter.
    /// @return  Index of parameter with the given name, or -1 if no
    ///          parameter with the given name could be found.
    virtual Smp::Int32 GetParameterIndex(Smp::String8 parameterName) const = 0;

    /// Assign a value to a parameter at a given position.
    /// This method raises an exception of type InvalidParameterIndex if
    /// called with an illegal parameter index. If called with an invalid
    /// parameter type, it raises an exception of type InvalidAnyType. If
    /// called with an invalid value for the parameter, it raises an
    /// exception of type InvalidParameterValue.
    /// @param   index Index of parameter (0-based).
    /// @param   value Value of parameter.
    /// @throws  Smp::InvalidAnyType
    /// @throws  Smp::IRequest::InvalidParameterIndex
    /// @throws  Smp::IRequest::InvalidParameterValue
```

```
        virtual void SetParameterValue(Smp::Int32 index, Smp::AnySimple value)
throw (
            Smp::InvalidAnyType,
            Smp::IRequest::InvalidParameterIndex,
            Smp::IRequest::InvalidParameterValue) = 0;

        /// Query a value of a parameter at a given position.
        /// This method raises an exception of type InvalidParameterIndex if
        /// called with an illegal parameter index.
        /// @param   index Index of parameter (0-based).
        /// @return  Value of parameter.
        /// @throws  Smp::IRequest::InvalidParameterIndex
        virtual Smp::AnySimple GetParameterValue(Smp::Int32 index) const throw (
            Smp::IRequest::InvalidParameterIndex) = 0;

        /// Assign the return value of the operation.
        /// This method raises an exception of type VoidOperation if called for
        /// a request object of a void operation. If called with an invalid
        /// return type, it raises an exception of type InvalidAnyType. If
        /// called with an invalid value for the return type, this method
        /// raises an exception of type InvalidReturnValue.
        /// @param   value Return value.
        /// @throws  Smp::InvalidAnyType
        /// @throws  Smp::IRequest::InvalidReturnValue
        /// @throws  Smp::IRequest::VoidOperation
        virtual void SetReturnValue(Smp::AnySimple value) throw (
            Smp::InvalidAnyType,
            Smp::IRequest::InvalidReturnValue,
            Smp::IRequest::VoidOperation) = 0;

        /// Query the return value of the operation.
        /// This method raises an exception of type VoidOperation if called for
        /// a request object of a void operation.
        /// @return  Return value of the operation.
        /// @throws  Smp::IRequest::VoidOperation
        virtual Smp::AnySimple GetReturnValue() const throw (
            Smp::IRequest::VoidOperation) = 0;
    };
```

**Base Interfaces**

None

**Operations**

| Name | Description |
|---|---|
| GetOperationName | Return the name of the operation that this request is for. |
| GetParameterCount | Return the number of parameters stored in the request. |
| GetParameterIndex | Query for a parameter index by parameter name. |
| GetParameterValue | Query a value of a parameter at a given position. |
| GetReturnValue | Query the return value of the operation. |
| SetParameterValue | Assign a value to a parameter at a given position. |
| SetReturnValue | Assign the return value of the operation. |

#### 3.3.5.2.1 Get Operation Name

Return the name of the operation that this request is for.

***Remark:*** A request is typically created using the CreateRequest() method to dynamically call a specific method of a component implementing the

IDynamicInvocation interface. This method returns the name passed to it, to allow finding out which method is actually called on Invoke().

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | String8 | Name of the operation. |

**Exceptions**

None

#### 3.3.5.2.2 Get Parameter Count

Return the number of parameters stored in the request.

Parameters can be accessed by their 0-based index. This index

- must not be negative,
- must be smaller than the parameter count.

***Remark:*** The GetParameterIndex() method may be used to access parameters by name.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | Int32 | Number of parameters in request object. |

**Exceptions**

None

#### 3.3.5.2.3 Get Parameter Index

Query for a parameter index by parameter name.

The index values are 0-based. An index of -1 indicates a wrong parameter name.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | Int32 | Index of parameter with the given name, or -1 if no parameter with the given name could be found. |
| parameterName | in | String8 | Name of parameter. |

**Exceptions**

None

#### 3.3.5.2.4 Get Parameter Value

Query a value of a parameter at a given position.

This method raises an exception of type InvalidParameterIndex if called with an illegal parameter index.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | AnySimple | Value of parameter. |
| index | in | Int32 | Index of parameter (0-based). |

**Exceptions**

Smp::IRequest::InvalidParameterIndex

#### 3.3.5.2.5 Get Return Value

Query the return value of the operation.

This method raises an exception of type VoidOperation if called for a request object of a void operation.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | AnySimple | Return value of the operation. |

**Exceptions**

Smp::IRequest::VoidOperation

#### 3.3.5.2.6 Set Parameter Value

Assign a value to a parameter at a given position.

This method raises an exception of type InvalidParameterIndex if called with an illegal parameter index. If called with an invalid parameter type, it raises an exception of type InvalidAnyType. If called with an invalid value for the parameter, it raises an exception of type InvalidParameterValue.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| index | in | Int32 | Index of parameter (0-based). |
| value | in | AnySimple | Value of parameter. |

**Exceptions**

Smp::InvalidAnyType, Smp::IRequest::InvalidParameterIndex, Smp::IRequest::InvalidParameterValue

#### 3.3.5.2.7 Set Return Value

Assign the return value of the operation.

This method raises an exception of type VoidOperation if called for a request object of a void operation. If called with an invalid return type, it raises an exception of type InvalidAnyType. If called with an invalid value for the return type, this method raises an exception of type InvalidReturnValue.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| value | in | AnySimple | Return value. |

**Exceptions**

Smp::InvalidAnyType, Smp::IRequest::InvalidReturnValue, Smp::IRequest::VoidOperation

#### 3.3.5.2.8 Invalid Parameter Index

This exception is raised when using an invalid parameter index to set (SetParameterValue()) or get (GetParameterValue()) a parameter value of an operation in a request.

**File**

#include "Smp/IRequest.h"

**Namespace**

Smp::IRequest

**Declaration of InvalidParameterIndex**

```
/// This exception is raised when using an invalid parameter index to
/// set (SetParameterValue()) or get (GetParameterValue()) a parameter
/// value of an operation in a request.
class InvalidParameterIndex : public Smp::Exception
{
public:
    /// Constructor for new exception.
    /// @param   operationName Name of operation.
    /// @param   parameterIndex Invalid parameter index used.
    /// @param   parameterCount Number of parameters of the operation.
    InvalidParameterIndex(
        Smp::String8 operationName,
        Smp::Int32 parameterIndex,
        Smp::Int32 parameterCount) throw();

    /// Copy constructor.
    InvalidParameterIndex(
        InvalidParameterIndex& ex) throw();

    /// Virtual destructor to release memory.
    virtual ~InvalidParameterIndex();

    /// Name of operation.
    Smp::String8 operationName;

    /// Invalid parameter index used.
    Smp::Int32 parameterIndex;

    /// Number of parameters of the operation.
    Smp::Int32 parameterCount;
};
```

**Fields**

| Name | Type | Description |
|---|---|---|
| operationName | String8 | Name of operation. |
| parameterCount | Int32 | Number of parameters of the operation. |
| parameterIndex | Int32 | Invalid parameter index used. |

#### 3.3.5.2.9 Invalid Parameter Value

This exception is raised when trying to assign an illegal value to a parameter of an operation in a request using SetParameterValue().

**File**

#include "Smp/IRequest.h"

**Namespace**

Smp::IRequest

**Declaration of InvalidParameterValue**

```
/// This exception is raised when trying to assign an illegal value to
/// a parameter of an operation in a request using SetParameterValue().
class InvalidParameterValue : public Smp::Exception
{
public:
    /// Constructor for new exception.
    /// @param   parameterName Name of parameter value was assigned to.
    /// @param   value Value that was passed as parameter.
    InvalidParameterValue(
        Smp::String8 parameterName,
        Smp::AnySimple value) throw();

    /// Copy constructor.
    InvalidParameterValue(
        InvalidParameterValue& ex) throw();

    /// Virtual destructor to release memory.
    virtual ~InvalidParameterValue();

    /// Name of parameter value was assigned to.
    Smp::String8 parameterName;

    /// Value that was passed as parameter.
    Smp::AnySimple value;
};
```

**Fields**

| Name | Type | Description |
|---|---|---|
| parameterName | String8 | Name of parameter value was assigned to. |
| value | AnySimple | Value that was passed as parameter. |

#### 3.3.5.2.10 Invalid Return Value

This exception is raised when trying to assign an invalid return value of an operation in a request using SetReturnValue().

**File**

#include "Smp/IRequest.h"

**Namespace**

Smp::IRequest

**Declaration of InvalidReturnValue**

```
/// This exception is raised when trying to assign an invalid return
/// value of an operation in a request using SetReturnValue().
```

```
class InvalidReturnValue : public Smp::Exception
{
public:
    /// Constructor for new exception.
    /// @param   operationName Name of operation the return value was
    ///          assigned to.
    /// @param   value Value that was passed as return value.
    InvalidReturnValue(
        Smp::String8 operationName,
        Smp::AnySimple value) throw();

    /// Copy constructor.
    InvalidReturnValue(
        InvalidReturnValue& ex) throw();

    /// Virtual destructor to release memory.
    virtual ~InvalidReturnValue();

    /// Name of operation the return value was assigned to.
    Smp::String8 operationName;

    /// Value that was passed as return value.
    Smp::AnySimple value;
};
```

**Fields**

| Name | Type | Description |
|---|---|---|
| operationName | String8 | Name of operation the return value was assigned to. |
| value | AnySimple | Value that was passed as return value. |

#### 3.3.5.2.11 Void Operation

This exception is raised when trying to read (GetReturnValue()) or write (SetReturnValue()) the return value of a void operation.

**File**

#include "Smp/IRequest.h"

**Namespace**

Smp::IRequest

**Declaration of VoidOperation**

```
/// This exception is raised when trying to read (GetReturnValue()) or
/// write (SetReturnValue()) the return value of a void operation.
class VoidOperation : public Smp::Exception
{
public:
    /// Constructor for new exception.
    /// @param   operationName Name of operation.
    VoidOperation(
        Smp::String8 operationName) throw();

    /// Copy constructor.
    VoidOperation(
        VoidOperation& ex) throw();

    /// Virtual destructor to release memory.
    virtual ~VoidOperation();

    /// Name of operation.
    Smp::String8 operationName;
```

```
        };
```

**Fields**

| Name | Type | Description |
|---|---|---|
| operationName | String8 | Name of operation. |

### 3.3.6 Persistence

Persistence of SMP components can be handled in one of two ways:

1. **External Persistence**: The simulation environment stores and restores the model's state by directly accessing the fields that are published to the simulation environment, i.e. via the IPublication interface.

   ***Remark***: This should be the preferred mechanism for the majority of models.

2. **Self-Persistence**: The component may implement the IPersist interface, which allows it to store and restore (part of) its state into or from storage that is provided by the simulation environment.

   ***Remark***: This mechanism is usually only needed by specialised models, for example embedded models that need to load on-board software from a specific file. Further, this mechanism can be used by simulation services if desired. For example, the Scheduler service may use it to store and restore its current state.

Like all mechanisms in this section, self-persistence of models and components is an optional mechanism, while external persistence (via the Store() and Restore() methods of the ISimulator interface) is a mandatory feature of every SMP simulation environment.

#### 3.3.6.1 IPersist

Interface of a self-persisting component that provides operations to allow for storing and restoring its state.

A component may implement this interface if it wants to have control over storing and restoring of its state. This is an optional interface which needs to be implemented by components with self-persistence only.

**File**

#include "Smp/IPersist.h"

**Namespace**

Smp

**Declaration of IPersist**

```
    /// Unique Identifier of type IPersist.
    extern const Uuid Uuid_IPersist;

    /// Interface of a self-persisting component that provides operations to
    /// allow for storing and restoring its state.
    /// A component may implement this interface if it wants to have control
    /// over storing and restoring of its state. This is an optional interface
    /// which needs to be implemented by components with self-persistence only.
```

```
class IPersist :
    public virtual Smp::IComponent
{
public:

    /// Virtual destructor to release memory.
    virtual ~IPersist() {}

    /// Restore component state from storage.
    /// This method raises an exception of type CannotRestore if reading
    /// data from the storage reader fails.
    /// @param   reader Interface that allows reading from storage.
    /// @throws  Smp::IPersist::CannotRestore
    virtual void Restore(Smp::IStorageReader* reader) throw (
        Smp::IPersist::CannotRestore) = 0;

    /// Store component state to storage.
    /// This method raises an exception of type CannotStore if writing data
    /// to the storage writer fails.
    /// @param   writer Interface that allows writing to storage.
    /// @throws  Smp::IPersist::CannotStore
    virtual void Store(Smp::IStorageWriter* writer) throw (
        Smp::IPersist::CannotStore) = 0;
};
```

**Base Interfaces**

Smp::IComponent

**Operations**

| Name | Description |
|---|---|
| Restore | Restore component state from storage. |
| Store | Store component state to storage. |

#### 3.3.6.1.1 Restore

Restore component state from storage.

This method raises an exception of type CannotRestore if reading data from the storage reader fails.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| reader | in | IStorageReader | Interface that allows reading from storage. |

**Exceptions**

Smp::IPersist::CannotRestore

#### 3.3.6.1.2 Store

Store component state to storage.

This method raises an exception of type CannotStore if writing data to the storage writer fails.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| writer | in | IStorageWriter | Interface that allows writing to storage. |

**Exceptions**

Smp::IPersist::CannotStore

#### 3.3.6.1.3 Cannot Restore

This exception is raised when the content of the storage reader passed to the Restore() method contains invalid data.

**File**

#include "Smp/IPersist.h"

**Namespace**

Smp::IPersist

**Declaration of CannotRestore**

```
/// This exception is raised when the content of the storage reader
/// passed to the Restore() method contains invalid data.
/// @remarks This typically happens when a Store() has been created
///          from a different configuration of components.
class CannotRestore : public Smp::Exception
{
public:
    /// Constructor for new exception.
    CannotRestore() throw();

    /// Copy constructor.
    CannotRestore(
        CannotRestore& ex) throw();

    /// Virtual destructor to release memory.
    virtual ~CannotRestore();

};
```

***Remark:*** This typically happens when a Store() has been created from a different configuration of components.

**Fields**

None

#### 3.3.6.1.4 Cannot Store

This exception is raised when the component cannot store its data to the storage writer given to the Store() method.

**File**

#include "Smp/IPersist.h"

**Namespace**

Smp::IPersist

**Declaration of CannotStore**

```
/// This exception is raised when the component cannot store its data
```

```
        /// to the storage writer given to the Store() method.
        /// @remarks This may e.g. be if there is no disk space left.
        class CannotStore : public Smp::Exception
        {
        public:
            /// Constructor for new exception.
            CannotStore() throw();

            /// Copy constructor.
            CannotStore(
                CannotStore& ex) throw();

            /// Virtual destructor to release memory.
            virtual ~CannotStore();

        };
```

***Remark:*** This may e.g. be if there is no disk space left.

**Fields**

None

#### 3.3.6.2 IStorage Reader

This interface provides functionality to read data from storage.

This interface is platform specific. For details see the SMP Platform Mappings.

**File**

#include "Smp/IStorageReader.h"

**Namespace**

Smp

**Declaration of IStorageReader**

```
    /// Unique Identifier of type IStorageReader.
    extern const Uuid Uuid_IStorageReader;

    /// This interface provides functionality to read data from storage.
    /// @remarks A client (typically the simulation environment) provides this
    ///          interface to allow components implementing the IPersist
    ///          interface to restore their state. It is passed to the
    ///          Restore() method of every model implementing IPersist.
    ///          This interface is platform specific. For details see the SMP
    ///          Platform Mappings.
    class IStorageReader
    {
    public:

        /// Virtual destructor to release memory.
        virtual ~IStorageReader() {}

        /// Restore data from storage.
        /// This method reads a memory block of data from the state vector. It
        /// is the component's responsibility to Store a block of the same size
        /// to IStorageWriter on Store.
        /// @param   address Memory address of memory block.
        /// @param   size Size of memory block.
        virtual void Restore(void* address, Smp::Int32 size) = 0;
    };
```

***Remark:*** A client (typically the simulation environment) provides this interface to allow components implementing the IPersist interface to restore their state. It is passed to the Restore() method of every model implementing IPersist.

**Base Interfaces**

None

**Operations**

| Name | Description |
|---|---|
| Restore | Restore data from storage. |

#### 3.3.6.2.1 Restore

Restore data from storage.

This method reads a memory block of data from the state vector. It is the component's responsibility to Store a block of the same size to IStorageWriter on Store.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| address | inout | void | Memory address of memory block. |
| size | in | Int32 | Size of memory block. |

**Exceptions**

None

### 3.3.6.3 IStorage Writer

This interface provides functionality to write data to storage.

This interface is platform specific. For details see the SMP Platform Mappings.

**File**

#include "Smp/IStorageWriter.h"

**Namespace**

Smp

**Declaration of IStorageWriter**

```
/// Unique Identifier of type IStorageWriter.
extern const Uuid Uuid_IStorageWriter;

/// This interface provides functionality to write data to storage.
/// @remarks A client (typically the simulation environment) provides this
///          interface to allow components implementing the IPersist
///          interface to store their state. It is passed to the Store()
///          method of every model implementing IPersist.
///          This interface is platform specific. For details see the SMP
///          Platform Mappings.
class IStorageWriter
{
public:

    /// Virtual destructor to release memory.
```

```
        virtual ~IStorageWriter() {}

        /// Store data to storage.
        /// This method writes a memory block of data to the state vector. It
        /// is the component's responsibility to Restore a block of the same
        /// size from IStorageReader on Restore.
        /// @param   address Memory address of memory block.
        /// @param   size Size of memory block.
        virtual void Store(void* address, Smp::Int32 size) = 0;
    };
```

***Remark:*** A client (typically the simulation environment) provides this interface to allow components implementing the IPersist interface to store their state. It is passed to the Store() method of every model implementing IPersist.

**Base Interfaces**

None

**Operations**

| Name | Description |
|---|---|
| Store | Store data to storage. |

#### 3.3.6.3.1 Store

Store data to storage.

This method writes a memory block of data to the state vector. It is the component's responsibility to Restore a block of the same size from IStorageReader on Restore.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| address | inout | void | Memory address of memory block. |
| size | in | Int32 | Size of memory block. |

**Exceptions**

None

## 3.4 Model Mechanisms

While the IModel interface defines the mandatory functionality every SMP model has to provide, this section introduces additional mechanisms available for more advanced use.

### 3.4.1 Fallible Models

Fallible models expose their failure state and a collection of failures.

#### 3.4.1.1 IFailure

Interface for a failure.

A Failure allows to query and to set its state to Failed or Unfailed.

**File**

#include "Smp/IFailure.h"

**Namespace**

Smp

**Declaration of IFailure**

```
/// Unique Identifier of type IFailure.
extern const Uuid Uuid_IFailure;

/// Interface for a failure.
/// A Failure allows to query and to set its state to Failed or Unfailed.
class IFailure :
    public virtual Smp::IObject
{
public:

    /// Virtual destructor to release memory.
    virtual ~IFailure() {}

    /// Sets the state of the failure to failed.
    virtual void Fail() = 0;

    /// Sets the state of the failure to unfailed.
    virtual void Unfail() = 0;

    /// Returns whether the failure's state is set to failed.
    /// @return  Returns true if the failure state is Failed, false
    ///          otherwise.
    virtual Smp::Bool IsFailed() const = 0;
};
```

**Base Interfaces**

Smp::IObject

**Operations**

| Name | Description |
|---|---|
| Fail | Sets the state of the failure to failed. |
| IsFailed | Returns whether the failure's state is set to failed. |
| Unfail | Sets the state of the failure to unfailed. |

##### 3.4.1.1.1 Fail

Sets the state of the failure to failed.

**Parameters**

None

**Exceptions**

None

#### 3.4.1.1.2 Is Failed

Returns whether the failure's state is set to failed.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | Bool | Returns true if the failure state is Failed, false otherwise. |

**Exceptions**

None

#### 3.4.1.1.3 Unfail

Sets the state of the failure to unfailed.

**Parameters**

None

**Exceptions**

None

### 3.4.1.2 IFallible Model

Interface for a fallible model that exposes its failure state and a collection of failures.

A fallible model allows querying for its failures by name.

**File**

#include "Smp/IFallibleModel.h"

**Namespace**

Smp

**Declaration of IFallibleModel**

```
/// Unique Identifier of type IFallibleModel.
extern const Uuid Uuid_IFallibleModel;

/// Interface for a fallible model that exposes its failure state and a
/// collection of failures.
/// A fallible model allows querying for its failures by name.
class IFallibleModel :
    public virtual Smp::IModel
{
public:

    /// Virtual destructor to release memory.
    virtual ~IFallibleModel() {}

    /// Query for whether the model is failed. A model is failed when at
    /// least one of its failures is failed.
    /// @return  Whether the model is failed or not.
    virtual Smp::Bool IsFailed() = 0;

    /// Get a failure by name.
    /// The returned failure may be null if no child with the given name
    /// could be found.
    /// @param   name Name of the failure to return.
    /// @return  Failure queried for by name, or null if no failure with
```

```
        ///          this name exists.
        virtual Smp::IFailure* GetFailure(Smp::String8 name) const = 0;

        /// Query for the collection of all failures.
        /// The returned collection may be empty if no failures exist for the
        /// fallible model.
        /// @return  Failure collection of the model.
        virtual const Smp::FailureCollection* GetFailures() const = 0;
    };
```

**Base Interfaces**

Smp::IModel

**Operations**

| Name | Description |
|---|---|
| GetFailure | Get a failure by name. |
| GetFailures | Query for the collection of all failures. |
| IsFailed | Query for whether the model is failed. A model is failed when at least one of its failures is failed. |

#### 3.4.1.2.1 Get Failure

Get a failure by name.

The returned failure may be null if no child with the given name could be found.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Name of the failure to return. |
| | return | IFailure | Failure queried for by name, or null if no failure with this name exists. |

**Exceptions**

None

#### 3.4.1.2.2 Get Failures

Query for the collection of all failures.

The returned collection may be empty if no failures exist for the fallible model.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | FailureCollection | Failure collection of the model. |

**Exceptions**

None

#### 3.4.1.2.3 Is Failed

Query for whether the model is failed. A model is failed when at least one of its failures is failed.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | Bool | Whether the model is failed or not. |

**Exceptions**

None

## 3.5 Management Interfaces

Managed interfaces allow external components to access all mechanisms by name. This includes the basic component mechanisms, optional component mechanisms and optional model mechanisms.

Managed interfaces allow full access to all functionality of components. For composition and aggregation, they extend the existing interfaces by methods to add new components or references, respectively. For entry points, event sources and event sinks, the managed interfaces provide access to the elements by name. For fields, access by name is provided by an extended model interface.

All management interfaces are optional, and only need to be provided for models used in a managed environment. Typically, in a managed environment a model configuration is built from an XML document (namely an SMDL Assembly) during the Creating phase.

***Remark:*** Typically, a Loader or Model Manager component calls these interfaces to push configuration information into the models, which has been read from an SMDL Assembly file. Initialisation of models from an SMDL Configuration file does not require managed models.

### 3.5.1 Managed Components

Managed components provide write access to their properties, i.e. they provide corresponding "setter" methods for the Name, Description, and Parent properties. This allows putting them into a hierarchy with a given name and description.

#### 3.5.1.1 IManaged Object

Interface of a managed object.

A managed object additionally allows assigning name and description.

**File**

#include "Smp/Management/IManagedObject.h"

**Namespace**

Smp::Management

**Declaration of IManagedObject**

```
/// Unique Identifier of type IManagedObject.
extern const Uuid Uuid_IManagedObject;

/// Interface of a managed object.
/// A managed object additionally allows assigning name and
/// description.
class IManagedObject :
    public virtual Smp::IObject
{
public:

    /// Virtual destructor to release memory.
    virtual ~IManagedObject() {}

    /// Define the name of the managed object ("property setter").
    /// This method throws an exception of type InvalidObjectName when
    /// the given name is not valid.
    /// Names must
    /// <ul>
    ///      - be unique within their context,
    ///      - not be empty,
    ///      - start with a letter, and
    ///      - only contain letters, digits, the underscore ("_") and
    /// brackets ("[" and "]").
    /// </ul>
    ///
    /// @remarks Except for the first rule (uniqueness of names), the
    ///          SetName() method should test for all other rules.
    /// @remarks It is recommended that names do not exceed 32
    ///          characters in size.
    /// @param   name Name of object.
    /// @throws  Smp::Management::IManagedObject::InvalidObjectName
    virtual void SetName(Smp::String8 name) throw (
        Smp::Management::IManagedObject::InvalidObjectName) = 0;

    /// Define the description of the managed object ("property
    /// setter").
    /// @param   description Description of object.
    virtual void SetDescription(Smp::String8 description) = 0;
};
```

**Base Interfaces**

Smp::IObject

**Operations**

| Name | Description |
|---|---|
| SetDescription | Define the description of the managed object ("property setter"). |
| SetName | Define the name of the managed object ("property setter"). |

#### 3.5.1.1.1 Set Description

Define the description of the managed object ("property setter").

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| description | in | String8 | Description of object. |

**Exceptions**

None

#### 3.5.1.1.2 Set Name

Define the name of the managed object ("property setter").

This method throws an exception of type InvalidObjectName when the given name is not valid.

Names must

- be unique within their context,
- not be empty,
- start with a letter, and
- only contain letters, digits, the underscore ("_") and brackets ("[" and "]").

***Remark:*** Except for the first rule (uniqueness of names), the SetName() method should test for all other rules.

***Remark:*** It is recommended that names do not exceed 32 characters in size.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Name of object. |

**Exceptions**

Smp::Management::IManagedObject::InvalidObjectName

#### 3.5.1.1.3 Invalid Object Name

This exception is raised when trying to set an object's name to an invalid name. Names

- must not be empty,
- must start with a letter, and
- must only contain letters, digits, the underscore ("_") and brackets ("[" and "]").

**File**

#include "Smp/Management/IManagedObject.h"

**Namespace**

Smp::Management::IManagedObject

**Declaration of InvalidObjectName**

```
/// This exception is raised when trying to set an object's name to
/// an invalid name. Names
/// <ul>
///      - must not be empty,
///      - must start with a letter, and
///      - must only contain letters, digits, the underscore ("_")
/// and brackets ("[" and "]").
/// </ul>
class InvalidObjectName : public Smp::Exception
{
public:
    /// Constructor for new exception.
    /// @param   objectName Invalid object name passed to SetName().
    InvalidObjectName(
        Smp::String8 objectName) throw();

    /// Copy constructor.
    InvalidObjectName(
        InvalidObjectName& ex) throw();

    /// Virtual destructor to release memory.
    virtual ~InvalidObjectName();

    /// Invalid object name passed to SetName().
    Smp::String8 objectName;
};
```

**Fields**

| Name | Type | Description |
|---|---|---|
| objectName | String8 | Invalid object name passed to SetName(). |

#### 3.5.1.2 IManaged Component

Interface of a managed component.

A managed component additionally allows assigning the parent.

**File**

#include "Smp/Management/IManagedComponent.h"

**Namespace**

Smp::Management

**Declaration of IManagedComponent**

```
/// Unique Identifier of type IManagedComponent.
extern const Uuid Uuid_IManagedComponent;

/// Interface of a managed component.
///  A managed component additionally allows assigning the parent.
class IManagedComponent :
    public virtual Smp::IComponent,
    public virtual Smp::Management::IManagedObject
{
public:

    /// Virtual destructor to release memory.
    virtual ~IManagedComponent() {}

    /// Define the parent component ("property setter"). Components
    /// link to their parent to allow traversing the tree of components
```

```
            /// upwards.
            /// @param   parent Parent composite of component.
            virtual void SetParent(Smp::IComposite* parent) = 0;
        };
```

**Base Interfaces**

Smp::Management::IManagedObject, Smp::IComponent

**Operations**

| Name | Description |
|---|---|
| SetParent | Define the parent component ("property setter"). Components link to their parent to allow traversing the tree of components upwards. |

#### 3.5.1.2.1 Set Parent

Define the parent component ("property setter"). Components link to their parent to allow traversing the tree of components upwards.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| parent | in | IComposite | Parent composite of component. |

**Exceptions**

None

### 3.5.2 Managed Component Mechanisms

The component mechanisms introduced in section 3.3 (Component Mechanisms) do not provide full access for external components, but only limited access:

- Aggregation provides collections of references, but does not allow adding new references to a Reference.
- Composition provides a tree of components, but does not allow adding new components to a Container.
- Event sinks can be connected by models if they have access to event sources, but an external component can not query for event sinks and event sources by name.
- Entry points can be registered with services by models, but an external component can not query for entry points by name.

To overcome these limitations, managed interfaces are provided with full access to all functionality. For composition and aggregation, these extend the existing interfaces by methods to add new components or references, respectively. For event sources and event sinks, the managed interfaces provide access to the elements by name. For entry points, the managed interface provides access to the entry points by name.

#### 3.5.2.1 IComponent Collection

Base interface for managed collections, which are either references or containers.

**File**

#include "Smp/Management/IComponentCollection.h"

**Namespace**

Smp::Management

**Declaration of IComponentCollection**

```
        /// Unique Identifier of type IComponentCollection.
        extern const Uuid Uuid_IComponentCollection;

        /// Base interface for managed collections, which are either references
        /// or containers.
        class IComponentCollection
        {
        public:

            /// Virtual destructor to release memory.
            virtual ~IComponentCollection() {}

            /// Query for the number of components in the collection.
            /// @return  Current number of components in the collection.
            virtual Smp::Int64 GetCount() const = 0;

            /// Query the maximum number of components in the collection.
            /// A return value of -1 indicates that the collection has no upper
            /// limit.
            ///
            /// @remarks This information can be used to check whether another
            ///          component can be added to the collection.
            /// @remarks This is consistent with the use of upper bounds in
            ///          UML, where a value of -1 represents no limit
            ///          (typically shown as *).
            /// @return  Maximum number of components in the collection. (-1 =
            ///          unlimited).
            virtual Smp::Int64 GetUpper() const = 0;

            /// Query the minimum number of components in the collection.
            /// @remarks This information can be used to validate a model
            ///          hierarchy. If a collection specifies a Lower value
            ///          above its current Count, then it is not properly
            ///          configured. An external component may use this
            ///          information to validate the configuration before
            ///          executing it.
            /// @return  Minimum number of components in the collection.
            virtual Smp::Int64 GetLower() const = 0;
        };
    }
```

**Base Interfaces**

None

**Operations**

| Name | Description |
|---|---|
| GetCount | Query for the number of components in the collection. |
| GetLower | Query the minimum number of components in the collection. |

| Name | Description |
|---|---|
| GetUpper | Query the maximum number of components in the collection. |

#### 3.5.2.1.1 Get Count

Query for the number of components in the collection.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | Int64 | Current number of components in the collection. |

**Exceptions**

None

#### 3.5.2.1.2 Get Lower

Query the minimum number of components in the collection.

***Remark:*** This information can be used to validate a model hierarchy. If a collection specifies a Lower value above its current Count, then it is not properly configured. An external component may use this information to validate the configuration before executing it.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | Int64 | Minimum number of components in the collection. |

**Exceptions**

None

#### 3.5.2.1.3 Get Upper

Query the maximum number of components in the collection.

A return value of -1 indicates that the collection has no upper limit.

***Remark:*** This information can be used to check whether another component can be added to the collection.

***Remark:*** This is consistent with the use of upper bounds in UML, where a value of -1 represents no limit (typically shown as *).

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | Int64 | Maximum number of components in the collection. (-1 = unlimited). |

**Exceptions**

None

#### 3.5.2.2 IManaged Reference

Interface of a managed reference.

A managed reference additionally allows querying the size limits and adding and removing referenced components.

**File**

#include "Smp/Management/IManagedReference.h"

**Namespace**

Smp::Management

**Declaration of IManagedReference**

```
/// Unique Identifier of type IManagedReference.
extern const Uuid Uuid_IManagedReference;

/// Interface of a managed reference.
/// A managed reference additionally allows querying the size limits
/// and adding and removing referenced components.
class IManagedReference :
    public virtual Smp::IReference,
    public virtual Smp::Management::IComponentCollection
{
public:

    /// Virtual destructor to release memory.
    virtual ~IManagedReference() {}

    /// Add a referenced component.
    /// This method raises an exception of type ReferenceFull if called
    /// for a full reference, i.e. when the Count has reached the Upper
    /// limit. This method may raise an exception of type
    /// InvalidObjectType when it expects the given component to
    /// implement another interface as well.
    /// Adding a component with a name that already exists in the
    /// reference does not throw an exception, although GetComponent()
    /// will no longer allow to return both referenced components by
    /// name.
    /// @param   component New referenced component.
    /// @throws  Smp::InvalidObjectType
    /// @throws  Smp::Management::IManagedReference::ReferenceFull
    virtual void AddComponent(Smp::IComponent* component) throw (
        Smp::InvalidObjectType,
        Smp::Management::IManagedReference::ReferenceFull) = 0;

    /// Remove a referenced component.
    /// This method raises an exception of type NotReferenced if called
    /// with a component that is not referenced. If the number of
    /// referenced components is less than or equal to the Lower limit,
    /// this method raises an exception of type CannotRemove.
    /// @param   component Referenced component to remove.
    /// @throws  Smp::Management::IManagedReference::CannotRemove
    /// @throws  Smp::Management::IManagedReference::NotReferenced
    virtual void RemoveComponent(Smp::IComponent* component) throw (
        Smp::Management::IManagedReference::CannotRemove,
        Smp::Management::IManagedReference::NotReferenced) = 0;
};
```

**Base Interfaces**

Smp::Management::IComponentCollection, Smp::IReference

**Operations**

| Name | Description |
|---|---|
| AddComponent | Add a referenced component. |
| RemoveComponent | Remove a referenced component. |

#### 3.5.2.2.1 Add Component

Add a referenced component.

This method raises an exception of type ReferenceFull if called for a full reference, i.e. when the Count has reached the Upper limit. This method may raise an exception of type InvalidObjectType when it expects the given component to implement another interface as well.

Adding a component with a name that already exists in the reference does not throw an exception, although GetComponent() will no longer allow to return both referenced components by name.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| component | in | IComponent | New referenced component. |

**Exceptions**

Smp::InvalidObjectType,
Smp::Management::IManagedReference::ReferenceFull

#### 3.5.2.2.2 Remove Component

Remove a referenced component.

This method raises an exception of type NotReferenced if called with a component that is not referenced. If the number of referenced components is less than or equal to the Lower limit, this method raises an exception of type CannotRemove.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| component | in | IComponent | Referenced component to remove. |

**Exceptions**

Smp::Management::IManagedReference::CannotRemove,
Smp::Management::IManagedReference::NotReferenced

#### 3.5.2.2.3 Cannot Remove

This exception is thrown when trying to remove a component from a reference when the number of referenced components is lower than or equal to the Lower limit.

**File**

#include "Smp/Management/IManagedReference.h"

**Namespace**

Smp::Management::IManagedReference

**Declaration of CannotRemove**

```
/// This exception is thrown when trying to remove a component from
/// a reference when the number of referenced components is lower
/// than or equal to the Lower limit.
class CannotRemove : public Smp::Exception
{
public:
    /// Constructor for new exception.
    /// @param   referenceName Name of reference.
    /// @param   component Component that could not be removed.
    /// @param   lowerLimit Lower limit of the reference.
    CannotRemove(
        Smp::String8 referenceName,
        Smp::IComponent* component,
        Smp::Int64 lowerLimit) throw();

    /// Copy constructor.
    CannotRemove(
        CannotRemove& ex) throw();

    /// Virtual destructor to release memory.
    virtual ~CannotRemove();

    /// Name of reference.
    Smp::String8 referenceName;

    /// Component that could not be removed.
    Smp::IComponent* component;

    /// Lower limit of the reference.
    Smp::Int64 lowerLimit;
};
```

**Fields**

| Name | Type | Description |
|---|---|---|
| component | IComponent | Component that could not be removed. |
| lowerLimit | Int64 | Lower limit of the reference. |
| referenceName | String8 | Name of reference. |

#### 3.5.2.2.4 Not Referenced

This exception is thrown when trying to remove a component from a reference which was not referenced before.

**File**

#include "Smp/Management/IManagedReference.h"

**Namespace**

Smp::Management::IManagedReference

**Declaration of NotReferenced**

```
/// This exception is thrown when trying to remove a component from
/// a reference which was not referenced before.
class NotReferenced : public Smp::Exception
{
public:
```

```
            /// Constructor for new exception.
            /// @param   referenceName Name of reference.
            /// @param   component Component that is not referenced.
            NotReferenced(
                Smp::String8 referenceName,
                Smp::IComponent* component) throw();

            /// Copy constructor.
            NotReferenced(
                NotReferenced& ex) throw();

            /// Virtual destructor to release memory.
            virtual ~NotReferenced();

            /// Name of reference.
            Smp::String8 referenceName;

            /// Component that is not referenced.
            Smp::IComponent* component;
        };
```

**Fields**

| Name | Type | Description |
|---|---|---|
| component | IComponent | Component that is not referenced. |
| referenceName | String8 | Name of reference. |

#### 3.5.2.2.5 Reference Full

This exception is raised when trying to add a component to a reference that is full, i.e. where the Count has reached the Upper limit.

**File**

#include "Smp/Management/IManagedReference.h"

**Namespace**

Smp::Management::IManagedReference

**Declaration of ReferenceFull**

```
        /// This exception is raised when trying to add a component to a
        /// reference that is full, i.e. where the Count has reached the
        /// Upper limit.
        class ReferenceFull : public Smp::Exception
        {
        public:
            /// Constructor for new exception.
            /// @param   referenceName Name of reference.
            /// @param   referenceSize Number of components in the
            ///          reference, which is its Upper limit when the
            ///          reference is full.
            ReferenceFull(
                Smp::String8 referenceName,
                Smp::Int64 referenceSize) throw();

            /// Copy constructor.
            ReferenceFull(
                ReferenceFull& ex) throw();

            /// Virtual destructor to release memory.
            virtual ~ReferenceFull();

            /// Name of reference.
```

```
            Smp::String8 referenceName;

            /// Number of components in the reference, which is its Upper
            /// limit when the reference is full.
            Smp::Int64 referenceSize;
        };
```

**Fields**

| Name | Type | Description |
|---|---|---|
| referenceName | String8 | Name of reference. |
| referenceSize | Int64 | Number of components in the reference, which is its Upper limit when the reference is full. |

#### 3.5.2.3 IManaged Container

Interface of a managed container.

A managed container additionally allows querying the size limits and adding contained components.

**File**

#include "Smp/Management/IManagedContainer.h"

**Namespace**

Smp::Management

**Declaration of IManagedContainer**

```
    /// Unique Identifier of type IManagedContainer.
    extern const Uuid Uuid_IManagedContainer;

    /// Interface of a managed container.
    /// A managed container additionally allows querying the size limits
    /// and adding contained components.
    class IManagedContainer :
        public virtual Smp::IContainer,
        public virtual Smp::Management::IComponentCollection
    {
    public:

        /// Virtual destructor to release memory.
        virtual ~IManagedContainer() {}

        /// Add a contained component to the container.
        /// This method raises an exception of type ContainerFull if called
        /// for a full container, i.e. when the Count has reached the Upper
        /// limit. It raises an exception of type DuplicateName when trying
        /// to add a component with a name that is already contained in the
        /// container, as this would lead to duplicate names in the
        /// container. This method may raise an exception of type
        /// InvalidObjectType when it expects the given component to
        /// implement another interface as well.
        /// @param   component New contained component.
        /// @throws  Smp::Management::IManagedContainer::ContainerFull
        /// @throws  Smp::DuplicateName
        /// @throws  Smp::InvalidObjectType
        virtual void AddComponent(Smp::IComponent* component) throw (
            Smp::Management::IManagedContainer::ContainerFull,
            Smp::DuplicateName,
            Smp::InvalidObjectType) = 0;
    };
```

**Base Interfaces**

Smp::Management::IComponentCollection, Smp::IContainer

**Operations**

| Name | Description |
|---|---|
| AddComponent | Add a contained component to the container. |

#### 3.5.2.3.1 Add Component

Add a contained component to the container.

This method raises an exception of type ContainerFull if called for a full container, i.e. when the Count has reached the Upper limit. It raises an exception of type DuplicateName when trying to add a component with a name that is already contained in the container, as this would lead to duplicate names in the container. This method may raise an exception of type InvalidObjectType when it expects the given component to implement another interface as well.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| component | in | IComponent | New contained component. |

**Exceptions**

Smp::DuplicateName, Smp::InvalidObjectType, Smp::Management::IManagedContainer::ContainerFull

#### 3.5.2.3.2 Container Full

This exception is raised when trying to add a component to a container that is full, i.e. where the Count has reached the Upper limit.

**File**

#include "Smp/Management/IManagedContainer.h"

**Namespace**

Smp::Management::IManagedContainer

**Declaration of ContainerFull**

```
/// This exception is raised when trying to add a component to a
/// container that is full, i.e. where the Count has reached the
/// Upper limit.
class ContainerFull : public Smp::Exception
{
public:
    /// Constructor for new exception.
    /// @param   containerName Name of full container.
    /// @param   containerSize Number of components in the
    ///          container, which is its Upper limit when the
    ///          container is full.
    ContainerFull(
        Smp::String8 containerName,
        Smp::Int64 containerSize) throw();

    /// Copy constructor.
    ContainerFull(
```

```
                    ContainerFull& ex) throw();

                /// Virtual destructor to release memory.
                virtual ~ContainerFull();

                /// Name of full container.
                Smp::String8 containerName;

                /// Number of components in the container, which is its Upper
                /// limit when the container is full.
                Smp::Int64 containerSize;
            };
```

**Fields**

| Name | Type | Description |
|---|---|---|
| containerName | String8 | Name of full container. |
| containerSize | Int64 | Number of components in the container, which is its Upper limit when the container is full. |

#### 3.5.2.4 IEvent Consumer

Interface of an event consumer.

An event sonsumer is a omponent that holds event sinks, which may be subscribed to other component's event sources.

This is an optional interface. It needs to be implemented for managed components only, which want to allow access to event sinks by name.

**File**

#include "Smp/Management/IEventConsumer.h"

**Namespace**

Smp::Management

**Declaration of IEventConsumer**

```
        /// Unique Identifier of type IEventConsumer.
        extern const Uuid Uuid_IEventConsumer;

        /// Interface of an event consumer.
        /// An event sonsumer is a omponent that holds event sinks, which may
        /// be subscribed to other component's event sources.
        ///
        /// This is an optional interface. It needs to be implemented for
        /// managed components only, which want to allow access to event sinks
        /// by name.
        class IEventConsumer :
            public virtual Smp::IComponent
        {
        public:

            /// Virtual destructor to release memory.
            virtual ~IEventConsumer() {}

            /// Query for the collection of all event sinks of the component.
            /// The collection may be empty if no event sinks exist.
            /// @return  Collection of event sinks.
            virtual const Smp::EventSinkCollection* GetEventSinks() const = 0;

            /// Query for an event sink of this component by its name.
            /// The returned event sink may be null if no event sink with the
            /// given name could be found.
```

```
            /// @param   name Event sink name.
            /// @return  Event sink with the given name, or null if no event
            ///          sink with the given name could be found.
            virtual Smp::IEventSink* GetEventSink(Smp::String8 name) const = 0;
        };
```

**Base Interfaces**

Smp::IComponent

**Operations**

| Name | Description |
|---|---|
| GetEventSink | Query for an event sink of this component by its name. |
| GetEventSinks | Query for the collection of all event sinks of the component. |

#### 3.5.2.4.1 Get Event Sink

Query for an event sink of this component by its name.

The returned event sink may be null if no event sink with the given name could be found.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | IEventSink | Event sink with the given name, or null if no event sink with the given name could be found. |
| name | in | String8 | Event sink name. |

**Exceptions**

None

#### 3.5.2.4.2 Get Event Sinks

Query for the collection of all event sinks of the component.

The collection may be empty if no event sinks exist.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | EventSinkCollection | Collection of event sinks. |

**Exceptions**

None

### 3.5.2.5 IEvent Provider

Interface of an event provider.

An event provider is a omponent that holds event sources, which allow other components to subscribe their event sinks.

This is an optional interface. It needs to be implemented for managed components only, which want to allow access to event sources by name.

**File**

#include "Smp/Management/IEventProvider.h"

**Namespace**

Smp::Management

**Declaration of IEventProvider**

```
/// Unique Identifier of type IEventProvider.
extern const Uuid Uuid_IEventProvider;

/// Interface of an event provider.
/// An event provider is a omponent that holds event sources, which
/// allow other components to subscribe their event sinks.
/// This is an optional interface. It needs to be implemented for
/// managed components only, which want to allow access to event
/// sources by name.
class IEventProvider :
    public virtual Smp::IComponent
{
public:

    /// Virtual destructor to release memory.
    virtual ~IEventProvider() {}

    /// Query for an event source of this component by its name.
    /// The returned event source may be null if no event source with
    /// the given name could be found.
    /// @param   name Event source name.
    /// @return  Event source with the given name or null if no event
    ///          source with the given name could be found.
    virtual Smp::IEventSource* GetEventSource(Smp::String8 name) const = 0;

    /// Query for the collection of all event sources of the component.
    /// The collection may be empty if no event sources exist.
    /// @return  Collection of event sources.
    virtual const Smp::EventSourceCollection* GetEventSources() const = 0;
};
```

**Base Interfaces**

Smp::IComponent

**Operations**

| Name | Description |
|---|---|
| GetEventSource | Query for an event source of this component by its name. |
| GetEventSources | Query for the collection of all event sources of the component. |

#### 3.5.2.5.1 Get Event Source

Query for an event source of this component by its name.

The returned event source may be null if no event source with the given name could be found.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | IEventSource | Event source with the given name or null if no event source with the given name could be found. |
| name | in | String8 | Event source name. |

**Exceptions**

None

#### 3.5.2.5.2 Get Event Sources

Query for the collection of all event sources of the component.

The collection may be empty if no event sources exist.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | EventSourceCollection | Collection of event sources. |

**Exceptions**

None

### 3.5.2.6 IEntry Point Publisher

Interface of an entry point publisher.

An entry point publisher is a model that publishes entry points.

This is an optional interface. It needs to be implemented for managed models only, which want to provide access to their entry points by name.

**File**

#include "Smp/Management/IEntryPointPublisher.h"

**Namespace**

Smp::Management

**Declaration of IEntryPointPublisher**

```
/// Unique Identifier of type IEntryPointPublisher.
extern const Uuid Uuid_IEntryPointPublisher;

/// Interface of an entry point publisher.
/// An entry point publisher is a model that publishes entry points.
///
/// @remarks The entry points may be registered, for example, with the
///          Scheduler or the Event Manager services.
///          This is an optional interface. It needs to be implemented
///          for managed models only, which want to provide access to
///          their entry points by name.
class IEntryPointPublisher :
    public virtual Smp::IComponent
{
public:

    /// Virtual destructor to release memory.
    virtual ~IEntryPointPublisher() {}
```

```
            /// Query for the collection of all entry points of the model.
            /// The collection may be empty if no entry points exist.
            /// @return  Collection of entry points.
            virtual const Smp::EntryPointCollection* GetEntryPoints() const = 0;

            /// Query for an entry point of this model by its name.
            /// The returned entry point may be null if no entry point with the
            /// given name could be found.
            /// @param   name Entry point name.
            /// @return  Entry point with given name, or null if no entry point
            ///          with given name could be found.
            virtual const Smp::IEntryPoint* GetEntryPoint(Smp::String8 name) const
= 0;
        };
```

***Remark:*** The entry points may be registered, for example, with the Scheduler or the Event Manager services.

**Base Interfaces**

Smp::IComponent

**Operations**

| Name | Description |
|---|---|
| GetEntryPoint | Query for an entry point of this model by its name. |
| GetEntryPoints | Query for the collection of all entry points of the model. |

#### 3.5.2.6.1 Get Entry Point

Query for an entry point of this model by its name.

The returned entry point may be null if no entry point with the given name could be found.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | IEntryPoint | Entry point with given name, or null if no entry point with given name could be found. |
| name | in | String8 | Entry point name. |

**Exceptions**

None

#### 3.5.2.6.2 Get Entry Points

Query for the collection of all entry points of the model.

The collection may be empty if no entry points exist.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | EntryPointCollection | Collection of entry points. |

**Exceptions**

None

### 3.5.3 Managed Model Mechanisms

The model mechanisms introduced in section 3.4 (Model Mechanisms) do not provide full access for external components, but only limited access:

- Fields are published against the simulation environment, but an external component can not query for fields by name.

To overcome these limitations, managed interfaces are provided with full access to all functionality. For fields, the managed interface allows for querying a dedicate field of a model by name. Via the returned ISimpleField and the IArrayField interface it is possible to read and write field values.

**Field Names:**

In SMP, models can have fields of various types, including not only the base types introduced in section 2 ( Base Types), but also complex types (strings, arrays, structures, classes). The available types are documented in the Metamodel. As the methods GetValue() and SetValue() of ISimpleField only support fields of simple types, structured types have to be accessed by accessing their individual members.

**Rule 1**: Members of structures and classes are separated with one of the following characters: "\", "/", "!", "."

**Rule 2**: Members of strings and arrays with non-simple item type are addressed by their 0-based index enclosed in brackets ("[" and "]").

Examples:

MyField.Position[2]

Structure!Field

Class/Array[1]/Structure/Field

#### 3.5.3.1 IManaged Model

Interface of a managed model.

A managed model additionally allows querying for fields by name.

This is an optional interface. It needs to be implemented for managed models only, which want to allow access to fields by name.

**File**

#include "Smp/Management/IManagedModel.h"

**Namespace**

Smp::Management

**Declaration of IManagedModel**

```
        /// Unique Identifier of type IManagedModel.
        extern const Uuid Uuid_IManagedModel;

        /// Interface of a managed model.
        /// A managed model additionally allows querying for fields by name.
        /// This is an optional interface. It needs to be implemented for
        /// managed models only, which want to allow access to fields by name.
        class IManagedModel :
            public virtual Smp::IModel,
            public virtual Smp::Management::IManagedComponent
        {
        public:

            /// Virtual destructor to release memory.
            virtual ~IManagedModel() {}

            /// Get the field of given name that is typed by a simple type.
            ///
            /// This method raises an exception of type InvalidFieldName if
            /// called with a field name for which no corresponding field
            /// exists or for which the corresponding field is not of simple
            /// type.
            /// This method can only be used to get fields of simple types.
            ///
            /// @remarks For getting access to fields of structured or array
            ///          types, this method may be called multiply, for example
            ///          by specifying a field name "MyField.Position[2]" in
            ///          order to access an array item of a structure.
            /// @param   fullName Fully qualified field name (relative to the
            ///          model).
            /// @return  Simple field.
            /// @throws  Smp::Management::IManagedModel::InvalidFieldName
            virtual Smp::ISimpleField* GetSimpleField(Smp::String8 fullName) const
throw (
                Smp::Management::IManagedModel::InvalidFieldName) = 0;

            /// Get the field of given name that is an array of a simple type.
            /// This method raises an exception of type InvalidFieldName if
            /// called with a field name for which no corresponding field
            /// exists or for which the corresponding field is not an array of
            /// simple type.
            ///
            /// This method can only be used to get array fields with items of
            /// simple type.
            /// @param   fullName Fully qualified array field name (relative to
            ///          the model)
            /// @return  Array field.
            /// @throws  Smp::Management::IManagedModel::InvalidFieldName
            virtual Smp::IArrayField* GetArrayField(Smp::String8 fullName) const
throw (
                Smp::Management::IManagedModel::InvalidFieldName) = 0;
        };
```

**Base Interfaces**

Smp::Management::IManagedComponent, Smp::IModel

**Operations**

| Name | Description |
|---|---|
| GetArrayField | Get the field of given name that is an array of a simple type. |
| GetSimpleField | Get the field of given name that is typed by a simple type. |

#### 3.5.3.1.1 Get Array Field

Get the field of given name that is an array of a simple type.

This method raises an exception of type InvalidFieldName if called with a field name for which no corresponding field exists or for which the corresponding field is not an array of simple type.

This method can only be used to get array fields with items of simple type.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| fullName | in | String8 | Fully qualified array field name (relative to the model) |
| | return | IArrayField | Array field. |

**Exceptions**

Smp::Management::IManagedModel::InvalidFieldName

#### 3.5.3.1.2 Get Simple Field

Get the field of given name that is typed by a simple type.

This method raises an exception of type InvalidFieldName if called with a field name for which no corresponding field exists or for which the corresponding field is not of simple type.

This method can only be used to get fields of simple types.

***Remark:*** For getting access to fields of structured or array types, this method may be called multiply, for example by specifying a field name "MyField.Position[2]" in order to access an array item of a structure.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | ISimpleField | Simple field. |
| fullName | in | String8 | Fully qualified field name (relative to the model). |

**Exceptions**

Smp::Management::IManagedModel::InvalidFieldName

#### 3.5.3.1.3 Invalid Field Name

This exception is raised when an invalid field name is specified.

**File**

#include "Smp/Management/IManagedModel.h"

**Namespace**

Smp::Management::IManagedModel

**Declaration of InvalidFieldName**

```
/// This exception is raised when an invalid field name is
/// specified.
class InvalidFieldName : public Smp::Exception
{
public:
    /// Constructor for new exception.
    /// @param   fieldName Fully qualified field name that is
    ///          invalid.
    InvalidFieldName(
        Smp::String8 fieldName) throw();

    /// Copy constructor.
    InvalidFieldName(
        InvalidFieldName& ex) throw();

    /// Virtual destructor to release memory.
    virtual ~InvalidFieldName();

    /// Fully qualified field name that is invalid.
    Smp::String8 fieldName;
};
```

**Fields**

| Name | Type | Description |
|---|---|---|
| fieldName | String8 | Fully qualified field name that is invalid. |

#### 3.5.3.2 IField

Interface of a field.

**File**

#include "Smp/IField.h"

**Namespace**

Smp

**Declaration of IField**

```
/// Unique Identifier of type IField.
extern const Uuid Uuid_IField;

/// Interface of a field.
class IField :
    public virtual Smp::IObject
{
public:

    /// Virtual destructor to release memory.
    virtual ~IField() {}

    /// Return View kind of the field.
    /// @return  The View kind of the field.
    /// The view kind indicates which user roles have visibility of the
    /// field.
    virtual Smp::ViewKind GetView() const = 0;

    /// Return State flag of the field.
    /// @return  The State flag of the field.
    /// When true, the state of the field shall be stored by the simulation
    /// infrastructure persistence mechanism on Store(), and restored on
    /// Restore().
    virtual Smp::Bool IsState() const = 0;
```

```
        /// Return Input flag of the field.
        /// @return  The Input flag of the field.
        /// When true, the field is considered an input into the model and can
        /// be used as target of a field link in data flow based design.
        virtual Smp::Bool IsInput() const = 0;

        /// Return Output flag of the field.
        /// @return  The Output flag of the field.
        /// When true, the field is considered an output of the model and can
        /// be used as source of a field link in data flow based design.
        virtual Smp::Bool IsOutput() const = 0;
    };
```

**Base Interfaces**

Smp::IObject

**Operations**

| Name | Description |
|---|---|
| GetView | Return View kind of the field. |
| IsInput | Return Input flag of the field. |
| IsOutput | Return Output flag of the field. |
| IsState | Return State flag of the field. |

#### 3.5.3.2.1 Get View

Return View kind of the field.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | ViewKind | The View kind of the field.<br>The view kind indicates which user roles have visibility of the field. |

**Exceptions**

None

#### 3.5.3.2.2 Is Input

Return Input flag of the field.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | Bool | The Input flag of the field.<br>When true, the field is considered an input into the model and can be used as target of a field link in data flow based design. |

**Exceptions**

None

#### 3.5.3.2.3 Is Output

Return Output flag of the field.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | Bool | The Output flag of the field.<br>When true, the field is considered an output of the model and can be used as source of a field link in data flow based design. |

**Exceptions**

None

#### 3.5.3.2.4 Is State

Return State flag of the field.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | Bool | The State flag of the field.<br>When true, the state of the field shall be stored by the simulation infrastructure persistence mechanism on Store(), and restored on Restore(). |

**Exceptions**

None

#### 3.5.3.2.5 Invalid Field Value

This exception is raised when trying to assign an illegal value to a field.

**File**

#include "Smp/IField.h"

**Namespace**

Smp::IField

**Declaration of InvalidFieldValue**

```
        /// This exception is raised when trying to assign an illegal value to
        /// a field.
        class InvalidFieldValue : public Smp::Exception
        {
        public:
            /// Constructor for new exception.
            /// @param   fieldName Fully qualified field name the value was
            ///          assigned to.
            /// @param   invalidFieldValue Value that was passed as new field
            ///          value.
            InvalidFieldValue(
                Smp::String8 fieldName,
                Smp::AnySimple invalidFieldValue) throw();

            /// Copy constructor.
            InvalidFieldValue(
```

```
                InvalidFieldValue& ex) throw();

            /// Virtual destructor to release memory.
            virtual ~InvalidFieldValue();

            /// Fully qualified field name the value was assigned to.
            Smp::String8 fieldName;

            /// Value that was passed as new field value.
            Smp::AnySimple invalidFieldValue;
        };
```

**Fields**

| Name | Type | Description |
|---|---|---|
| fieldName | String8 | Fully qualified field name the value was assigned to. |
| invalidFieldValue | AnySimple | Value that was passed as new field value. |

#### 3.5.3.3 IArray Field

Interface of a field which is an array of a simple type.

**File**

#include "Smp/IArrayField.h"

**Namespace**

Smp

**Declaration of IArrayField**

```
    /// Unique Identifier of type IArrayField.
    extern const Uuid Uuid_IArrayField;

    /// Interface of a field which is an array of a simple type.
    class IArrayField :
        public virtual Smp::IField
    {
    public:

        /// Virtual destructor to release memory.
        virtual ~IArrayField() {}

        /// Get a value from a specific index of the array field.
        /// This method raises an exception of type InvalidIndex if called with
        /// an index value beyond the size of the array.
        /// @param   index Index of value to get.
        /// @return  Value from given index.
        /// @throws  Smp::IArrayField::InvalidIndex
        virtual Smp::AnySimple GetValue(Smp::UInt64 index) const throw (
            Smp::IArrayField::InvalidIndex) = 0;

        /// Set a value at given index of the array field.
        /// This method raises an exception of type InvalidFieldValue if called
        /// with an invalid value and an exception of type InvalidIndex if
        /// called with an index value beyond the size of the array.
        /// @param   index Index of value to set.
        /// @param   value Value to set at given index.
        /// @throws  Smp::IField::InvalidFieldValue
        /// @throws  Smp::IArrayField::InvalidIndex
        virtual void SetValue(Smp::UInt64 index, Smp::AnySimple value) throw (
            Smp::IField::InvalidFieldValue,
```

```
            Smp::IArrayField::InvalidIndex) = 0;

        /// Get all values of the array field.
        /// This method raises an exception of type InvalidArraySize if called
        /// with an array of wrong size.
        /// The array with the values has to be pre-allocated by the caller,
        /// and has to be released by the caller as well. Therefore, an inout
        /// parameter is used, not a return value of the method.
        /// @param   length Size of given values array.
        /// @param   values Pre-allocated array of values to store result to.
        /// @throws  Smp::IArrayField::InvalidArraySize
        virtual void GetValues(Smp::UInt64 length, Smp::AnySimpleArray values)
const throw (
            Smp::IArrayField::InvalidArraySize) = 0;

        /// Set all values of the array field.
        /// This method raises an exception of type InvalidArraySize if called
        /// with an array of wrong size and an exception of type
        /// InvalidArrayValue if called with an invalid values array.
        /// @param   length Size of given values array.
        /// @param   values Array of values to store in array field.
        /// @throws  Smp::IArrayField::InvalidArraySize
        /// @throws  Smp::IArrayField::InvalidArrayValue
        virtual void SetValues(Smp::UInt64 length, Smp::AnySimpleArray values)
throw (
            Smp::IArrayField::InvalidArraySize,
            Smp::IArrayField::InvalidArrayValue) = 0;

        /// Get the size (number of array items) of the field.
        /// @return  Size (number of array items) of the field.
        virtual Smp::UInt64 GetSize() const = 0;
    };
```

**Base Interfaces**

Smp::IField

**Operations**

| Name | Description |
|---|---|
| GetSize | Get the size (number of array items) of the field. |
| GetValue | Get a value from a specific index of the array field. |
| GetValues | Get all values of the array field. |
| SetValue | Set a value at given index of the array field. |
| SetValues | Set all values of the array field. |

#### 3.5.3.3.1 Get Size

Get the size (number of array items) of the field.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | UInt64 | Size (number of array items) of the field. |

**Exceptions**

None

#### 3.5.3.3.2 Get Value

Get a value from a specific index of the array field.

This method raises an exception of type InvalidIndex if called with an index value beyond the size of the array.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | AnySimple | Value from given index. |
| index | in | UInt64 | Index of value to get. |

**Exceptions**

Smp::IArrayField::InvalidIndex

#### 3.5.3.3.3 Get Values

Get all values of the array field.

This method raises an exception of type InvalidArraySize if called with an array of wrong size.

The array with the values has to be pre-allocated by the caller, and has to be released by the caller as well. Therefore, an inout parameter is used, not a return value of the method.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| length | in | UInt64 | Size of given values array. |
| values | inout | AnySimpleArray | Pre-allocated array of values to store result to. |

**Exceptions**

Smp::IArrayField::InvalidArraySize

#### 3.5.3.3.4 Set Value

Set a value at given index of the array field.

This method raises an exception of type InvalidFieldValue if called with an invalid value and an exception of type InvalidIndex if called with an index value beyond the size of the array.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| index | in | UInt64 | Index of value to set. |
| value | in | AnySimple | Value to set at given index. |

**Exceptions**

Smp::IField::InvalidFieldValue, Smp::IArrayField::InvalidIndex

#### 3.5.3.3.5 Set Values

Set all values of the array field.

This method raises an exception of type InvalidArraySize if called with an array of wrong size and an exception of type InvalidArrayValue if called with an invalid values array.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| length | in | UInt64 | Size of given values array. |
| values | in | AnySimpleArray | Array of values to store in array field. |

**Exceptions**

Smp::IArrayField::InvalidArraySize, Smp::IArrayField::InvalidArrayValue

#### 3.5.3.3.6 Invalid Array Size

This exception is raised when an invalid array size is specified.

**File**

#include "Smp/IArrayField.h"

**Namespace**

Smp::IArrayField

**Declaration of InvalidArraySize**

```
/// This exception is raised when an invalid array size is specified.
class InvalidArraySize : public Smp::Exception
{
public:
    /// Constructor for new exception.
    /// @param   fieldName Name of field that has been accessed.
    /// @param   invalidSize Invalid array size.
    /// @param   arraySize Real array size.
    InvalidArraySize(
        Smp::String8 fieldName,
        Smp::Int64 invalidSize,
        Smp::Int64 arraySize) throw();

    /// Copy constructor.
    InvalidArraySize(
        InvalidArraySize& ex) throw();

    /// Virtual destructor to release memory.
    virtual ~InvalidArraySize();

    /// Name of field that has been accessed.
    Smp::String8 fieldName;

    /// Invalid array size.
    Smp::Int64 invalidSize;

    /// Real array size.
    Smp::Int64 arraySize;
};
```

**Fields**

| Name | Type | Description |
|---|---|---|
| arraySize | Int64 | Real array size. |
| fieldName | String8 | Name of field that has been accessed. |
| invalidSize | Int64 | Invalid array size. |

#### 3.5.3.3.7 Invalid Array Value

This exception is raised when trying to assign an illegal value to an array field.

**File**

#include "Smp/IArrayField.h"

**Namespace**

Smp::IArrayField

**Declaration of InvalidArrayValue**

```
/// This exception is raised when trying to assign an illegal value to
/// an array field.
class InvalidArrayValue : public Smp::Exception
{
public:
    /// Constructor for new exception.
    /// @param   fieldName Fully qualified field name the value was
    ///          assigned to.
    /// @param   validIndex Index in array where the first invalid
    ///          value was found.
    InvalidArrayValue(
        Smp::String8 fieldName,
        Smp::Int32 invalidValueIndex) throw();

    /// Copy constructor.
    InvalidArrayValue(
        InvalidArrayValue& ex) throw();

    /// Virtual destructor to release memory.
    virtual ~InvalidArrayValue();

    /// Fully qualified field name the value was assigned to.
    Smp::String8 fieldName;

    /// Index in array where the first invalid value was found.
    Smp::Int32 invalidValueIndex;
};
```

**Fields**

| Name | Type | Description |
|---|---|---|
| fieldName | String8 | Fully qualified field name the value was assigned to. |
| invalidValueIndex | Int32 | Index in array where the first invalid value was found. |

#### 3.5.3.3.8 Invalid Index

This exception is raised when an invalid index is specified.

**File**

#include "Smp/IArrayField.h"

**Namespace**

Smp::IArrayField

**Declaration of InvalidIndex**

```
/// This exception is raised when an invalid index is specified.
class InvalidIndex : public Smp::Exception
{
public:
    /// Constructor for new exception.
    /// @param   fieldName Fully qualified field name the value was
    ///          assigned to.
    /// @param   arraySize Real array size.
    /// @param   invalidIndex Invalid Index.
    InvalidIndex(
        Smp::String8 fieldName,
        Smp::Int64 arraySize,
        Smp::Int64 invalidIndex) throw();

    /// Copy constructor.
    InvalidIndex(
        InvalidIndex& ex) throw();

    /// Virtual destructor to release memory.
    virtual ~InvalidIndex();

    /// Fully qualified field name the value was assigned to.
    Smp::String8 fieldName;

    /// Real array size.
    Smp::Int64 arraySize;

    /// Invalid Index.
    Smp::Int64 invalidIndex;
};
```

**Fields**

| Name | Type | Description |
|---|---|---|
| arraySize | Int64 | Real array size. |
| fieldName | String8 | Fully qualified field name the value was assigned to. |
| invalidIndex | Int64 | Invalid Index. |

#### 3.5.3.4 ISimple Field

Interface of a field of simple type.

**File**

#include "Smp/ISimpleField.h"

**Namespace**

Smp

**Declaration of ISimpleField**

```
/// Unique Identifier of type ISimpleField.
extern const Uuid Uuid_ISimpleField;
```

```
/// Interface of a field of simple type.
class ISimpleField :
    public virtual Smp::IField
{
public:

    /// Virtual destructor to release memory.
    virtual ~ISimpleField() {}

    /// Get the value of the simple field.
    /// @return  Field value.
    virtual Smp::AnySimple GetValue() const = 0;

    /// Set the value of the simple field.
    /// This method raises an exception of type InvalidFieldValue if called
    /// with an invalid value.
    /// @param   value Field value.
    /// @throws  Smp::IField::InvalidFieldValue
    virtual void SetValue(Smp::AnySimple value) throw (
        Smp::IField::InvalidFieldValue) = 0;
};
```

**Base Interfaces**

Smp::IField

**Operations**

| Name | Description |
|---|---|
| GetValue | Get the value of the simple field. |
| SetValue | Set the value of the simple field. |

#### 3.5.3.4.1 Get Value

Get the value of the simple field.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | AnySimple | Field value. |

**Exceptions**

None

#### 3.5.3.4.2 Set Value

Set the value of the simple field.

This method raises an exception of type InvalidFieldValue if called with an invalid value.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| value | in | AnySimple | Field value. |

**Exceptions**

Smp::IField::InvalidFieldValue

#### 3.5.3.5 IForcible Field

Interface of a forcible field.

**File**

#include "Smp/IForcibleField.h"

**Namespace**

Smp

**Declaration of IForcibleField**

```
/// Unique Identifier of type IForcibleField.
extern const Uuid Uuid_IForcibleField;

/// Interface of a forcible field.
class IForcibleField :
    public virtual Smp::ISimpleField
{
public:

    /// Virtual destructor to release memory.
    virtual ~IForcibleField() {}

    /// Force field to given value.
    /// This method raises an exception of type InvalidFieldValue if called
    /// with an invalid value.
    /// @param   value Value to force field to.
    /// @throws  Smp::IField::InvalidFieldValue
    virtual void Force(Smp::AnySimple value) throw (
        Smp::IField::InvalidFieldValue) = 0;

    /// Force field to its current value.
    virtual void Freeze() = 0;

    /// Unforce field.
    virtual void Unforce() = 0;

    /// Query for the forced state of the field.
    /// @return  Whether the field is forced or not.
    virtual Smp::Bool IsForced() = 0;
};
```

**Base Interfaces**

Smp::ISimpleField

**Operations**

| Name | Description |
|---|---|
| Force | Force field to given value. |
| Freeze | Force field to its current value. |
| IsForced | Query for the forced state of the field. |
| Unforce | Unforce field. |

##### 3.5.3.5.1 Force

Force field to given value.

This method raises an exception of type InvalidFieldValue if called with an invalid value.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| value | in | AnySimple | Value to force field to. |

**Exceptions**

Smp::IField::InvalidFieldValue

#### 3.5.3.5.2 Freeze

Force field to its current value.

**Parameters**

None

**Exceptions**

None

#### 3.5.3.5.3 Is Forced

Query for the forced state of the field.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | Bool | Whether the field is forced or not. |

**Exceptions**

None

#### 3.5.3.5.4 Unforce

Unforce field.

**Parameters**

None

**Exceptions**

None

### 3.5.3.6 View Kind

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

The simulation infrastructure must at least support the "None" and the "EndUser" roles (i.e. hidden or always visible).

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

**File**

#include "Smp/ViewKind.h"

**Namespace**

Smp

**Declaration of ViewKind**

```
/// Unique Identifier of type ViewKind.
extern const Uuid Uuid_ViewKind;

/// This enumeration defines possible options for the View attribute, which
/// can be used to control if and how an element is made visible when
/// published to the simulation infrastructure.
/// The simulation infrastructure must at least support the "None" and the
/// "EndUser" roles (i.e. hidden or always visible).
/// The simulation infrastructure may support the selection of different
/// user roles ("Debug", "Expert", "End User"), in which case the "Debug"
/// and the "Expert" role must also be supported as described.
enum ViewKind
{
    /// The element is not made visible to the user (this is the default).
    VK_None,

    /// The element is made visible for debugging purposes.
    /// The element is not visible to end users. If the simulation
    /// infrastructure supports the selection of different user roles, then
    /// the element shall be visible to "Debug" users only.
    VK_Debug,

    /// The element is made visible for expert users.
    /// The element is not visible to end users. If the simulation
    /// infrastructure supports the selection of different user roles, then
    /// the element shall be visible to "Debug" and "Expert" users.
    VK_Expert,

    /// The element is made visible to all users.
    VK_All
};
```

**Table 4 - Enumeration Literals of ViewKind**

| Name | Description |
|---|---|
| VK_None | The element is not made visible to the user (this is the default). |
| VK_Debug | The element is made visible for debugging purposes.<br>The element is not visible to end users. If the simulation infrastructure supports the selection of different user roles, then the element shall be visible to "Debug" users only. |
| VK_Expert | The element is made visible for expert users.<br>The element is not visible to end users. If the simulation infrastructure supports the selection of different user roles, then the element shall be visible to "Debug" and "Expert" users. |
| VK_All | The element is made visible to all users. |

## 3.6 Simulation Environments

A Simulation Environment has to implement the ISimulator interface to give access to the models and services. This interface is derived from IComposite to give access to at least two managed containers, namely the "Models" and "Services" containers. Finally, a simulation environment has to pass a publication server to all models in the Publishing state.

### 3.6.1 Simulators

The Simulation Environment is always in one of the defined simulator states, with well-defined state transition methods between these states.

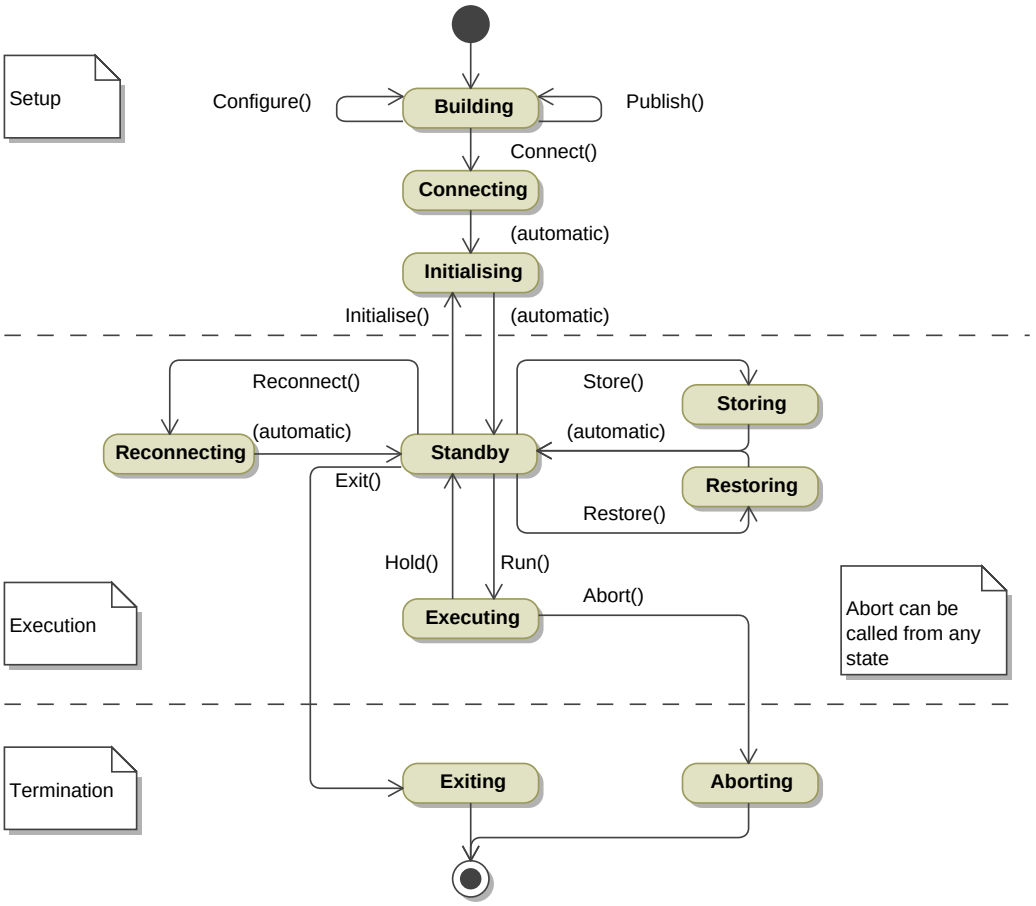


**Figure 2 - Simulation Environment State Diagram with State Transition Methods**

The available simulator states are enumerated by the SimulatorStateKind enumeration, while the ISimulator interface provides the corresponding state transition methods. Except for the Abort() state transition, which can be called from any other state, all other state transitions should be called only from the appropriate states, as shown in the Simulation Environment State Diagram in Figure 2, and explained in the following subsections. However, when calling a state transition from another state, the simulation environment shall not raise an exception, but ignore the state transition. It may use the Logger service to log a warning message.

The simulator states correspond to dedicated Model states. Therefore a model is always in one of the defined model states too, with well-defined state transition methods between these states. The available model states are enumerated by the ModelStateKind enumeration, while the IModel interface provides the corresponding state transition methods.

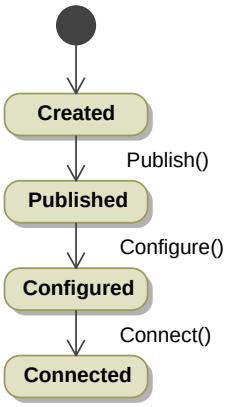


**Figure 3 - Model State Diagram with State Transition Methods**

During the Building state the Simulation Environment builds the models and calls Publish() and Configure() to initiate the transition of the models from Created via Published to Configured state. During the Connecting state the Simulation Environment initiates the transition of the models into Connected state.

#### 3.6.1.1 Simulator State Kind

This is an enumeration of the available states of the simulator. The Setup phase is split into three different states, the Execution phase has five different states, and the Termination phase has two states.

**File**

#include "Smp/ISimulator.h"

**Namespace**

Smp

**Declaration of SimulatorStateKind**

```
/// Unique Identifier of type SimulatorStateKind.
extern const Uuid Uuid_SimulatorStateKind;

/// This is an enumeration of the available states of the simulator. The
/// Setup phase is split into three different states, the Execution phase
/// has five different states, and the Termination phase has two states.
enum SimulatorStateKind
{
    /// In Building state, the model hierarchy is created. This is done by
    /// an external component, not by the simulator.
    /// This state is entered automatically after the simulation
    /// environment has performed its initialisation.
    /// This state is left with the Connect() state transition method.
    SSK_Building,

    /// In Connecting state, the simulation environment traverses the model
```

```
        /// hierarchy and calls the Connect() method of each model.
        /// This state is entered using the Connect() state transition.
        /// After connecting all models to the simulator, an automatic state
        /// transition to the Initialising state is performed.
        SSK_Connecting,

        /// In Initialising state, the simulation environment executes all
        /// initialisation entry points in the order they have been added to
        /// the simulator using the AddInitEntryPoint() method.
        /// This state is either entered automatically after the simulation
        /// environment has connected all models to the simulator, or manually
        /// from Standby state using the Initialise() state transition.
        /// After calling all initialisation entry points, an automatic state
        /// transition to the Standby state is performed.
        SSK_Initialising,

        /// In Standby state, the simulation environment (namely the Time
        /// Keeper Service) does not progress simulation time. Only entry
        /// points registered relative to Zulu time are executed.
        /// This state is entered automatically from the Initialising, Storing,
        /// and Restoring states, or manually from the Executing state using
        /// the Hold() state transition.
        /// This state is left with one of the Run(), Store(), Restore(),
        /// Initialise(), Reconnect() or Exit() state transitions.
        SSK_Standby,

        /// In Executing state, the simulation environment (namely the Time
        /// Keeper Service) does progress simulation time. Entry points
        /// registered with any of the available time kinds are executed.
        /// This state is entered using the Run() state transition.
        /// This state is left using the Hold() state transition.
        SSK_Executing,

        /// In Storing state, the simulation environment first stores the
        /// values of all fields published with the State attribute to storage
        /// (typically a file). Afterwards, the Store() method of all
        /// components (Models and Services) implementing the optional IPersist
        /// interface is called, to allow custom storing of additional
        /// information. While in this state, fields published with the State
        /// attribute must not be modified by the models, to ensure that a
        /// consistent set of field values is stored.
        /// This state is entered using the Store() state transition.
        /// After storing the simulator state, an automatic state transition to
        /// the Standby state is performed.
        SSK_Storing,

        /// In Restoring state, the simulation environment first restores the
        /// values of all fields published with the State attribute from
        /// storage. Afterwards, the Restore() method of all components
        /// implementing the optional IPersist interface is called, to allow
        /// custom restoring of additional information. While in this state,
        /// fields published with the State attribute must not be modified by
        /// the models, to ensure that a consistent set of field values is
        /// restored.
        /// This state is entered using the Restore() state transition.
        /// After restoring the simulator state, an automatic state transition
        /// to the Standby state is performed.
        SSK_Restoring,

        /// In Reconnecting state, the simulation environment makes sure that
        /// models that have been added to the simulator after leaving the
        /// Building state are properly published, configured and connected.
        /// This state is entered using the Reconnect() state transition.
        /// After connecting all new models, an automatic state transition to
        /// the Standby state is performed.
        SSK_Reconnecting,

        /// In Exiting state, the simulation environment is properly
```

```
        /// terminating a running simulation.
        /// This state is entered using the Exit() state transition. After
        /// exiting, the simulator is in an undefined state.
        SSK_Exiting,

        /// In this state, the simulation environment performs an abnormal
        /// simulation shut-down.
        /// This state is entered using the Abort() state transition. After
        /// aborting, the simulator is in an undefined state.
        SSK_Aborting
    };
```

**Table 5 - Enumeration Literals of SimulatorStateKind**

| Name | Description |
|---|---|
| SSK_Building | In Building state, the model hierarchy is created. This is done by an external component, not by the simulator.<br>This state is entered automatically after the simulation environment has performed its initialisation.<br>This state is left with the Connect() state transition method. |
| SSK_Connecting | In Connecting state, the simulation environment traverses the model hierarchy and calls the Connect() method of each model.<br>This state is entered using the Connect() state transition.<br>After connecting all models to the simulator, an automatic state transition to the Initialising state is performed. |
| SSK_Initialising | In Initialising state, the simulation environment executes all initialisation entry points in the order they have been added to the simulator using the AddInitEntryPoint() method.<br>This state is either entered automatically after the simulation environment has connected all models to the simulator, or manually from Standby state using the Initialise() state transition.<br>After calling all initialisation entry points, an automatic state transition to the Standby state is performed. |
| SSK_Standby | In Standby state, the simulation environment (namely the Time Keeper Service) does not progress simulation time. Only entry points registered relative to Zulu time are executed.<br>This state is entered automatically from the Initialising, Storing, and Restoring states, or manually from the Executing state using the Hold() state transition.<br>This state is left with one of the Run(), Store(), Restore(), Initialise(), Reconnect() or Exit() state transitions. |

| Name | Description |
|---|---|
| SSK_Executing | In Executing state, the simulation environment (namely the Time Keeper Service) does progress simulation time. Entry points registered with any of the available time kinds are executed.<br>This state is entered using the Run() state transition.<br>This state is left using the Hold() state transition. |
| SSK_Storing | In Storing state, the simulation environment first stores the values of all fields published with the State attribute to storage (typically a file). Afterwards, the Store() method of all components (Models and Services) implementing the optional IPersist interface is called, to allow custom storing of additional information. While in this state, fields published with the State attribute must not be modified by the models, to ensure that a consistent set of field values is stored.<br>This state is entered using the Store() state transition.<br>After storing the simulator state, an automatic state transition to the Standby state is performed. |
| SSK_Restoring | In Restoring state, the simulation environment first restores the values of all fields published with the State attribute from storage. Afterwards, the Restore() method of all components implementing the optional IPersist interface is called, to allow custom restoring of additional information. While in this state, fields published with the State attribute must not be modified by the models, to ensure that a consistent set of field values is restored.<br>This state is entered using the Restore() state transition.<br>After restoring the simulator state, an automatic state transition to the Standby state is performed. |
| SSK_Reconnecting | In Reconnecting state, the simulation environment makes sure that models that have been added to the simulator after leaving the Building state are properly published, configured and connected.<br>This state is entered using the Reconnect() state transition.<br>After connecting all new models, an automatic state transition to the Standby state is performed. |
| SSK_Exiting | In Exiting state, the simulation environment is properly terminating a running simulation.<br>This state is entered using the Exit() state transition. After exiting, the simulator is in an undefined state. |

| Name | Description |
|---|---|
| SSK_Aborting | In this state, the simulation environment performs an abnormal simulation shut-down.<br>This state is entered using the Abort() state transition. After aborting, the simulator is in an undefined state. |

#### 3.6.1.2 ISimulator

This interface gives access to the simulation environment state and state transitions. Further, it provides convenience methods to add models, and to add and retrieve simulation services.

This is a mandatory interface that every SMP compliant simulation environment has to implement.

**File**

#include "Smp/ISimulator.h"

**Namespace**

Smp

**Declaration of ISimulator**

```
/// Unique Identifier of type ISimulator.
extern const Uuid Uuid_ISimulator;

/// This interface gives access to the simulation environment state and
/// state transitions. Further, it provides convenience methods to add
/// models, and to add and retrieve simulation services.
/// This is a mandatory interface that every SMP compliant simulation
/// environment has to implement.
class ISimulator :
    public virtual Smp::IComposite
{
public:

    /// Virtual destructor to release memory.
    virtual ~ISimulator() {}

    /// This method asks the simulation environment to call all
    /// initialisation entry points again.
    /// This method can only be called when in Standby state, and enters
    /// Initialising state. After completion, the simulator automatically
    /// returns to Standby state.
    /// The entry points will be executed in the order they have been added
    /// to the simulator using the AddInitEntryPoint() method.
    virtual void Initialise() = 0;

    /// This method asks the simulation environment to call the Publish()
    /// method of all model instances in the model hierarchy which are
    /// still in Created state.
    ///
    /// This method must only be called when in Building state.
    /// @remarks This method is typically called by an external component
    ///          after creating new model instances, typically from
    ///          information in an SMDL Assembly.
    virtual void Publish() = 0;

    /// This method asks the simulation environment to call the Configure()
    /// method of all model instances which are still in Publishing state.
    ///
    /// This method must only be called when in Building state.
```

```
        /// @remarks This method is typically called by an external component
        ///          after setting field values of new model instances,
        ///          typically using the information in an SMDL Assembly or
        ///          SMDL Configuration.
        virtual void Configure() = 0;

        /// This method informs the simulation environment that the hierarchy
        /// of model instances has been configured, and can now be connected to
        /// the simulator. Thus, the simulation environment calls the Connect()
        /// method of all model instances.
        /// In this method, the simulation environment first enters Connecting
        /// state and calls the Connect() method of every model in the model
        /// hierarchy, then enters Initialising state and calls the
        /// initialisation entry points, and finally enters Standby state.
        /// This method must only be called when in Building state.
        /// @remarks This method is typically called by an external component
        ///          after configuring all model instances.
        virtual void Connect() = 0;

        /// This method changes from Standby to Executing state.
        /// This method must only be called when in Standby state, and enters
        /// Executing state.
        virtual void Run() = 0;

        /// This method changes from Executing to Standby state.
        /// This method must only be called when in Executing state, and enters
        /// Standby state.
        virtual void Hold() = 0;

        /// This method is used to store a state vector to file.
        /// This method must only be called when in Standby state, and enters
        /// Storing state. On completion, it automatically returns to Standby
        /// state.
        /// @param   filename Name to use for simulation state vector file.
        virtual void Store(Smp::String8 filename) = 0;

        /// This method is used to restore a state vector from file.
        /// This method must only be called when in Standby state, and enters
        /// Restoring state. On completion, it automatically returns to Standby
        /// state.
        /// @param   filename Name of simulation state vector file.
        virtual void Restore(Smp::String8 filename) = 0;

        /// This method asks the simulation environment to reconnect the
        /// component hierarchy starting at the given root component.
        ///  This method must only be called when in Standby state.
        /// @remarks This method is typically called after creating additional
        ///          model instances and adding them to the existing model
        ///          hierarchy.
        ///          The simulation environment has to ensure that all models
        ///          under the given root (but not the root itself) are
        ///          published, configured and connected, so that all child
        ///          models are finally in Connected state.
        /// @param   root Root component to start reconnecting from. This can
        ///          be the parent component of a newly added model, or e.g.
        ///          the simulator itself.
        virtual void Reconnect(Smp::IComponent* root) = 0;

        /// This method is used for a normal termination of a simulation.
        /// This method must only be called when in Standby state, and enters
        /// Exiting state.
        virtual void Exit() = 0;

        /// This method is used for an abnormal termination of a simulation.
        /// This method can be called from any other state, and enters Aborting
        /// state.
        virtual void Abort() = 0;
```

```
        /// Return the current simulator state.
        /// @return  Current simulator state.
        virtual Smp::SimulatorStateKind GetState() const = 0;

        /// This method can be used to add entry points that shall be executed
        /// in the Initialising state.
        /// @remarks The ITask interface (which is derived from IEntryPoint)
        ///          can be used to add several entry points in a well-defined
        ///          order.
        ///          The entry points will be executed in the order they have
        ///          been added to the simulator.
        /// @param   entryPoint Entry point to execute in Initialising state.
        virtual void AddInitEntryPoint(Smp::IEntryPoint* entryPoint) = 0;

        /// This method adds a model to the models collection of the simulator,
        /// i.e. to the "Models" container.
        /// This method raises an exception of type DuplicateName if the name
        /// of the new model conflicts with the name of an existing component
        /// (model or service).
        /// The container for the models has no upper limit and thus the
        /// ContainerFull exception will never be thrown.
        /// The method will never throw the InvalidObjectType exception either,
        /// as it gets a component implementing the IModel interface.
        /// @param   model New model to add to collection of root models, i.e.
        ///          to the "Models" container.
        /// @throws  Smp::DuplicateName
        virtual void AddModel(Smp::IModel* model) throw (
            Smp::DuplicateName) = 0;

        /// This method adds a user-defined service to the services collection,
        /// i.e. to the "Services" container.
        /// This method raises an exception of type DuplicateName if the name
        /// of the new service conflicts with the name of an existing component
        /// (model or service).
        /// The container for the services has no upper limit and thus the
        /// ContainerFull exception will never be thrown.
        /// The method will never throw the InvalidObjectType exception either,
        /// as it gets a component implementing the IService interface.
        /// @remarks It is recommended that custom services include a project
        ///          or company acronym as prefix in their name, to avoid
        ///          collision of service names.
        ///
        ///          The container for the services has no upper limit and thus
        ///          the ContainerFull exception will never be thrown.
        ///          The method will never throw the InvalidObjectType
        ///          exception, as it expects a component implementing the
        ///          IService interface.
        /// @param   service Service to add to services collection.
        /// @throws  Smp::DuplicateName
        virtual void AddService(Smp::IService* service) throw (
            Smp::DuplicateName) = 0;

        /// Query for a service component by its name.
        /// The returned component is null if no service with the given name
        /// could be found. Standard names are defined for the standardised
        /// services, while custom services use custom names.
        /// The existence of custom services is not guaranteed, so models
        /// should expect to get a null reference.
        /// @param   name Service name.
        /// @return  Service with the given name, or null if no service with
        ///          the given name could be found.
        virtual Smp::IService* GetService(Smp::String8 name) const = 0;

        /// Return interface to logger service.
        /// @remarks This is a type-safe convenience method, to avoid having to
        ///          use the generic GetService() method. For the standardised
        ///          services, it is recommended to use the convenience
        ///          methods, which are guaranteed to return a valid reference.
```

```
        /// @return  Interface to mandatory logger service.
        virtual Smp::Services::ILogger* GetLogger() const = 0;

        /// Return interface to time keeper service.
        /// @remarks This is a type-safe convenience method, to avoid having to
        ///          use the generic GetService() method. For the standardised
        ///          services, it is recommended to use the convenience
        ///          methods, which are guaranteed to return a valid reference.
        /// @return  Interface to mandatory time keeper service.
        virtual Smp::Services::ITimeKeeper* GetTimeKeeper() const = 0;

        /// Return interface to scheduler service.
        /// @remarks This is a type-safe convenience method, to avoid having to
        ///          use the generic GetService() method. For the standardised
        ///          services, it is recommended to use the convenience
        ///          methods, which are guaranteed to return a valid reference.
        /// @return  Interface to mandatory scheduler service.
        virtual Smp::Services::IScheduler* GetScheduler() const = 0;

        /// Return interface to event manager service.
        /// @remarks This is a type-safe convenience method, to avoid having to
        ///          use the generic GetService() method. For the standardised
        ///          services, it is recommended to use the convenience
        ///          methods, which are guaranteed to return a valid reference.
        /// @return  Interface to mandatory event manager service.
        virtual Smp::Services::IEventManager* GetEventManager() const = 0;

        /// Return interface to resolver service.
        /// @remarks This is a type-safe convenience method, to avoid having to
        ///          use the generic GetService() method. For the standardised
        ///          services, it is recommended to use the convenience
        ///          methods, which are guaranteed to return a valid reference.
        /// @return  Interface to mandatory resolver service.
        virtual Smp::Services::IResolver* GetResolver() const = 0;

        /// Return interface to link registry service.
        /// @remarks This is a type-safe convenience method, to avoid having to
        ///          use the generic GetService() method. For the standardised
        ///          services, it is recommended to use the convenience
        ///          methods, which are guaranteed to return a valid reference.
        /// @return  Interface to mandatory link registry service.
        virtual Smp::Services::ILinkRegistry* GetLinkRegistry() const = 0;
    };
```

**Base Interfaces**

Smp::IComposite

**Constants**

| Name | Type | Description | Value |
|---|---|---|---|
| SMP_SimulatorModels | String8 | Name of the model container. | Models |
| SMP_SimulatorServices | String8 | Name of the service container. | Services |

**Operations**

| Name | Description |
|---|---|
| Initialise | This method asks the simulation environment to call all initialisation entry points again. |

| Name | Description |
|---|---|
| Publish | This method asks the simulation environment to call the Publish() method of all model instances in the model hierarchy which are still in Created state. |
| Configure | This method asks the simulation environment to call the Configure() method of all model instances which are still in Publishing state. |
| Connect | This method informs the simulation environment that the hierarchy of model instances has been configured, and can now be connected to the simulator. Thus, the simulation environment calls the Connect() method of all model instances. |
| Run | This method changes from Standby to Executing state. |
| Hold | This method changes from Executing to Standby state. |
| Store | This method is used to store a state vector to file. |
| Restore | This method is used to restore a state vector from file. |
| Reconnect | This method asks the simulation environment to reconnect the component hierarchy starting at the given root component. |
| Exit | This method is used for a normal termination of a simulation. |
| Abort | This method is used for an abnormal termination of a simulation. |
| GetState | Return the current simulator state. |
| AddInitEntryPoint | This method can be used to add entry points that shall be executed in the Initialising state. |
| AddModel | This method adds a model to the models collection of the simulator, i.e. to the "Models" container. |
| AddService | This method adds a user-defined service to the services collection, i.e. to the "Services" container. |
| GetService | Query for a service component by its name. |
| GetLogger | Return interface to logger service. |
| GetTimeKeeper | Return interface to time keeper service. |
| GetScheduler | Return interface to scheduler service. |
| GetEventManager | Return interface to event manager service. |
| GetResolver | Return interface to resolver service. |
| GetLinkRegistry | Return interface to link registry service. |

#### 3.6.1.2.1 Initialise

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

This method can only be called when in Standby state, and enters Initialising state. After completion, the simulator automatically returns to Standby state.

The entry points will be executed in the order they have been added to the simulator using the AddInitEntryPoint() method.

**Parameters**

None

**Exceptions**

None

#### 3.6.1.2.2 Publish

This method asks the simulation environment to call the Publish() method of all model instances in the model hierarchy which are still in Created state.

This method must only be called when in Building state.

***Remark:*** This method is typically called by an external component after creating new model instances, typically from information in an SMDL Assembly.

**Parameters**

None

**Exceptions**

None

#### 3.6.1.2.3 Configure

This method asks the simulation environment to call the Configure() method of all model instances which are still in Publishing state.

This method must only be called when in Building state.

***Remark:*** This method is typically called by an external component after setting field values of new model instances, typically using the information in an SMDL Assembly or SMDL Configuration.

**Parameters**

None

**Exceptions**

None

#### 3.6.1.2.4 Connect

This method informs the simulation environment that the hierarchy of model instances has been configured, and can now be connected to the simulator. Thus, the simulation environment calls the Connect() method of all model instances.

In this method, the simulation environment first enters Connecting state and calls the Connect() method of every model in the model hierarchy, then enters Initialising state and calls the initialisation entry points, and finally enters Standby state.

This method must only be called when in Building state.

***Remark:*** This method is typically called by an external component after configuring all model instances.

**Parameters**

None

**Exceptions**

None

#### 3.6.1.2.5 Run

This method changes from Standby to Executing state.

This method must only be called when in Standby state, and enters Executing state.

**Parameters**

None

**Exceptions**

None

#### 3.6.1.2.6 Hold

This method changes from Executing to Standby state.

This method must only be called when in Executing state, and enters Standby state.

**Parameters**

None

**Exceptions**

None

#### 3.6.1.2.7 Store

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

This method must only be called when in Standby state, and enters Storing state. On completion, it automatically returns to Standby state.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| filename | in | String8 | Name to use for simulation state vector file. |

**Exceptions**

None

#### 3.6.1.2.8 Restore

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

This method must only be called when in Standby state, and enters Restoring state. On completion, it automatically returns to Standby state.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| filename | in | String8 | Name of simulation state vector file. |

**Exceptions**

None

#### 3.6.1.2.9 Reconnect

This method asks the simulation environment to reconnect the component hierarchy starting at the given root component.

This method must only be called when in Standby state.

The simulation environment has to ensure that all models under the given root (but not the root itself) are published, configured and connected, so that all child models are finally in Connected state.

***Remark:*** This method is typically called after creating additional model instances and adding them to the existing model hierarchy.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| root | in | IComponent | Root component to start reconnecting from. This can be the parent component of a newly added model, or e.g. the simulator itself. |

**Exceptions**

None

#### 3.6.1.2.10 Exit

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

This method must only be called when in Standby state, and enters Exiting state.

**Parameters**

None

**Exceptions**

None

#### 3.6.1.2.11 Abort

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

This method can be called from any other state, and enters Aborting state.

**Parameters**

None

**Exceptions**

None

#### 3.6.1.2.12 Get State

Return the current simulator state.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | SimulatorStateKind | Current simulator state. |

**Exceptions**

None

#### 3.6.1.2.13 Add Init Entry Point

This method can be used to add entry points that shall be executed in the Initialising state.

The entry points will be executed in the order they have been added to the simulator.

***Remark:*** The ITask interface (which is derived from IEntryPoint) can be used to add several entry points in a well-defined order.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| entryPoint | in | IEntryPoint | Entry point to execute in Initialising state. |

**Exceptions**

None

#### 3.6.1.2.14 Add Model

This method adds a model to the models collection of the simulator, i.e. to the "Models" container.

This method raises an exception of type DuplicateName if the name of the new model conflicts with the name of an existing component (model or service).

The container for the models has no upper limit and thus the ContainerFull exception will never be thrown.

The method will never throw the InvalidObjectType exception either, as it gets a component implementing the IModel interface.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| model | in | IModel | New model to add to collection of root models, i.e. to the "Models" container. |

**Exceptions**

Smp::DuplicateName

#### 3.6.1.2.15 Add Service

This method adds a user-defined service to the services collection, i.e. to the "Services" container.

This method raises an exception of type DuplicateName if the name of the new service conflicts with the name of an existing component (model or service).

The container for the services has no upper limit and thus the ContainerFull exception will never be thrown.

The method will never throw the InvalidObjectType exception either, as it gets a component implementing the IService interface.

The container for the services has no upper limit and thus the ContainerFull exception will never be thrown.

The method will never throw the InvalidObjectType exception, as it expects a component implementing the IService interface.

***Remark:*** It is recommended that custom services include a project or company acronym as prefix in their name, to avoid collision of service names.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| service | in | IService | Service to add to services collection. |

**Exceptions**

Smp::DuplicateName

#### 3.6.1.2.16 Get Service

Query for a service component by its name.

The returned component is null if no service with the given name could be found. Standard names are defined for the standardised services, while custom services use custom names.

The existence of custom services is not guaranteed, so models should expect to get a null reference.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | IService | Service with the given name, or null if no service with the given name could be found. |
| name | in | String8 | Service name. |

**Exceptions**

None

#### 3.6.1.2.17 Get Logger

Return interface to logger service.

***Remark:*** This is a type-safe convenience method, to avoid having to use the generic GetService() method. For the standardised services, it is recommended to use the convenience methods, which are guaranteed to return a valid reference.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | ILogger | Interface to mandatory logger service. |

**Exceptions**

None

#### 3.6.1.2.18 Get Time Keeper

Return interface to time keeper service.

***Remark:*** This is a type-safe convenience method, to avoid having to use the generic GetService() method. For the standardised services, it is recommended to use the convenience methods, which are guaranteed to return a valid reference.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | ITimeKeeper | Interface to mandatory time keeper service. |

**Exceptions**

None

#### 3.6.1.2.19 Get Scheduler

Return interface to scheduler service.

***Remark:*** This is a type-safe convenience method, to avoid having to use the generic GetService() method. For the standardised services, it is recommended to use the convenience methods, which are guaranteed to return a valid reference.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | IScheduler | Interface to mandatory scheduler service. |

**Exceptions**

None

#### 3.6.1.2.20 Get Event Manager

Return interface to event manager service.

***Remark:*** This is a type-safe convenience method, to avoid having to use the generic GetService() method. For the standardised services, it is recommended to use the convenience methods, which are guaranteed to return a valid reference.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | IEventManager | Interface to mandatory event manager service. |

**Exceptions**

None

#### 3.6.1.2.21 Get Resolver

Return interface to resolver service.

***Remark:*** This is a type-safe convenience method, to avoid having to use the generic GetService() method. For the standardised services, it is recommended to use the convenience methods, which are guaranteed to return a valid reference.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | IResolver | Interface to mandatory resolver service. |

**Exceptions**

None

#### 3.6.1.2.22 Get Link Registry

Return interface to link registry service.

***Remark:*** This is a type-safe convenience method, to avoid having to use the generic GetService() method. For the standardised services, it is recommended to use the convenience methods, which are guaranteed to return a valid reference.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | ILinkRegistry | Interface to mandatory link registry service. |

**Exceptions**

None

### 3.6.1.3 IManaged Simulator

This interface gives access to a managed simulator.

This interface extends the ISimulator interface and adds methods to create components (typically models) from component factories. It makes use of the IFactory interface for component factories.

This is an optional interface the simulation environment may implement.

**File**

#include "Smp/Management/IManagedSimulator.h"

**Namespace**

Smp::Management

**Declaration of IManagedSimulator**

```
/// Unique Identifier of type IManagedSimulator.
extern const Uuid Uuid_IManagedSimulator;

/// This interface gives access to a managed simulator.
/// This interface extends the ISimulator interface and adds methods to
/// create components (typically models) from component factories. It
/// makes use of the IFactory interface for component factories.
```

```
        /// This is an optional interface the simulation environment may
        /// implement.
        class IManagedSimulator :
            public virtual Smp::ISimulator
        {
        public:

            /// Virtual destructor to release memory.
            virtual ~IManagedSimulator() {}

            /// This method registers a component factory with the managed
            /// simulator. The managed simulator can use this factory to create
            /// component instances of the component implementation in its
            /// CreateInstance() method.
            /// This method raises an exception of type DuplicateUuid if
            /// another factory has been registered using the same
            /// implementation identifier already.
            /// @remarks This method is typically called early in the Building
            ///          state to register the available component factories
            ///          before the hierarchy of model instances is created.
            /// @param   componentFactory Factory to create instance of the
            ///          component implementation.
            /// @throws  Smp::Management::IManagedSimulator::DuplicateUuid
            virtual void RegisterFactory(Smp::IFactory* componentFactory) throw (
                Smp::Management::IManagedSimulator::DuplicateUuid) = 0;

            /// This method creates an instance of the component with the given
            /// implementation identifier.
            /// @remarks This method is typically called during Creating state
            ///          when building the hierarchy of models.
            /// @param   implUuid Implementation identifier of the component.
            /// @return  New instance of the component with the given
            ///          implementation identifier or null in case no factory
            ///          for the given implementation identifier has been
            ///          registered.
            virtual Smp::IComponent* CreateInstance(Smp::Uuid implUuid) = 0;

            /// This method returns the factory of the component with the given
            /// implementation identifier.
            /// @param   implUuid Implementation identifier of the component.
            /// @return  Factory of the component with the given implementation
            ///          identifier or null in case no factory for the given
            ///          implementation identifier has been registered.
            virtual Smp::IFactory* GetFactory(Smp::Uuid implUuid) const = 0;

            /// This method returns 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.
            /// @param   specUuid Specification identifier of the component.
            /// @return  Collection of factories for the given specification
            ///          identifier.
            virtual const Smp::FactoryCollection* GetFactories(Smp::Uuid specUuid)
const = 0;
        };
```

**Base Interfaces**

Smp::ISimulator

**Operations**

| Name | Description |
|---|---|
| CreateInstance | This method creates an instance of the component with the given implementation identifier. |

| Name | Description |
|---|---|
| GetFactories | This method returns all factories of components with the given specification identifier. |
| GetFactory | This method returns the factory of the component with the given implementation identifier. |
| RegisterFactory | This method registers a component factory with the managed simulator. The managed simulator can use this factory to create component instances of the component implementation in its CreateInstance() method. |

#### 3.6.1.3.1 Create Instance

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

***Remark:*** This method is typically called during Creating state when building the hierarchy of models.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | IComponent | New instance of the component with the given implementation identifier or null in case no factory for the given implementation identifier has been registered. |
| implUuid | in | Uuid | Implementation identifier of the component. |

**Exceptions**

None

#### 3.6.1.3.2 Get Factories

This method returns 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.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | FactoryCollection | Collection of factories for the given specification identifier. |
| specUuid | in | Uuid | Specification identifier of the component. |

**Exceptions**

None

#### 3.6.1.3.3 Get Factory

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

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | IFactory | Factory of the component with the given implementation identifier or null in case no factory for the given implementation identifier has been registered. |
| implUuid | in | Uuid | Implementation identifier of the component. |

**Exceptions**

None

#### 3.6.1.3.4 Register Factory

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

This method raises an exception of type DuplicateUuid if another factory has been registered using the same implementation identifier already.

***Remark:*** This method is typically called early in the Building state to register the available component factories before the hierarchy of model instances is created.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| componentFactory | in | IFactory | Factory to create instance of the component implementation. |

**Exceptions**

Smp::Management::IManagedSimulator::DuplicateUuid

#### 3.6.1.3.5 Duplicate Uuid

This exception is raised when trying to register a factory under a Uuid that has already been used to register another (or the same) factory.This would lead to duplicate implementation Uuids.

**File**

#include "Smp/Management/IManagedSimulator.h"

**Namespace**

Smp::Management::IManagedSimulator

**Declaration of DuplicateUuid**

```
/// This exception is raised when trying to register a factory
/// under a Uuid that has already been used to register another (or
/// the same) factory.This would lead to duplicate implementation
/// Uuids.
class DuplicateUuid : public Smp::Exception
```

```
        {
        public:
            /// Constructor for new exception.
            /// @param   newName Name of factory that tried to register
            ///          under this Uuid.
            /// @param   oldName Name of factory already registered under
            ///          this Uuid.
            DuplicateUuid(
                Smp::String8 newName,
                Smp::String8 oldName) throw();

            /// Copy constructor.
            DuplicateUuid(
                DuplicateUuid& ex) throw();

            /// Virtual destructor to release memory.
            virtual ~DuplicateUuid();

            /// Name of factory that tried to register under this Uuid.
            Smp::String8 newName;

            /// Name of factory already registered under this Uuid.
            Smp::String8 oldName;
        };
```

**Fields**

| Name | Type | Description |
|---|---|---|
| newName | String8 | Name of factory that tried to register under this Uuid. |
| oldName | String8 | Name of factory already registered under this Uuid. |

#### 3.6.1.4 IFactory

Interface for a component factory.

**File**

#include "Smp/IFactory.h"

**Namespace**

Smp

**Declaration of IFactory**

```
/// Unique Identifier of type IFactory.
extern const Uuid Uuid_IFactory;

/// Interface for a component factory.
class IFactory :
    public virtual Smp::IObject
{
public:

    /// Virtual destructor to release memory.
    virtual ~IFactory() {}

    /// Get specification identifier of factory.
    /// @return  Universally unique identifier of component specification.
    virtual Smp::Uuid GetSpecification() const = 0;

    /// Get implementation identifier of factory.
    /// @return  Universally unique identifier of component implementation.
    virtual Smp::Uuid GetImplementation() const = 0;
```

```
        /// Create a new instance.
        /// @return  New component instance.
        virtual Smp::IComponent* CreateInstance() = 0;

        /// Delete an existing instance.
        /// @param   instance Instance to delete.
        virtual void DeleteInstance(Smp::IComponent* instance) = 0;
    };
```

**Base Interfaces**

Smp::IObject

**Operations**

| Name | Description |
|---|---|
| CreateInstance | Create a new instance. |
| DeleteInstance | Delete an existing instance. |
| GetImplementation | Get implementation identifier of factory. |
| GetSpecification | Get specification identifier of factory. |

#### 3.6.1.4.1 Create Instance

Create a new instance.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | IComponent | New component instance. |

**Exceptions**

None

#### 3.6.1.4.2 Delete Instance

Delete an existing instance.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| instance | in | IComponent | Instance to delete. |

**Exceptions**

None

#### 3.6.1.4.3 Get Implementation

Get implementation identifier of factory.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | Uuid | Universally unique identifier of component implementation. |

**Exceptions**

None

#### 3.6.1.4.4 Get Specification

Get specification identifier of factory.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | Uuid | Universally unique identifier of component specification. |

**Exceptions**

None

### 3.6.1.5 Factory Collection

A factory collection is an ordered collection of factories, which allows iterating all members.

This type is platform specific. For details see the SMP Platform Mappings.

**File**

#include "Smp/IFactory.h"

**Namespace**

Smp

**Declaration of FactoryCollection**

```
/// Unique Identifier of type FactoryCollection.
extern const Uuid Uuid_FactoryCollection;

/// A factory collection is an ordered collection of factories, which
/// allows iterating all members.
///
/// This type is platform specific. For details see the SMP Platform
/// Mappings.
typedef std::vector<IFactory*> FactoryCollection;
```

## 3.6.2 Publication

As part of the initialisation, every model needs to be given access to a publication receiver to publish its fields and operations. While the simulation environment does not have to implement the IPublication interface itself, it has to provide a publication receiver to each model during the Publishing state.

As the publication mechanism for C++ is complex, it has been moved to a dedicated section 5 (Publication).

## 3.6.3 Service Acquisition

Simulation services are closely related to models: Models need a mechanism to acquire simulation services, and most services interact with models. Further,

simulation services are themselves components of the SMP Component Model. Therefore, this document puts simulation services into context with models and the simulation environment.

When the simulation environment connects a model, it calls the Connect() method of the IModel interface, passing it a global reference of ISimulator. A model can either immediately use this reference to query services, or store the reference to query services on demand.

Before a Model can call an operation of a Service, the following steps are needed:

1. The Simulator calls the Connect() operation of the Model, passing it a reference to itself.
2. The Model calls the GetService() operation of the Simulator, passing it the name of the required service.
3. The Simulator returns the required Service to the Model.
4. The Model calls the desired operation of the Service.
5. The Service returns the desired value to the Model.
6. The Model returns control to the Simulator.

If the model keeps a reference to the service, it can call the service again at any other time.

7. The Model calls the desired operation of the Service.
8. The Service returns the desired value to the Model.

These steps are shown in a sequence diagram in Figure 4.

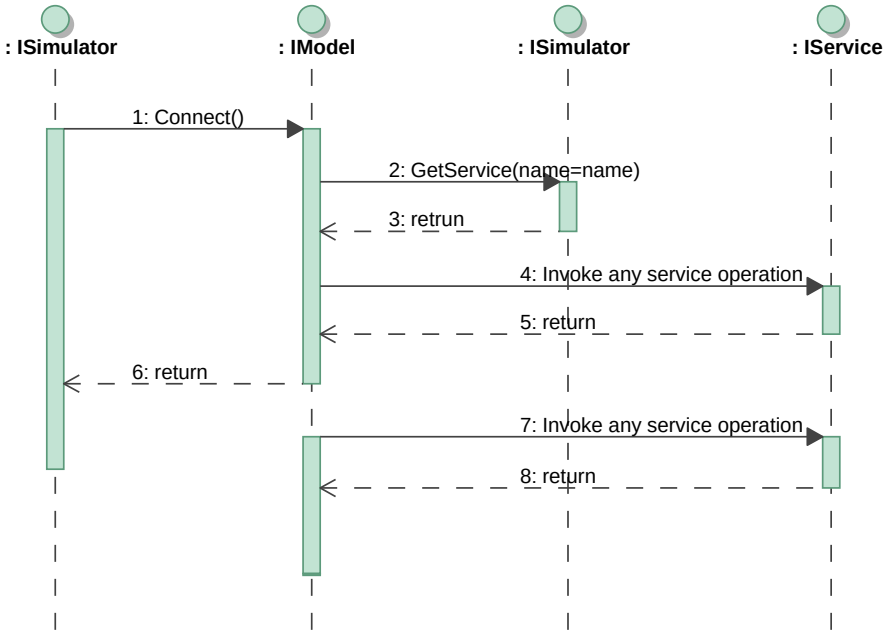


**Figure 4 - Sequence of calls for service acquisition**

# 4
# Simulation Services

In order to facilitate the inter-operability between SMP compliant simulation environments (i.e. run-time simulation kernels), several *Simulation Services* are defined in the SMP specification. All services are mandatory.

Any SMP compliant simulation environment *shall* support the following standard services.

## 4.1 Logger

The Logger service provides a method to send a log message to the simulation log.

### 4.1.1 ILogger

This interface gives access to the Logger Service.

All objects in a simulation can log messages using this service. Objects can either use pre-defined log message kinds, or define their own message kinds.

**File**

#include "Smp/Services/ILogger.h"

**Namespace**

Smp::Services

**Declaration of ILogger**

```
/// Unique Identifier of type ILogger.
extern const Uuid Uuid_ILogger;

/// This interface gives access to the Logger Service.
/// All objects in a simulation can log messages using this service.
/// Objects can either use pre-defined log message kinds, or define
/// their own message kinds.
class ILogger :
    public virtual Smp::IService
{
public:

    /// Virtual destructor to release memory.
    virtual ~ILogger() {}

    /// Return identifier of log message kind by name.
    ///
    /// @remarks This method can be used for predefined log message
    ///          kinds, but is especially useful for user-defined log
    ///          message kinds.
    ///          It is guaranteed that this method always returns the
    ///          same id for the same messageKindName string.
    /// @param   messageKindName Name of log message kind.
```

```
            /// @return  Identifier of log message kind.
            virtual Smp::Services::LogMessageKind QueryLogMessageKind(Smp::String8
messageKindName) = 0;

            /// This function logs a message to the simulation log.
            /// @param   sender Object that sends the message.
            /// @param   message The message to log.
            /// @param   kind Kind of message.
            virtual void Log(
                const Smp::IObject* sender,
                Smp::String8 message,
                Smp::Services::LogMessageKind kind = 0) = 0;
        };
```

**Base Interfaces**

Smp::IService

**Constants**

| Name | Type | Description | Value |
|---|---|---|---|
| LMK_Debug | LogMessageKind | The message contains debug information. | 4 |
| LMK_DebugName | String8 | The message contains debug information. | Debug |
| LMK_Error | LogMessageKind | The message has been raised because of an error. | 3 |
| LMK_ErrorName | String8 | The message has been raised because of an error. | Error |
| LMK_Event | LogMessageKind | The message has been sent from an event, typically from a state transition. | 1 |
| LMK_EventName | String8 | The message has been sent from an event, typically from a state transition. | Event |
| LMK_Information | LogMessageKind | The message contains general information. | 0 |
| LMK_InformationName | String8 | The message contains general information. | Information |
| LMK_Warning | LogMessageKind | The message contains a warning. | 2 |
| LMK_WarningName | String8 | The message contains a warning. | Warning |
| SMP_Logger | String8 | Name of the logger service. | Logger |

**Operations**

| Name | Description |
|---|---|
| Log | This function logs a message to the simulation log. |
| QueryLogMessageKind | Return identifier of log message kind by name. |

#### 4.1.1.1 Log

This function logs a message to the simulation log.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| sender | in | IObject | Object that sends the message. |
| message | in | String8 | The message to log. |
| kind | in | LogMessageKind | Kind of message. |

**Exceptions**

None

#### 4.1.1.2 Query Log Message Kind

Return identifier of log message kind by name.

It is guaranteed that this method always returns the same id for the same messageKindName string.

***Remark:*** This method can be used for predefined log message kinds, but is especially useful for user-defined log message kinds.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | LogMessageKind | Identifier of log message kind. |
| messageKindName | in | String8 | Name of log message kind. |

**Exceptions**

None

### 4.1.2 Log Message Kind

This type is used as identifier of a log message kind.

**File**

#include "Smp/Services/ILogger.h"

**Namespace**

Smp::Services

**Declaration of LogMessageKind**

```
/// Unique Identifier of type LogMessageKind.
extern const Uuid Uuid_LogMessageKind;

/// This type is used as identifier of a log message kind.
typedef Smp::Int32 LogMessageKind;

/// The message contains general information.
const Smp::Services::LogMessageKind LMK_Information = 0;

/// The message has been sent from an event, typically from a state
/// transition.
const Smp::Services::LogMessageKind LMK_Event = 1;

/// The message contains a warning.
const Smp::Services::LogMessageKind LMK_Warning = 2;

/// The message has been raised because of an error.
const Smp::Services::LogMessageKind LMK_Error = 3;

/// The message contains debug information.
const Smp::Services::LogMessageKind LMK_Debug = 4;

/// The message contains general information.
const Smp::String8 LMK_InformationName = "Information";

/// The message contains debug information.
const Smp::String8 LMK_DebugName = "Debug";

/// The message has been raised because of an error.
const Smp::String8 LMK_ErrorName = "Error";

/// The message contains a warning.
const Smp::String8 LMK_WarningName = "Warning";

/// The message has been sent from an event, typically from a state
/// transition.
const Smp::String8 LMK_EventName = "Event";
```

**Table 6 - Specification of LogMessageKind**

| Minimum | Maximum | Primitive Type |
|---|---|---|
| 0 | 2147483647 | Int32 |

### 4.1.3 Predefined Log Message Kinds

When logging a message with the logger service, an additional kind parameter is passed to the Log() method to identify the kind of message. The application can use any valid number, for example to allow filtering messages by message kind. However, the standard pre-defines a few message kinds that are assumed to be used in most simulations.

| Message Kind | Id | Description |
|---|---|---|
| Information | 0 | The message contains general information. |
| Event | 1 | The message has been sent from an event, typically from a state transition. |
| Warning | 2 | The message contains a warning. |

| | | |
|---|---|---|
| Error | 3 | The message has been raised because of an error. |
| Debug | 4 | The message contains debug information. |

### 4.1.4 User defined Log Message Kinds

With the QueryLogMessageKind() method, it is possible to add a user-defined log message kind to the logger service. The first call of this method with a user-defined message kind name returns a new, unique identifier that can be used as third parameter for the Log() method. Further calls of the method QueryLogMessageKind() with the same user-defined message kind name are guaranteed to return the same identifier again.

This mechanism allows using a user-defined log message kind within several different models, without the need to store the log message kind identifier into a global variable. Further, it assigns a user-readable name to each log message kind, so that e.g. a log message viewer can show log message kinds by name rather than by identifier.

## 4.2 Time Keeper

SMP supports four different kinds of time. The time managed by the Time Keeper simulation service is called Simulation Time. The service keeps track of simulation time and puts it into relation with epoch time and mission time. Further, the service provides Zulu time based on the clock of the computer.

### 4.2.1 ITime Keeper

This interface gives access to the Time Keeper Service.

Components can query for the time (using the four available time kinds), and can change the epoch or mission time.

**File**

#include "Smp/Services/ITimeKeeper.h"

**Namespace**

Smp::Services

**Declaration of ITimeKeeper**

```
/// Unique Identifier of type ITimeKeeper.
extern const Uuid Uuid_ITimeKeeper;

/// This interface gives access to the Time Keeper Service.
/// Components can query for the time (using the four available time
/// kinds), and can change the epoch or mission time.
class ITimeKeeper :
    public virtual Smp::IService
{
public:

    /// Virtual destructor to release memory.
    virtual ~ITimeKeeper() {}

    /// Return Simulation time.
    /// Simulation time is a relative time that starts at 0.
```

```
            /// @return  Current simulation time.
            virtual Smp::Duration GetSimulationTime() const = 0;

            /// Return Epoch time.
            ///  Epoch time is an absolute time with a fixed offset to
            /// simulation time. Epoch time typically progresses with
            /// simulation time, but can be changed with SetEpochTime.
            /// @return  Current epoch time.
            virtual Smp::DateTime GetEpochTime() const = 0;

            /// Return Mission time.
            /// Mission time is a relative time with a fixed offset to
            /// simulation time. Mission time typically progresses with
            /// simulation time, but can be changed with the two methods
            /// SetMissionTime and SetMissionStart. Further, mission time is
            /// updated when changing epoch time with SetEpochTime.
            /// @return  Current mission time.
            virtual Smp::Duration GetMissionTime() const = 0;

            /// Return Zulu time.
            /// Zulu time is a system dependent time and not related to
            /// simulation time. Zulu time is typically related to the system
            /// clock of the computer.
            /// @return  Current Zulu time.
            virtual Smp::DateTime GetZuluTime() const = 0;

            /// Set Epoch time.
            ///
            ///  Changes the offset between simulation time and epoch time.
            /// Calling this method shall raise a global EpochTimeChanged event
            /// in the Event Manager.
            /// @param   epochTime New epoch time.
            virtual void SetEpochTime(Smp::DateTime epochTime) = 0;

            /// Set Mission time by defining the mission start time.
            /// Changes the offset between simulation time and mission time.
            /// The mission time itself will be calculated as the offset
            /// between the current epoch time and the given mission start:
            /// MissionTime = EpochTime - MissionStart
            ///
            /// Calling this method shall raise a global MissionTimeChanged
            /// event in the Event Manager.
            /// @param   missionStart New mission start date and time.
            virtual void SetMissionStart(Smp::DateTime missionStart) = 0;

            /// Set Mission time.
            /// Changes the offset between simulation time and mission time.
            /// Calling this method shall raise a global MissionTimeChanged
            /// event in the Event Manager.
            /// @param   missionTime New mission time.
            virtual void SetMissionTime(Smp::Duration missionTime) = 0;
        };
```

**Base Interfaces**

Smp::IService

**Constants**

| Name | Type | Description | Value |
|---|---|---|---|
| SMP_TimeKeeper | String8 | Name of the TimeKeeper service. | TimeKeeper |

**Operations**

| Name | Description |
|---|---|
| GetEpochTime | Return Epoch time. |
| GetMissionTime | Return Mission time. |
| GetSimulationTime | Return Simulation time. |
| GetZuluTime | Return Zulu time. |
| SetEpochTime | Set Epoch time. |
| SetMissionStart | Set Mission time by defining the mission start time. |
| SetMissionTime | Set Mission time. |

#### 4.2.1.1 Get Epoch Time

Return Epoch time.

Epoch time is an absolute time with a fixed offset to simulation time. Epoch time typically progresses with simulation time, but can be changed with SetEpochTime.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | DateTime | Current epoch time. |

**Exceptions**

None

#### 4.2.1.2 Get Mission Time

Return Mission time.

Mission time is a relative time with a fixed offset to simulation time. Mission time typically progresses with simulation time, but can be changed with the two methods SetMissionTime and SetMissionStart. Further, mission time is updated when changing epoch time with SetEpochTime.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | Duration | Current mission time. |

**Exceptions**

None

#### 4.2.1.3 Get Simulation Time

Return Simulation time.

Simulation time is a relative time that starts at 0.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | Duration | Current simulation time. |

**Exceptions**

None

#### 4.2.1.4 Get Zulu Time

Return Zulu time.

Zulu time is a system dependent time and not related to simulation time. Zulu time is typically related to the system clock of the computer.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | DateTime | Current Zulu time. |

**Exceptions**

None

#### 4.2.1.5 Set Epoch Time

Set Epoch time.

Changes the offset between simulation time and epoch time.

Calling this method shall raise a global EpochTimeChanged event in the Event Manager.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| epochTime | in | DateTime | New epoch time. |

**Exceptions**

None

#### 4.2.1.6 Set Mission Start

Set Mission time by defining the mission start time.

Changes the offset between simulation time and mission time. The mission time itself will be calculated as the offset between the current epoch time and the given mission start:

MissionTime = EpochTime - MissionStart

Calling this method shall raise a global MissionTimeChanged event in the Event Manager.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| missionStart | in | DateTime | New mission start date and time. |

**Exceptions**

None

#### 4.2.1.7 Set Mission Time

Set Mission time.

Changes the offset between simulation time and mission time.

Calling this method shall raise a global MissionTimeChanged event in the Event Manager.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| missionTime | in | Duration | New mission time. |

**Exceptions**

None

### 4.2.2 Time Kind

Enumeration of supported time kinds.

**File**

#include "Smp/Services/IScheduler.h"

**Namespace**

Smp::Services

**Declaration of TimeKind**

```
/// Unique Identifier of type TimeKind.
extern const Uuid Uuid_TimeKind;

/// Enumeration of supported time kinds.
enum TimeKind
{
    /// Simulation time.
    TK_SimulationTime,

    /// Mission time.
    TK_MissionTime,

    /// Epoch time.
    TK_EpochTime,

    /// Zulu time.
    TK_ZuluTime
};
```

**Table 7 - Enumeration Literals of TimeKind**

| Name | Description |
|---|---|
| TK_SimulationTime | Simulation time.<br>Simulation time is a relative time. It does only exist within the time keeper service. The following holds for simulation time:<br>• Simulation time is a non-negative value measured in nanoseconds, which is the lowest level of granularity supported for time in SMP.<br>• Simulation time is stored in a signed 64-bit integer value. This allows specifying time values of more than 290 years.<br>• Simulation time can be queried using the GetSimulationTime() method of the time keeper (via the ITimeKeeper interface).<br>• 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.<br>• Simulation time is only progressed when the simulation environment is in Executing state.<br>• When storing a state vector, simulation time is stored as well.<br>• When restoring a state vector, simulation time is restored as well.<br>The standard does not define how quickly simulation time is progressed when the simulator is in Executing state. Typical examples are:<br>• **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, while 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.<br>• **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.<br>• **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 |

| Name | Description |
|---|---|
| | scheduler, which underlines the close relationship between these two services (Time Keeper and Scheduler).<br>• **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.<br>SMP does not mandate which of these modes a simulation environment has to support. |
| TK_MissionTime | Mission time.<br>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:<br>• Mission time is a value measured in nanoseconds, which is the lowest level of granularity supported for time in SMP.<br>• Mission time is returned as a signed 64-bit integer value, relative to a mission start date and time (which itself is not stored).<br>• Mission time can be queried using the GetMissionTime() method of the time keeper (via the ITimeKeeper-interface).<br>• 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 as well 01.01.2000, 12:00.<br>• 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.<br>• When storing a state vector, mission time is stored as well.<br>• When restoring a state vector, mission time is restored as well. |

| Name | Description |
|---|---|
| TK_EpochTime | Epoch time.<br>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 as well 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 off-set to simulation time, and hence progresses together with simulation time, except for the case when the offset is changed.<br>The following holds for epoch time:<br>• Epoch time is a value measured in nanoseconds, which is the lowest level of granularity supported for time in SMP.<br>• 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 2000+0.5). This allows specifying time values roughly between 1710 and 2290.<br>• Epoch time can be queried using the GetEpochTime() method of the time keeper (via the ITimeKeeperinter-face).<br>• 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.<br>• 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.<br>• When storing a state vector, epoch time (i.e. its offset to simulation time) is stored as well.<br>• When restoring a state vector, epoch time (i.e. its offset to simulation time) is restored as well. |

| Name | Description |
|---|---|
| TK_ZuluTime | Zulu time.<br>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):<br>"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.<br>Times are usually written in military time or 24 hour format such as 1830Z (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."<br>In SMP, 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:<br>• Zulu time is measured in nanoseconds.<br>• Zulu time is returned as a signed 64-bit integer value, relative to the epoch reference time.<br>• Zulu time represents the current time at Greenwich, England, called as well UTC or GMT.<br>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, for example a ground station or some Hardware-In-The-Loop (HITL), Zulu time is often used as a time stamp. |

## 4.3 Scheduler

The scheduler service calls entry points of models based on events triggered by one of the four time kinds.

### 4.3.1 IScheduler

This interface gives access to the Scheduler Service.

Components can register (Add) and unregister (Remove) entry points for scheduling. Further, they can set (Set) individual attributes of events on the scheduler.

**File**

#include "Smp/Services/IScheduler.h"

**Namespace**

Smp::Services

**Declaration of IScheduler**

```
        /// Unique Identifier of type IScheduler.
        extern const Uuid Uuid_IScheduler;

        /// This interface gives access to the Scheduler Service.
        /// Components can register (Add) and unregister (Remove) entry points
        /// for scheduling. Further, they can set (Set) individual attributes
        /// of events on the scheduler.
        class IScheduler :
            public virtual Smp::IService
        {
        public:

            /// Virtual destructor to release memory.
            virtual ~IScheduler() {}

            /// Add an immediate event to the scheduler.
            /// An immediate event is an event that will be added as a
            /// simulation time event (with simulation delta time of 0) at the
            /// front of the event queue. As such, it will be executed when the
            /// scheduler processes its simulation time events again, but not
            /// immediately in the call to AddImmediateEvent().
            /// When the simulator is in Standby state, simulation time does
            /// not progress, and simulation time events (including immediate
            /// events) are not processed.
            /// @remarks To execute an entry point immediately without going
            ///          through the scheduler, its Execute() method can be
            ///          called.
            /// @param   entryPoint Entry point to call from event.
            /// @return  Event identifier that can be used to change or remove
            ///          event.
            virtual Smp::Services::EventId AddImmediateEvent(const
Smp::IEntryPoint* entryPoint) = 0;

            /// Add event to scheduler that is called based on simulation time.
            /// An event with repeat=0 is not cyclic. It will be removed
            /// automatically after is has been triggered.
            /// An event with repeat>0 is cyclic, and will be repeated repeat
            /// times. Therefore, it will be called repeat+1 times, and then it
            /// will be removed automatically.
            /// An event with repeat=-1 is cyclic as well, but it will be
            /// triggered forever, unless it is removed from the scheduler
            /// using the RemoveEvent() method.
            /// For a cyclic event, the cycleTime needs to be positive. For
            /// non-cyclic events, it is ignored.
            /// The simulationTime must not be negative. Otherwise, the event
            /// will never be executed, but immediately removed.
            /// @param   entryPoint Entry point to call from event.
            /// @param   simulationTime Duration from now when to trigger the
            ///          event for the first time.
            /// @param   cycleTime Duration between two triggers of the event.
            /// @param   repeat Number of times the event shall be repeated, or
            ///          0 for a single event, or -1 for no limit.
            /// @return  Event identifier that can be used to change or remove
            ///          event.
```

```
        /// @throws  Smp::Services::IScheduler::InvalidCycleTime
        /// @throws  Smp::Services::IScheduler::InvalidEventTime
        virtual Smp::Services::EventId AddSimulationTimeEvent(
            const IEntryPoint* entryPoint,
            Smp::Duration simulationTime,
            Smp::Duration cycleTime = 0,
            Smp::Int64 repeat = 0) throw (
            Smp::Services::IScheduler::InvalidCycleTime,
            Smp::Services::IScheduler::InvalidEventTime) = 0;

        /// Add event to scheduler that is called based on mission time.
        /// An event with repeat=0 is not cyclic. It will be removed
        /// automatically after is has been triggered.
        /// An event with repeat>0 is cyclic, and will be repeated repeat
        /// times. Therefore, it will be called repeat+1 times, and then it
        /// will be removed automatically.
        /// An event with repeat=-1 is cyclic as well, but it will be
        /// triggered forever, unless it is removed from the scheduler
        /// using the RemoveEvent() method.
        /// For a cyclic event, the cycleTime needs to be positive. For
        /// non-cyclic events, it is ignored.
        /// The missionTime must not be before the current mission time of
        /// the ITimeKeeper service. Otherwise, the event will never be
        /// executed, but immediately removed.
        /// @param   entryPoint Entry point to call from event.
        /// @param   missionTime Absolute mission time when to trigger the
        ///          event for the first time.
        /// @param   cycleTime Duration between two triggers of the event.
        /// @param   repeat Number of times the event shall be repeated, or
        ///          0 for a single event, or -1 for no limit.
        /// @return  Event identifier that can be used to change or remove
        ///          event.
        /// @throws  Smp::Services::IScheduler::InvalidCycleTime
        /// @throws  Smp::Services::IScheduler::InvalidEventTime
        virtual Smp::Services::EventId AddMissionTimeEvent(
            const IEntryPoint* entryPoint,
            Smp::Duration missionTime,
            Smp::Duration cycleTime = 0,
            Smp::Int64 repeat = 0) throw (
            Smp::Services::IScheduler::InvalidCycleTime,
            Smp::Services::IScheduler::InvalidEventTime) = 0;

        /// Add event to scheduler that is called based on epoch time.
        /// An event with repeat=0 is not cyclic. It will be removed
        /// automatically after is has been triggered.
        /// An event with repeat>0 is cyclic, and will be repeated repeat
        /// times. Therefore, it will be called repeat+1 times, and then it
        /// will be removed automatically.
        /// An event with repeat=-1 is cyclic as well, but it will be
        /// triggered forever, unless it is removed from the scheduler
        /// using the RemoveEvent() method.
        /// For a cyclic event, the cycleTime needs to be positive. For
        /// non-cyclic events, it is ignored.
        /// The epochTime must not be before the current epoch time of the
        /// ITimeKeeper service. Otherwise, the event will never be
        /// executed, but immediately removed.
        /// @param   entryPoint Entry point to call from event.
        /// @param   epochTime Epoch time when to trigger the event for the
        ///          first time.
        /// @param   cycleTime Duration between two triggers of the event.
        /// @param   repeat Number of times the event shall be repeated, or
        ///          0 for a single event, or -1 for no limit.
        /// @return  Event identifier that can be used to change or remove
        ///          event.
        /// @throws  Smp::Services::IScheduler::InvalidCycleTime
        /// @throws  Smp::Services::IScheduler::InvalidEventTime
        virtual Smp::Services::EventId AddEpochTimeEvent(
            const IEntryPoint* entryPoint,
```

```
                Smp::DateTime epochTime,
                Smp::Duration cycleTime = 0,
                Smp::Int64 repeat = 0) throw (
                Smp::Services::IScheduler::InvalidCycleTime,
                Smp::Services::IScheduler::InvalidEventTime) = 0;

            /// Add event to scheduler that is called based on Zulu time.
            /// An event with repeat=0 is not cyclic. It will be removed
            /// automatically after is has been triggered.
            /// An event with repeat>0 is cyclic, and will be repeated repeat
            /// times. Therefore, it will be called repeat+1 times, and then it
            /// will be removed automatically.
            /// An event with repeat=-1 is cyclic as well, but it will be
            /// triggered forever, unless it is removed from the scheduler
            /// using the RemoveEvent() method.
            /// For a cyclic event, the cycleTime needs to be positive. For
            /// non-cyclic events, it is ignored.
            /// The zuluTime must not be before the current Zulu time of the
            /// ITimeKeeper service. Otherwise, the event will never be
            /// executed, but immediately removed.
            /// @param   entryPoint Entry point to call from event.
            /// @param   simulationTime Absolute (Zulu) time when to trigger
            ///          the event for the first time.
            /// @param   cycleTime Duration between two triggers of the event.
            /// @param   repeat Number of times the event shall be repeated, or
            ///          0 for a single event, or -1 for no limit.
            /// @return  Event identifier that can be used to change or remove
            ///          event.
            /// @throws  Smp::Services::IScheduler::InvalidCycleTime
            /// @throws  Smp::Services::IScheduler::InvalidEventTime
            virtual Smp::Services::EventId AddZuluTimeEvent(
                const IEntryPoint* entryPoint,
                Smp::DateTime simulationTime,
                Smp::Duration cycleTime = 0,
                Smp::Int64 repeat = 0) throw (
                Smp::Services::IScheduler::InvalidCycleTime,
                Smp::Services::IScheduler::InvalidEventTime) = 0;

            /// Update when an existing simulation time event on the scheduler
            /// shall be triggered.
            /// When the given event Id is not a valid identifier of a
            /// scheduler event, the method throws an exception of type
            /// InvalidEventId. In case an event is registered under the given
            /// event Id but it is not an simulation time event, the method
            /// throws an exception of type InvalidEventId as well.
            ///
            /// The simulationTime must not be negative. Otherwise, the event
            /// will never be executed, but immediately removed.
            /// @param   event Identifier of event to modify.
            /// @param   simulationTime Duration from now when to trigger event.
            /// @throws  Smp::Services::InvalidEventId
            virtual void SetEventSimulationTime(Smp::Services::EventId event,
Smp::Duration simulationTime) throw (
                Smp::Services::InvalidEventId) = 0;

            /// Update when an existing mission time event on the scheduler
            /// shall be triggered.
            /// When the given event Id is not a valid identifier of a
            /// scheduler event, the method throws an exception of type
            /// InvalidEventId. In case an event is registered under the given
            /// event Id but it is not an mission time event, the method throws
            /// an exception of type InvalidEventId as well.
            ///
            /// The missionTime must not be before the current mission time of
            /// the ITimeKeeper service. Otherwise, the event will never be
            /// executed, but immediately removed.
            /// @param   event Identifier of event to modify.
            /// @param   missionTime Absolute mission time when to trigger event.
```

```
            /// @throws  Smp::Services::InvalidEventId
            virtual void SetEventMissionTime(Smp::Services::EventId event,
Smp::Duration missionTime) throw (
                Smp::Services::InvalidEventId) = 0;

            /// Update when an existing epoch time event on the scheduler (an
            /// event that has been registered using AddEpochTimeEvent()) shall
            /// be triggered.
            /// When the given event Id  is not a valid identifier of a
            /// scheduler event, the method throws an exception of type
            /// InvalidEventId. In case an event is registered under the given
            /// event Id but it is not an epoch time event, the method throws
            /// an exception of type InvalidEventId as well.
            ///
            /// The epochTime must not be before the current epoch time of the
            /// ITimeKeeper service. Otherwise, the event will never be
            /// executed, but immediately removed.
            /// @param   event Identifier of event to modify.
            /// @param   epochTime Epoch time when to trigger event.
            /// @throws  Smp::Services::InvalidEventId
            virtual void SetEventEpochTime(Smp::Services::EventId event,
Smp::DateTime epochTime) throw (
                Smp::Services::InvalidEventId) = 0;

            /// Update when an existing zulu time event on the scheduler shall
            /// be triggered.
            /// When the given event Id is not a valid identifier of a
            /// scheduler event, the method throws an exception of type
            /// InvalidEventId. In case an event is registered under the given
            /// event Id but it is not an zulu time event, the method throws an
            /// exception of type InvalidEventId as well.
            /// The zuluTime must not be before the current Zulu time of the
            /// ITimeKeeper service. Otherwise, the event will never be
            /// executed, but immediately removed.
            /// @param   event Identifier of event to modify.
            /// @param   zuluTime Absolute (Zulu) time when to trigger event.
            /// @throws  Smp::Services::InvalidEventId
            virtual void SetEventZuluTime(Smp::Services::EventId event,
Smp::DateTime zuluTime) throw (
                Smp::Services::InvalidEventId) = 0;

            /// Update cycle time of an existing event on the scheduler.
            /// When the given event is not a valid identifier of a scheduler
            /// event, the method throws an exception of type InvalidEventId.
            /// For a cyclic event, the cycleTime needs to be positive. For
            /// non-cyclic events, it is ignored.
            /// @param   event Identifier of event to modify.
            /// @param   cycleTime Duration between two triggers of the event.
            /// @throws  Smp::Services::InvalidEventId
            virtual void SetEventCycleTime(Smp::Services::EventId event,
Smp::Duration cycleTime) throw (
                Smp::Services::InvalidEventId) = 0;

            /// Update the count of an existing event on the scheduler.
            /// When the given event is not a valid identifier of a scheduler
            /// event, the method throws an exception of type InvalidEventId.
            /// An event with count=0 is not cyclic. It will be removed
            /// automatically after is has been triggered.
            /// An event with count>0 is cyclic, and will be repeated count
            /// times. Therefore, it will be called count+1times, and then it
            /// will be removed automatically.
            /// An event with count=-1 is cyclic as well, but it will be
            /// triggered forever, unless it is removed from the scheduler
            /// using the RemoveEvent() method.
            /// For a cyclic event, the cycleTime needs to be positive. For
            /// non-cyclic events, it is ignored.
            /// @param   event Identifier of event to modify.
            /// @param   count Number of times the event shall be repeated, or
```

```
            ///          0 for a single event, or -1 for no limit.
            /// @throws  Smp::Services::InvalidEventId
            virtual void SetEventCount(Smp::Services::EventId event, Smp::Int64
count) throw (
                Smp::Services::InvalidEventId) = 0;

            /// Remove an event from the scheduler.
            /// When the given event is not a valid identifier of a scheduler
            /// event, the method throws an exception of type InvalidEventId.
            /// An event with count=0 is removed automatically after it has
            /// been triggered.
            /// @param   event Event identifier of the event to remove.
            /// @throws  Smp::Services::InvalidEventId
            virtual void RemoveEvent(Smp::Services::EventId event) throw (
                Smp::Services::InvalidEventId) = 0;
        };
```

**Base Interfaces**

Smp::IService

**Constants**

| Name | Type | Description | Value |
|---|---|---|---|
| SMP_Scheduler | String8 | Name of the Scheduler service. | Scheduler |

**Operations**

| Name | Description |
|---|---|
| AddEpochTimeEvent | Add event to scheduler that is called based on epoch time. |
| AddImmediateEvent | Add an immediate event to the scheduler. |
| AddMissionTimeEvent | Add event to scheduler that is called based on mission time. |
| AddSimulationTimeEvent | Add event to scheduler that is called based on simulation time. |
| AddZuluTimeEvent | Add event to scheduler that is called based on Zulu time. |
| RemoveEvent | Remove an event from the scheduler. |
| SetEventCount | Update the count of an existing event on the scheduler. |
| SetEventCycleTime | Update cycle time of an existing event on the scheduler. |
| SetEventEpochTime | Update when an existing epoch time event on the scheduler (an event that has been registered using AddEpochTimeEvent()) shall be triggered. |
| SetEventMissionTime | Update when an existing mission time event on the scheduler shall be triggered. |
| SetEventSimulationTime | Update when an existing simulation time event on the scheduler shall be triggered. |
| SetEventZuluTime | Update when an existing zulu time event on the scheduler shall be triggered. |

#### 4.3.1.1 Add Epoch Time Event

Add event to scheduler that is called based on epoch time.

An event with repeat=0 is not cyclic. It will be removed automatically after is has been triggered.

An event with repeat>0 is cyclic, and will be repeated repeat times. Therefore, it will be called repeat+1 times, and then it will be removed automatically.

An event with repeat=-1 is cyclic as well, but it will be triggered forever, unless it is removed from the scheduler using the RemoveEvent() method.

For a cyclic event, the cycleTime needs to be positive. For non-cyclic events, it is ignored.

The epochTime must not be before the current epoch time of the ITimeKeeper service. Otherwise, the event will never be executed, but immediately removed.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | EventId | Event identifier that can be used to change or remove event. |
| entryPoint | in | IEntryPoint | Entry point to call from event. |
| epochTime | in | DateTime | Epoch time when to trigger the event for the first time. |
| cycleTime | in | Duration | Duration between two triggers of the event. |
| repeat | in | Int64 | Number of times the event shall be repeated, or 0 for a single event, or -1 for no limit. |

**Exceptions**

Smp::Services::IScheduler::InvalidCycleTime,
Smp::Services::IScheduler::InvalidEventTime

#### 4.3.1.2 Add Immediate Event

Add an immediate event to the scheduler.

An immediate event is an event that will be added as a simulation time event (with simulation delta time of 0) at the front of the event queue. As such, it will be executed when the scheduler processes its simulation time events again, but not immediately in the call to AddImmediateEvent().

When the simulator is in Standby state, simulation time does not progress, and simulation time events (including immediate events) are not processed.

***Remark:*** To execute an entry point immediately without going through the scheduler, its Execute() method can be called.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| entryPoint | in | IEntryPoint | Entry point to call from event. |
| | return | EventId | Event identifier that can be used to change or remove event. |

**Exceptions**

None

#### 4.3.1.3 Add Mission Time Event

Add event to scheduler that is called based on mission time.

An event with repeat=0 is not cyclic. It will be removed automatically after is has been triggered.

An event with repeat>0 is cyclic, and will be repeated repeat times. Therefore, it will be called repeat+1 times, and then it will be removed automatically.

An event with repeat=-1 is cyclic as well, but it will be triggered forever, unless it is removed from the scheduler using the RemoveEvent() method.

For a cyclic event, the cycleTime needs to be positive. For non-cyclic events, it is ignored.

The missionTime must not be before the current mission time of the ITimeKeeper service. Otherwise, the event will never be executed, but immediately removed.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | EventId | Event identifier that can be used to change or remove event. |
| entryPoint | in | IEntryPoint | Entry point to call from event. |
| missionTime | in | Duration | Absolute mission time when to trigger the event for the first time. |
| cycleTime | in | Duration | Duration between two triggers of the event. |
| repeat | in | Int64 | Number of times the event shall be repeated, or 0 for a single event, or -1 for no limit. |

**Exceptions**

Smp::Services::IScheduler::InvalidCycleTime,
Smp::Services::IScheduler::InvalidEventTime

#### 4.3.1.4 Add Simulation Time Event

Add event to scheduler that is called based on simulation time.

An event with repeat=0 is not cyclic. It will be removed automatically after is has been triggered.

An event with repeat>0 is cyclic, and will be repeated repeat times. Therefore, it will be called repeat+1 times, and then it will be removed automatically.

An event with repeat=-1 is cyclic as well, but it will be triggered forever, unless it is removed from the scheduler using the RemoveEvent() method.

For a cyclic event, the cycleTime needs to be positive. For non-cyclic events, it is ignored.

The simulationTime must not be negative. Otherwise, the event will never be executed, but immediately removed.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | EventId | Event identifier that can be used to change or remove event. |
| entryPoint | in | IEntryPoint | Entry point to call from event. |
| simulationTime | in | Duration | Duration from now when to trigger the event for the first time. |
| cycleTime | in | Duration | Duration between two triggers of the event. |
| repeat | in | Int64 | Number of times the event shall be repeated, or 0 for a single event, or -1 for no limit. |

**Exceptions**

Smp::Services::IScheduler::InvalidCycleTime,
Smp::Services::IScheduler::InvalidEventTime

#### 4.3.1.5 Add Zulu Time Event

Add event to scheduler that is called based on Zulu time.

An event with repeat=0 is not cyclic. It will be removed automatically after is has been triggered.

An event with repeat>0 is cyclic, and will be repeated repeat times. Therefore, it will be called repeat+1 times, and then it will be removed automatically.

An event with repeat=-1 is cyclic as well, but it will be triggered forever, unless it is removed from the scheduler using the RemoveEvent() method.

For a cyclic event, the cycleTime needs to be positive. For non-cyclic events, it is ignored.

The zuluTime must not be before the current Zulu time of the ITimeKeeper service. Otherwise, the event will never be executed, but immediately removed.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | EventId | Event identifier that can be used to change or remove event. |
| entryPoint | in | IEntryPoint | Entry point to call from event. |
| simulationTime | in | DateTime | Absolute (Zulu) time when to trigger the event for the first time. |

| Name | Dir. | Type | Description |
|---|---|---|---|
| cycleTime | in | Duration | Duration between two triggers of the event. |
| repeat | in | Int64 | Number of times the event shall be repeated, or 0 for a single event, or -1 for no limit. |

**Exceptions**

Smp::Services::IScheduler::InvalidCycleTime,
Smp::Services::IScheduler::InvalidEventTime

#### 4.3.1.6 Remove Event

Remove an event from the scheduler.

When the given event is not a valid identifier of a scheduler event, the method throws an exception of type InvalidEventId.

An event with count=0 is removed automatically after it has been triggered.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| event | in | EventId | Event identifier of the event to remove. |

**Exceptions**

Smp::Services::InvalidEventId

#### 4.3.1.7 Set Event Count

Update the count of an existing event on the scheduler.

When the given event is not a valid identifier of a scheduler event, the method throws an exception of type InvalidEventId.

An event with count=0 is not cyclic. It will be removed automatically after is has been triggered.

An event with count>0 is cyclic, and will be repeated count times. Therefore, it will be called count+1times, and then it will be removed automatically.

An event with count=-1 is cyclic as well, but it will be triggered forever, unless it is removed from the scheduler using the RemoveEvent() method.

For a cyclic event, the cycleTime needs to be positive. For non-cyclic events, it is ignored.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| event | in | EventId | Identifier of event to modify. |
| count | in | Int64 | Number of times the event shall be repeated, or 0 for a single event, or -1 for no limit. |

**Exceptions**

Smp::Services::InvalidEventId

#### 4.3.1.8 Set Event Cycle Time

Update cycle time of an existing event on the scheduler.

When the given event is not a valid identifier of a scheduler event, the method throws an exception of type InvalidEventId.

For a cyclic event, the cycleTime needs to be positive. For non-cyclic events, it is ignored.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| event | in | EventId | Identifier of event to modify. |
| cycleTime | in | Duration | Duration between two triggers of the event. |

**Exceptions**

Smp::Services::InvalidEventId

#### 4.3.1.9 Set Event Epoch Time

Update when an existing epoch time event on the scheduler (an event that has been registered using AddEpochTimeEvent()) shall be triggered.

When the given event Id is not a valid identifier of a scheduler event, the method throws an exception of type InvalidEventId. In case an event is registered under the given event Id but it is not an epoch time event, the method throws an exception of type InvalidEventId as well.

The epochTime must not be before the current epoch time of the ITimeKeeper service. Otherwise, the event will never be executed, but immediately removed.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| event | in | EventId | Identifier of event to modify. |
| epochTime | in | DateTime | Epoch time when to trigger event. |

**Exceptions**

Smp::Services::InvalidEventId

#### 4.3.1.10 Set Event Mission Time

Update when an existing mission time event on the scheduler shall be triggered.

When the given event Id is not a valid identifier of a scheduler event, the method throws an exception of type InvalidEventId. In case an event is registered under the given event Id but it is not an mission time event, the method throws an exception of type InvalidEventId as well.

The missionTime must not be before the current mission time of the ITimeKeeper service. Otherwise, the event will never be executed, but immediately removed.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| event | in | EventId | Identifier of event to modify. |
| missionTime | in | Duration | Absolute mission time when to trigger event. |

**Exceptions**

Smp::Services::InvalidEventId

#### 4.3.1.11 Set Event Simulation Time

Update when an existing simulation time event on the scheduler shall be triggered.

When the given event Id is not a valid identifier of a scheduler event, the method throws an exception of type InvalidEventId. In case an event is registered under the given event Id but it is not an simulation time event, the method throws an exception of type InvalidEventId as well.

The simulationTime must not be negative. Otherwise, the event will never be executed, but immediately removed.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| event | in | EventId | Identifier of event to modify. |
| simulationTime | in | Duration | Duration from now when to trigger event. |

**Exceptions**

Smp::Services::InvalidEventId

#### 4.3.1.12 Set Event Zulu Time

Update when an existing zulu time event on the scheduler shall be triggered.

When the given event Id is not a valid identifier of a scheduler event, the method throws an exception of type InvalidEventId. In case an event is registered under the given event Id but it is not an zulu time event, the method throws an exception of type InvalidEventId as well.

The zuluTime must not be before the current Zulu time of the ITimeKeeper service. Otherwise, the event will never be executed, but immediately removed.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| event | in | EventId | Identifier of event to modify. |
| zuluTime | in | DateTime | Absolute (Zulu) time when to trigger event. |

**Exceptions**

Smp::Services::InvalidEventId

#### 4.3.1.13 Invalid Cycle Time

This exception is thrown by one of the AddEvent() methods of the scheduler when the event is a cyclic event (i.e. repeat is not 0), but the cycle time specified is not a positive duration.

**File**

#include "Smp/Services/IScheduler.h"

**Namespace**

Smp::Services::IScheduler

**Declaration of InvalidCycleTime**

```
/// This exception is thrown by one of the AddEvent() methods of
/// the scheduler when the event is a cyclic event (i.e. repeat is
/// not 0), but the cycle time specified is not a positive
/// duration.
class InvalidCycleTime : public Smp::Exception
{
public:
    /// Constructor for new exception.
    InvalidCycleTime() throw();

    /// Copy constructor.
    InvalidCycleTime(
        InvalidCycleTime& ex) throw();

    /// Virtual destructor to release memory.
    virtual ~InvalidCycleTime();

};
```

**Fields**

None

#### 4.3.1.14 Invalid Event Time

This exception is thrown by one of the AddEvent() methods of the scheduler when the time specified for the first execution of the event is in the past.

**File**

#include "Smp/Services/IScheduler.h"

**Namespace**

Smp::Services::IScheduler

**Declaration of InvalidEventTime**

```
/// This exception is thrown by one of the AddEvent() methods of
/// the scheduler when the time specified for the first execution
/// of the event is in the past.
class InvalidEventTime : public Smp::Exception
{
public:
    /// Constructor for new exception.
    InvalidEventTime() throw();

    /// Copy constructor.
    InvalidEventTime(
        InvalidEventTime& ex) throw();

    /// Virtual destructor to release memory.
```

```
            virtual ~InvalidEventTime();

        };
```

**Fields**

None

## 4.4 Event Manager

The event manager service provides a global notification mechanism. Components can register entry points with a global event. Several pre-defined event types exist, but applications can define their own, specific global events as well.

Although it is possible that any component triggers one of the pre-defined events by calling the Emit() method, models shall not emit pre-defined events, but only user-defined events.

To prevent accidentally emitting pre-defined events, these have been put into a "namespace", i.e. a prefix string "Smp_" has been added. It is recommended that user events include a "namespace" as well, for example "MyApp_MyEvent1".

### 4.4.1 IEvent Manager

This interface gives access to the Event Manager Service.

Components can register entry points with events, and they can define and emit events.

**File**

#include "Smp/Services/IEventManager.h"

**Namespace**

Smp::Services

**Declaration of IEventManager**

```
    /// Unique Identifier of type IEventManager.
    extern const Uuid Uuid_IEventManager;

    /// This interface gives access to the Event Manager Service.
    /// Components can register entry points with events, and they can
    /// define and emit events.
    class IEventManager :
        public virtual Smp::IService
    {
    public:

        /// Virtual destructor to release memory.
        virtual ~IEventManager() {}

        /// Get unique event identifier for an event name.
        /// It is guaranteed that this method will always return the same
        /// value when called with the same event name. This holds for
        /// pre-defined event names as well as for user-defined events.
```

```
            /// @param   eventName Name of the global event.
            /// @return  Event identifier for global event with given name.
            virtual Smp::Services::EventId QueryEventId(Smp::String8 eventName) =
0;

            /// Subscribe entry point to a global event.
            /// This method raises an exception of type InvalidEventId when
            /// called with an invalid event identifier. When the entry point
            /// is already subscribed to the same event, an exception of type
            /// AlreadySubscribed is raised.
            ///
            /// An entry point can only be subscribed once to an event.
            /// @param   event Event identifier of global event to subscribe to.
            /// @param   entryPoint Entry point to subscribe to global event.
            /// @throws  Smp::Services::IEventManager::AlreadySubscribed
            /// @throws  Smp::Services::InvalidEventId
            virtual void Subscribe(Smp::Services::EventId event, const
Smp::IEntryPoint* entryPoint) throw (
                Smp::Services::IEventManager::AlreadySubscribed,
                Smp::Services::InvalidEventId) = 0;

            /// Unsubscribe entry point from a global event.
            /// This method raises an exception of type InvalidEventId when
            /// called with an invalid event identifier. When the entry point
            /// is not subscribed to the event, an exception of type
            /// NotSubscribed is raised.
            /// An entry point can only be unsubscribed from an event when it
            /// has been subscribed earlier using Subscribe().
            /// @param   event Event identifier of global event to unsubscribe
            ///          from.
            /// @param   entryPoint Entry point to unsubscribe from global event.
            /// @throws  Smp::Services::InvalidEventId
            /// @throws  Smp::Services::IEventManager::NotSubscribed
            virtual void Unsubscribe(Smp::Services::EventId event, const
Smp::IEntryPoint* entryPoint) throw (
                Smp::Services::InvalidEventId,
                Smp::Services::IEventManager::NotSubscribed) = 0;

            /// Emit a global event.
            /// This will call all entry points that are subscribed to the
            /// global event with the given identifier at the time Emit() is
            /// called. Entry point subscription/unsubscription during the
            /// execution of Emit() is taken into account the next time Emit()
            /// is called. Entry points will be called in the order they have
            /// been subscribed to the global event.
            /// @param   event Event identifier of global event to emit.
            /// @param   synchronous Flag whether to emit the given event
            ///          synchronously (the default) or asynchronously.
            virtual void Emit(Smp::Services::EventId event, Smp::Bool synchronous =
true) = 0;
        };
```

**Base Interfaces**

Smp::IService

**Constants**

| Name | Type | Description | Value |
|---|---|---|---|
| SMP_EnterAborting | String8 | Enter Aborting state. | SMP_EnterAborting |
| SMP_EnterAbortingId | EventId | This event is raised when entering the Aborting state with the Abort() state transition command from any other state. | 13 |
| SMP_EnterExecuting | String8 | Enter Executing state. | SMP_EnterExecuting |
| SMP_EnterExecutingId | EventId | This event is raised when entering the Executing state with the Run() state transition command from Standby state. | 6 |
| SMP_EnterExiting | String8 | Enter Exiting state. | SMP_EnterExiting |
| SMP_EnterExitingId | EventId | This event is raised when entering the Exiting state with the Exit() state transition command from Standby state. | 12 |
| SMP_EnterInitialising | String8 | Enter Initialising state. | SMP_EnterInitialising |

| Name | Type | Description | Value |
|---|---|---|---|
| SMP_EnterInitialisingId | EventId | This event is raised when entering the Initialising state with an automatic state transition from Connecting state, or with the Initialise() state transition. | 2 |
| SMP_EnterReconnecting | String8 | Enter Reconnecting state. | SMP_EnterReconnecting |
| SMP_EnterReconnectingId | EventId | This event is raised when entering the Reconnecting state with the Reconnect() state transition from Standby state. | 16 |
| SMP_EnterRestoring | String8 | Enter Restoring state. | SMP_EnterRestoring |
| SMP_EnterRestoringId | EventId | This event is raised when entering the Restoring state with the Restore() state transition command from Standby state. | 10 |
| SMP_EnterStandby | String8 | Enter Standby state. | SMP_EnterStandby |

| Name | Type | Description | Value |
|---|---|---|---|
| SMP_EnterStandbyId | EventId | This event is raised when entering the Standby state with an automatic state transition from Initialising, Storing or Restoring state, or with the Hold() state transition command from Executing state. | 4 |
| SMP_EnterStoring | String8 | Enter Storing state. | SMP_EnterStoring |
| SMP_EnterStoringId | EventId | This event is raised when entering the Storing state with the Store() state transition command from Standby state. | 8 |
| SMP_EpochTimeChanged | String8 | Epoch Time has changed. | SMP_EpochTimeChanged |
| SMP_EpochTimeChangedId | EventId | This event is raised when changing the epoch time with the SetEpochTime() method of the time keeper service. | 14 |
| SMP_EventManager | String8 | Name of the EventManager service. | EventManager |
| SMP_LeaveConnecting | String8 | Leave Connecting state. | SMP_LeaveConnecting |

| Name | Type | Description | Value |
|---|---|---|---|
| SMP_LeaveConnectingId | EventId | This event is raised when leaving the Connecting state with an automatic state transition to Initializing state. | 1 |
| SMP_LeaveExecuting | String8 | Leave Executing state. | SMP_LeaveExecuting |
| SMP_LeaveExecutingId | EventId | This event is raised when leaving the Executing state with the Hold() state transition command to Standby state. | 7 |
| SMP_LeaveInitialising | String8 | Leave Initialising state. | SMP_LeaveInitialising |
| SMP_LeaveInitialisingId | EventId | This event is raised when leaving the Initialising state with an automatic state transition to Standby state. | 3 |
| SMP_LeaveReconnecting | String8 | Leave Reconnecting state. | SMP_LeaveReconnecting |
| SMP_LeaveReconnectingId | EventId | This event is raised when leaving the Reconnecting state with an automatic state transition to Standby state. | 17 |
| SMP_LeaveRestoring | String8 | Leave Restoring state. | SMP_LeaveRestoring |

| Name | Type | Description | Value |
|---|---|---|---|
| SMP_LeaveRestoringId | EventId | This event is raised when leaving the Restoring state with an automatic state transition to Standby state. | 11 |
| SMP_LeaveStandby | String8 | Leave Standby state. | SMP_LeaveStandby |
| SMP_LeaveStandbyId | EventId | This event is raised when leaving the Standby state with the Run() state transition command to Executing state, with the Store() state transition command to Storing state, with the Restore() state transition command to Restoring state, or with the Initialise() state transition command to Initialising state. | 5 |
| SMP_LeaveStoring | String8 | Leave Storing state. | SMP_LeaveStoring |
| SMP_LeaveStoringId | EventId | This event is raised when leaving the Storing state with an automatic state transition to Standby state. | 9 |
| SMP_MissionTimeChanged | String8 | Mission time has changed. | SMP_MissionTimeChanged |

| Name | Type | Description | Value |
|---|---|---|---|
| SMP_MissionTimeChangedId | EventId | This event is raised when changing the mission time with one of the SetMissionTime() and SetMissionStart() methods of the time keeper service. | 15 |

**Operations**

| Name | Description |
|---|---|
| Emit | Emit a global event. |
| QueryEventId | Get unique event identifier for an event name. |
| Subscribe | Subscribe entry point to a global event. |
| Unsubscribe | Unsubscribe entry point from a global event. |

#### 4.4.1.1 Emit

Emit a global event.

This will call all entry points that are subscribed to the global event with the given identifier at the time Emit() is called. Entry point subscription/unsubscription during the execution of Emit() is taken into account the next time Emit() is called. Entry points will be called in the order they have been subscribed to the global event.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| event | in | EventId | Event identifier of global event to emit. |
| synchronous | in | Bool | Flag whether to emit the given event synchronously (the default) or asynchronously. |

**Exceptions**

None

#### 4.4.1.2 Query Event Id

Get unique event identifier for an event name.

It is guaranteed that this method will always return the same value when called with the same event name. This holds for pre-defined event names as well as for user-defined events.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | EventId | Event identifier for global event with given name. |
| eventName | in | String8 | Name of the global event. |

**Exceptions**

None

#### 4.4.1.3 Subscribe

Subscribe entry point to a global event.

This method raises an exception of type InvalidEventId when called with an invalid event identifier. When the entry point is already subscribed to the same event, an exception of type AlreadySubscribed is raised.

An entry point can only be subscribed once to an event.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| event | in | EventId | Event identifier of global event to subscribe to. |
| entryPoint | in | IEntryPoint | Entry point to subscribe to global event. |

**Exceptions**

Smp::Services::InvalidEventId,
Smp::Services::IEventManager::AlreadySubscribed

#### 4.4.1.4 Unsubscribe

Unsubscribe entry point from a global event.

This method raises an exception of type InvalidEventId when called with an invalid event identifier. When the entry point is not subscribed to the event, an exception of type NotSubscribed is raised.

An entry point can only be unsubscribed from an event when it has been subscribed earlier using Subscribe().

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| event | in | EventId | Event identifier of global event to unsubscribe from. |
| entryPoint | in | IEntryPoint | Entry point to unsubscribe from global event. |

**Exceptions**

Smp::Services::InvalidEventId, Smp::Services::IEventManager::NotSubscribed

#### 4.4.1.5 Already Subscribed

This exception is raised when trying to subscribe an entry point to an event that is already subscribed.

**File**

#include "Smp/Services/IEventManager.h"

**Namespace**

Smp::Services::IEventManager

**Declaration of AlreadySubscribed**

```
/// This exception is raised when trying to subscribe an entry
/// point to an event that is already subscribed.
class AlreadySubscribed : public Smp::Exception
{
public:
    /// Constructor for new exception.
    /// @param   eventName Name of event the entry point is already
    ///          subscribed to.
    /// @param   entryPoint Entry point that is already subscribed.
    AlreadySubscribed(
        Smp::String8 eventName,
        const IEntryPoint* _entryPoint) throw() ;

    /// Copy constructor.
    AlreadySubscribed(
        AlreadySubscribed& ex) throw();

    /// Virtual destructor to release memory.
    virtual ~AlreadySubscribed();

    /// Name of event the entry point is already subscribed to.
    Smp::String8 eventName;

    /// Entry point that is already subscribed.
    Smp::IEntryPoint* entryPoint;
};
```

**Fields**

| Name | Type | Description |
|---|---|---|
| entryPoint | IEntryPoint | Entry point that is already subscribed. |
| eventName | String8 | Name of event the entry point is already subscribed to. |

#### 4.4.1.6 Not Subscribed

This exception is raised when trying to unsubscribe an entry point from an event that is not subscribed to it.

**File**

#include "Smp/Services/IEventManager.h"

**Namespace**

Smp::Services::IEventManager

**Declaration of NotSubscribed**

```
/// This exception is raised when trying to unsubscribe an entry
/// point from an event that is not subscribed to it.
class NotSubscribed : public Smp::Exception
{
public:
    /// Constructor for new exception.
```

```
                /// @param   eventName Name of event the entry point is not
                ///          subscribed to.
                /// @param   entryPoint Entry point that is not subscribed.
                NotSubscribed(
                    Smp::String8 eventName,
                    Smp::IEntryPoint* entryPoint) throw();

                /// Copy constructor.
                NotSubscribed(
                    NotSubscribed& ex) throw();

                /// Virtual destructor to release memory.
                virtual ~NotSubscribed();

                /// Name of event the entry point is not subscribed to.
                Smp::String8 eventName;

                /// Entry point that is not subscribed.
                const Smp::IEntryPoint* entryPoint;
            };
```

**Fields**

| Name | Type | Description |
|---|---|---|
| entryPoint | IEntryPoint | Entry point that is not subscribed. |
| eventName | String8 | Name of event the entry point is not subscribed to. |

### 4.4.2 Predefined Event Types


**Figure 5 - Predefined Event Types**

The Event Manager supports some global event names and ids defined for state changes of the simulation environment, or for a modified epoch or mission time. The state transition events clearly indicate in their names whether they are emitted when entering the corresponding state, or when leaving it.

The events indicating changes in either mission or epoch time are raised after the corresponding time has been changed, so that an immediate call to the time keeper service will return the new epoch or mission time, respectively.

The names and ids of the predefined event kinds are as listed in the table below. As most of these events relate to state changes, the state diagram of the simulation environment is shown in Figure 2 - Simulation Environment State Diagram with State Transition Methods on page 118.

| Event Name | Id | Description |
|---|---|---|
| LeaveConnecting | 1 | This event is raised when leaving the `Connecting` state with an automatic state transition to `Initializing` state. |
| EnterInitialising | 2 | This event is raised when entering the `Initialising` state with an automatic state transition from `Connecting` state, or with the `Initialise()` state transition. |
| LeaveInitialising | 3 | This event is raised when leaving the `Initialising` state with an automatic state transition to `Standby` state. |
| EnterStandby | 4 | This event is raised when entering the `Standby` state with an automatic state transition from `Initialising`, `Storing` or `Restoring` state, or with the `Hold()` state transition command from `Executing` state. |
| LeaveStandby | 5 | This event is raised when leaving the Standby state with the `Run()` state transition command to `Executing` state, with the `Store()` state transition command to `Storing` state, with the `Restore()` state transition command to `Restoring` state, or with the `Initialise()` state transition command to `Initialising` state. |
| EnterExecuting | 6 | This event is raised when entering the `Executing` state with the `Run()` state transition command from `Standby` state. |
| LeaveExecuting | 7 | This event is raised when leaving the `Executing` state with the `Hold()` state transition command to `Standby` state. |
| EnterStoring | 8 | This event is raised when entering the `Storing` state with the `Store()` state transition command from `Standby` state. |
| LeaveStoring | 9 | This event is raised when leaving the `Storing` state with an automatic state transition to `Standby` state. |
| EnterRestoring | 10 | This event is raised when entering the `Restoring` state with the `Restore()` state transition command from `Standby` state. |
| LeaveRestoring | 11 | This event is raised when leaving the `Restoring` state with an automatic state transition to `Standby` state. |
| EnterExiting | 12 | This event is raised when entering the `Exiting` state with the `Exit()` state transition command from `Standby` state. |
| EnterAborting | 13 | This event is raised when entering the `Aborting` state with the `Abort()` state transition command from any other state. |
| EpochTimeChanged | 14 | This event is raised when changing the epoch time with the `SetEpochTime()` method of the time keeper service. |
| MissionTimeChanged | 15 | This event is raised when changing the mission time with one of the `SetMissionTime()` and `SetMissionStart()` methods of the time keeper service. |
| EnterReconnecting | 16 | This event is raised when entering the Reconnecting state with the Reconnect() state transition from Standby state. |
| LeaveReconnecting | 17 | This event is raised when leaving the Reconnecting state with an automatic state transition to Standby state. |

User-defined event ids can be generated using the QueryEventId() method, which will return the same identifier every time it is called with the identical event name.

## 4.5 Resolver

Components can use the Resolver to resolve references to other components by name. References can either be specified using a fully qualified path, or using a path relative to some other component.

### 4.5.1 IResolver

This interface gives access to the Resolver Service.

**File**

#include "Smp/Services/IResolver.h"

**Namespace**

Smp::Services

**Declaration of IResolver**

```
        /// Unique Identifier of type IResolver.
        extern const Uuid Uuid_IResolver;

        /// This interface gives access to the Resolver Service.
        class IResolver :
            public virtual Smp::IService
        {
        public:

            /// Virtual destructor to release memory.
            virtual ~IResolver() {}

            /// Resolve reference to component via absolute path.
            ///
            /// An absolute path contains the name of either the Models or the
            /// Services container, but not the name of the simulator, although
            /// the simulator itself is the top-level component. This allows
            /// keeping names as short as possible, and avoids a dependency on
            /// the name of the simulator itself.
            /// @param   absolutePath Absolute path to component in simulation.
            /// @return  Component identified by path, or null if no component
            ///          with the given path could be found.
            virtual Smp::IComponent* ResolveAbsolute(Smp::String8 absolutePath) =
0;

            /// Resolve reference to component via relative path.
            /// @param   relativePath Relative path to component in simulation.
            /// @param   sender Component that asks for resolving the reference.
            /// @return  Component identified by path, or null if no component
            ///          with the given path could be found.
            virtual Smp::IComponent* ResolveRelative(Smp::String8 relativePath,
Smp::IComponent* sender) = 0;
        };
```

**Base Interfaces**

Smp::IService

**Constants**

| Name | Type | Description | Value |
|---|---|---|---|
| SMP_Resolver | String8 | Name of the Resolver service. | Resolver |

**Operations**

| Name | Description |
|---|---|
| ResolveAbsolute | Resolve reference to component via absolute path. |
| ResolveRelative | Resolve reference to component via relative path. |

#### 4.5.1.1 Resolve Absolute

Resolve reference to component via absolute path.

An absolute path contains the name of either the Models or the Services container, but not the name of the simulator, although the simulator itself is the top-level component. This allows keeping names as short as possible, and avoids a dependency on the name of the simulator itself.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | IComponent | Component identified by path, or null if no component with the given path could be found. |
| absolutePath | in | String8 | Absolute path to component in simulation. |

**Exceptions**

None

#### 4.5.1.2 Resolve Relative

Resolve reference to component via relative path.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | IComponent | Component identified by path, or null if no component with the given path could be found. |
| relativePath | in | String8 | Relative path to component in simulation. |
| sender | in | IComponent | Component that asks for resolving the reference. |

**Exceptions**

None

### 4.5.2 Component Paths

The Resolver can be used to resolve component references by name, either by a full path or by a relative path with respect to a given component. Therefore, it needs to be specified how path names have to be built. As all components in the tree of a simulation have a name, all that needs to be specified is how these names compose a path name, and how the parent component is identified.

**Rule 1**: Component names are assembled with one of the following characters: "\", "/", "!"

**Rule 2**: The parent component is specified by the following string: ".."

**Examples**:

```
Models/Satellite/Receivers/Receiver1
Services!Logger
..\..\Transmitters\Transmitter4
```

## 4.6 Link Registry

The Link Registry service maintains a list of all links between components, and can be used to find out whether a component can be safely removed from the simulation.

### 4.6.1 ILink Registry

This interface is implemented by the Link Registry Service.

The link registry maintains a global collection of links between components. Links can be added and removed, and can be queried for. Further, the link registry supports fetching and removing all links to a given target.

**File**

#include "Smp/Services/ILinkRegistry.h"

**Namespace**

Smp::Services

**Declaration of ILinkRegistry**

```
        /// Unique Identifier of type ILinkRegistry.
        extern const Uuid Uuid_ILinkRegistry;

        /// This interface is implemented by the Link Registry Service.
        /// The link registry maintains a global collection of links between
        /// components. Links can be added and removed, and can be queried for.
        /// Further, the link registry supports fetching and removing all links
        /// to a given target.
        class ILinkRegistry :
            public virtual Smp::IService
        {
        public:

            /// Virtual destructor to release memory.
            virtual ~ILinkRegistry() {}

            /// Add a link from source component to target component.
            /// This method informs the link registry that a link between two
            /// components has been created. The link registry does not create
            /// this link, it only gets told about its existence.
            /// This method can be called several times with the same
```

```
            /// arguments, when a source component has several links to the
            /// same target component.
            /// @param   source Source component of link (i.e. the component
            ///          that links to another component).
            /// @param   target Target component of link (i.e. the component
            ///          that is being linked to by another component).
            virtual void AddLink(Smp::IComponent* source, Smp::IComponent* target)
= 0;

            /// Returns true if a link between source and target exists, false
            /// otherwise.
            /// @param   source Source component of link  (i.e. the component
            ///          that links to another component).
            /// @param   target Target component of link  (i.e. the component
            ///          that is being linked to by another component).
            /// @return  True if such a link has been added before (and not
            ///          been removed), false otherwise.
            virtual Smp::Bool HasLink(Smp::IComponent* source, Smp::IComponent*
target) = 0;

            /// Remove a link between source and target that has been added to
            /// the service using AddLink() before.
            /// This method informs the link registry that a link between two
            /// components has been deleted. The link registry does not delete
            /// this link, it only gets told about the fact that the link no
            /// longer exists.
            /// This method can be called several times with the same
            /// arguments, when a source component had several links to the
            /// same target component.
            /// @param   source Source component of link (i.e. the component
            ///          that links to another component).
            /// @param   target Target component of link (i.e. the component
            ///          that is being linked to by another component).
            virtual void RemoveLink(Smp::IComponent* source, Smp::IComponent* tar-
get) = 0;

            /// Returns a collection of all sources that have a link to the
            /// given target.
            /// This method returns the collection of source components for
            /// which a link to the given target component has been added to
            /// the link registry.
            /// @param   target Target component to returns links for.
            /// @return  Collection of source components which link to the
            ///          given target.
            virtual const Smp::ComponentCollection* GetLinks(Smp::IComponent* tar-
get) const = 0;

            /// Returns true if all sources linking to the given target can be
            /// asked to remove their link(s), false otherwise.
            /// This method checks whether all sources that have a link to the
            /// given target implement the optional interface
            /// ILinkingComponent. If so, they can be asked to remove their
            /// links. The method returns false if at least one source exists
            /// that does not implement the ILinkingComponent interface.
            /// @param   target Target component to check for links.
            /// @return  True if all links to the given target can be removed,
            ///          false otherwise.
            virtual Smp::Bool CanRemove(Smp::IComponent* target) = 0;

            /// Removes all links to the given target.
            /// This method calls the RemoveLinks() method of all source
            /// components that implement the optional ILinkingComponent
            /// interface, so it asks all link sources to remove their links to
            /// the given target.
            /// @param   target Target component of link (i.e. the component
            ///          that is being linked to by other components).
            virtual void RemoveLinks(Smp::IComponent* target) = 0;
        };
```

**Base Interfaces**

Smp::IService

**Constants**

| Name | Type | Description | Value |
|---|---|---|---|
| SMP_LinkRegistry | String8 | Name of the LinkRegistry service. | LinkRegistry |

**Operations**

| Name | Description |
|---|---|
| AddLink | Add a link from source component to target component. |
| CanRemove | Returns true if all sources linking to the given target can be asked to remove their link(s), false otherwise. |
| GetLinks | Returns a collection of all sources that have a link to the given target. |
| HasLink | Returns true if a link between source and target exists, false otherwise. |
| RemoveLink | Remove a link between source and target that has been added to the service using AddLink() before. |
| RemoveLinks | Removes all links to the given target. |

#### 4.6.1.1 Add Link

Add a link from source component to target component.

This method informs the link registry that a link between two components has been created. The link registry does not create this link, it only gets told about its existence.

This method can be called several times with the same arguments, when a source component has several links to the same target component.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| source | in | IComponent | Source component of link (i.e. the component that links to another component). |
| target | in | IComponent | Target component of link (i.e. the component that is being linked to by another component). |

**Exceptions**

None

#### 4.6.1.2 Can Remove

Returns true if all sources linking to the given target can be asked to remove their link(s), false otherwise.

This method checks whether all sources that have a link to the given target implement the optional interface ILinkingComponent. If so, they can be asked to remove their links. The method returns false if at least one source exists that does not implement the ILinkingComponent interface.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | Bool | True if all links to the given target can be removed, false otherwise. |
| target | in | IComponent | Target component to check for links. |

**Exceptions**

None

#### 4.6.1.3 Get Links

Returns a collection of all sources that have a link to the given target.

This method returns the collection of source components for which a link to the given target component has been added to the link registry.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | ComponentCollection | Collection of source components which link to the given target. |
| target | in | IComponent | Target component to returns links for. |

**Exceptions**

None

#### 4.6.1.4 Has Link

Returns true if a link between source and target exists, false otherwise.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | Bool | True if such a link has been added before (and not been removed), false otherwise. |
| source | in | IComponent | Source component of link (i.e. the component that links to another component). |
| target | in | IComponent | Target component of link (i.e. the component that is being linked to by another component). |

**Exceptions**

None

#### 4.6.1.5 Remove Link

Remove a link between source and target that has been added to the service using AddLink() before.

This method informs the link registry that a link between two components has been deleted. The link registry does not delete this link, it only gets told about the fact that the link no longer exists.

This method can be called several times with the same arguments, when a source component had several links to the same target component.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| source | in | IComponent | Source component of link (i.e. the component that links to another component). |
| target | in | IComponent | Target component of link (i.e. the component that is being linked to by another component). |

**Exceptions**

None

#### 4.6.1.6 Remove Links

Removes all links to the given target.

This method calls the RemoveLinks() method of all source components that implement the optional ILinkingComponent interface, so it asks all link sources to remove their links to the given target.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| target | in | IComponent | Target component of link (i.e. the component that is being linked to by other components). |

**Exceptions**

None

# 5 Publication

SMP Models can publish their fields, operations and properties to a publication receiver by calling the operations declared in the IPublication interface that is provided by the simulation environment when calling the model's Publish() method. For every field, operation, property or parameter, a type needs to be specified during publication. As C++ does not have a type reflection mechanism, the C++ implementation of publication provides a type registry.

Therefore, the C++ implementation of publication is split into two major parts:

1. It allows registering user-defined types using their unique type identification (UUID).
2. It allows publishing fields, operations and properties of models.

Fields are published to allow store and restore of their values (state), to show them at run-time (view), and to support dataflow based simulation (input/output). Operations and properties are published to show them at run-time (view), and to support their use in script files. As this is limited to value types, only these are published into the type registry. Consequently, only properties of value types, and operations that only use value types for their parameters and return values can be published using the IPublication interface.

## 5.1 Type Registry

The Simulation Environment has to provide a single Type Registry that provides the following operations:

- A set of `Add...()` operations to register user-defined types.
- The `GetType()` operation to query for already registered types.

Registration of types is defined as part of the Package (see Smdl Package in section 6.6), but can be done by other mechanisms as well. However, it is mandatory that a type has been registered before it can be used by a feature. Further, a type most only be registered once. Trying to register the same type a second time (under the same UUID) results in an exception.

A type is basically registered by its name, description and UUID. For complex types, additional information (e.g. enumeration literals, or fields of a structure) is added. As the UUID of a type must be unique, there can be only one type registered under a given UUID.

The type registry provides the following operations to add types to it:

`AddFloatType` Registers a user-defined `Float` taking the minimum, maximum, the inclusive flags and the unit name as additional publication attributes.

| | |
|---|---|
| `AddIntegerType` | Registers a user-defined `Integer` taking the minimum and maximum values as additional publication attributes. |
| `AddEnumerationType` | Registers a user-defined `Enumeration`, taking the size of the memory enumeration as additional publication attribute. The enumeration literals have to be added in subsequent calls to the returned `IEnumerationType` interface. |
| `AddArrayType` | Registers a user-defined `Array`, taking the item type, item size and the array size as additional publication attributes. |
| `AddStringType` | Registers a user-defined `String`, taking the string size as additional publication attribute. |
| `AddStructureType` | Registers a user-defined `Structure`. The fields of the structure have to be added in subsequent calls to the returned `IStructureType` interface. |
| `AddClassType` | Registers a user-defined `Class`, taking the type of a potential base class as additional publication attribute. The fields of the class have to be published in subsequent calls of the returned `IClassType` interface. |

Each of these operations returns an instance of `IType`, or a derived interface.

### 5.1.1 IType Registry

This interface defines a registration mechanism for user types.

**File**

#include "Smp/Publication/ITypeRegistry.h"

**Namespace**

Smp::Publication

**Declaration of ITypeRegistry**

```
        /// Unique Identifier of type ITypeRegistry.
        extern const Uuid Uuid_ITypeRegistry;

        /// This interface defines a registration mechanism for user types.
        class ITypeRegistry
        {
        public:

            /// Virtual destructor to release memory.
            virtual ~ITypeRegistry() {}

            /// Returns a type by its simple type kind.
            /// @remarks This method can be used to map primitive types to the
            ///          IType interface, to treat all types identically.
            /// @param   type Primitive type the type is requested for.
            /// @return  Interface to the requested type.
            virtual Smp::Publication::IType* GetType(Smp::PrimitiveTypeKind type)
const = 0;

            /// Returns a type by universally unique identifier.
            /// @remarks This method can be used to find out whether a specific
```

```
            ///          type has been registered before.
            /// @param   typeUuid Universally unique identifier for the
            ///          requested type.
            /// @return  Interface of the requested type, or null if no type
            ///          with the registered Uuid could be found.
            virtual Smp::Publication::IType* GetType(Smp::Uuid typeUuid) const = 0;

            /// Add a float type to the registry.
            /// IManagedModel and IDynamicInvocation support fields, parameters
            /// and operations of Float types via the ST_Float32 and ST_Float64
            /// primitive type, as a Float is mapped either to Float32 or
            /// Float64.
            /// @param   name Name of the registered type.
            /// @param   description Description of the registered type.
            /// @param   typeUuid Universally unique identifier of the
            ///          registered type.
            /// @param   minimum Minimum value for float.
            /// @param   maximum Maximum value for float.
            /// @param   minInclusive Flag whether the minimum value for float
            ///          is valid or not.
            /// @param   maxInclusive Flag whether the maximum value for float
            ///          is valid or not.
            /// @param   unit Unit of the type.
            /// @param   type Primitive type to use for Float type.
            /// @return  Interface to new type.
            /// @throws  Smp::Publication::ITypeRegistry::AlreadyRegistered
            virtual Smp::Publication::IType* AddFloatType(
                Smp::String8 name,
                Smp::String8 description,
                Smp::Uuid typeUuid,
                Smp::Float64 minimum,
                Smp::Float64 maximum,
                Smp::Bool minInclusive,
                Smp::Bool maxInclusive,
                Smp::String8 unit,
                Smp::PrimitiveTypeKind type = PTK_Float64) throw (
                Smp::Publication::ITypeRegistry::AlreadyRegistered) = 0;

            /// Add an integer type to the registry.
            /// IManagedModel and IDynamicInvocation support fields, parameters
            /// and operations of Integer types via the ST_Int primitive types,
            /// as an Integer is mapped to one of Int8 / Int16 / Int32 / Int64
            /// / UInt8 / UInt16 / UInt32.
            /// @param   name Name of the registered type.
            /// @param   description Description of the registered type.
            /// @param   typeUuid Universally unique identifier of the
            ///          registered type.
            /// @param   minimum Minimum value for integer.
            /// @param   maximum Maximum value for integer.
            /// @param   unit Unit of the type.
            /// @param   type Primitive type to use for Integer type.
            /// @return  Interface to new type.
            /// @throws  Smp::Publication::ITypeRegistry::AlreadyRegistered
            virtual Smp::Publication::IType* AddIntegerType(
                Smp::String8 name,
                Smp::String8 description,
                Smp::Uuid typeUuid,
                Smp::Int64 minimum,
                Smp::Int64 maximum,
                Smp::String8 unit,
                Smp::PrimitiveTypeKind type = PTK_Int32) throw (
                Smp::Publication::ITypeRegistry::AlreadyRegistered) = 0;

            /// Add an enumeration type to the registry.
            /// @param   name Name of the registered type.
            /// @param   description Description of the registered type.
            /// @param   typeUuid Universally unique identifier of the
            ///          registered type.
```

```
            /// @param   memorySize Size of an instance of this enumeration in
            ///          bytes. Valid values are 1, 2, 4, 8
            /// @return  Interface to new type.
            /// @throws  Smp::Publication::ITypeRegistry::AlreadyRegistered
            virtual Smp::Publication::IType* AddEnumerationType(
                Smp::String8 name,
                Smp::String8 description,
                Smp::Uuid typeUuid,
                Smp::Int16 memorySize) throw (
                Smp::Publication::ITypeRegistry::AlreadyRegistered) = 0;

            /// Add an array type to the registry.
            /// @param   name Name of the registered type.
            /// @param   description Description of the registered type.
            /// @param   typeUuid Universally unique identifier of the
            ///          registered type.
            /// @param   itemTypeUuid Universally unique identifier of the Type
            ///          of the array items.
            /// @param   itemSize Size of an array item in bytes.
            /// @param   arrayCount Number of elements in the array.
            /// @return  Interface to new type.
            /// @throws  Smp::Publication::ITypeRegistry::AlreadyRegistered
            virtual Smp::Publication::IType* AddArrayType(
                Smp::String8 name,
                Smp::String8 description,
                Smp::Uuid typeUuid,
                Smp::Uuid itemTypeUuid,
                Smp::Int64 itemSize,
                Smp::Int64 arrayCount) throw (
                Smp::Publication::ITypeRegistry::AlreadyRegistered) = 0;

            /// Add a string type to the registry.
            /// @param   name Name of the registered type.
            /// @param   description Description of the registered type.
            /// @param   typeUuid Universally unique identifier of the
            ///          registered type.
            /// @param   length Maximum length of the string.
            /// @return  Interface to new type.
            /// @throws  Smp::Publication::ITypeRegistry::AlreadyRegistered
            virtual Smp::Publication::IType* AddStringType(
                Smp::String8 name,
                Smp::String8 description,
                Smp::Uuid typeUuid,
                Smp::Int64 length) throw (
                Smp::Publication::ITypeRegistry::AlreadyRegistered) = 0;

            /// Add a structure type to the registry.
            /// @param   name Name of the registered type.
            /// @param   description Description of the registered type.
            /// @param   typeUuid Universally unique identifier of the
            ///          registered type.
            /// @return  Interface to new type that allows adding fields.
            /// @throws  Smp::Publication::ITypeRegistry::AlreadyRegistered
            virtual Smp::Publication::IStructureType* AddStructureType(
                Smp::String8 name,
                Smp::String8 description,
                Smp::Uuid typeUuid) throw (
                Smp::Publication::ITypeRegistry::AlreadyRegistered) = 0;

            /// Add a class type to the registry.
            /// @param   name Name of the registered type.
            /// @param   description Description of the registered type.
            /// @param   typeUuid Universally unique identifier of the
            ///          registered type.
            /// @param   bassClassUuid baseClassUuid  Universally unique
            ///          identifier of the base class.
            /// @return  Interface to new type that allows adding fields.
            /// @throws  Smp::Publication::ITypeRegistry::AlreadyRegistered
```

```
            virtual Smp::Publication::IClassType* AddClassType(
                Smp::String8 name,
                Smp::String8 description,
                Smp::Uuid typeUuid,
                Smp::Uuid bassClassUuid) throw (
                Smp::Publication::ITypeRegistry::AlreadyRegistered) = 0;
        };
```

**Base Interfaces**

None

**Operations**

| Name | Description |
|---|---|
| AddArrayType | Add an array type to the registry. |
| AddClassType | Add a class type to the registry. |
| AddEnumerationType | Add an enumeration type to the registry. |
| AddFloatType | Add a float type to the registry. |
| AddIntegerType | Add an integer type to the registry. |
| AddStringType | Add a string type to the registry. |
| AddStructureType | Add a structure type to the registry. |
| GetType | Returns a type by its simple type kind. |
| GetType | Returns a type by universally unique identifier. |

#### 5.1.1.1 Add Array Type

Add an array type to the registry.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Name of the registered type. |
| description | in | String8 | Description of the registered type. |
| typeUuid | in | Uuid | Universally unique identifier of the registered type. |
| itemTypeUuid | in | Uuid | Universally unique identifier of the Type of the array items. |
| itemSize | in | Int64 | Size of an array item in bytes. |
| arrayCount | in | Int64 | Number of elements in the array. |
| | return | IType | Interface to new type. |

**Exceptions**

Smp::Publication::ITypeRegistry::AlreadyRegistered

#### 5.1.1.2 Add Class Type

Add a class type to the registry.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Name of the registered type. |
| description | in | String8 | Description of the registered type. |
| typeUuid | in | Uuid | Universally unique identifier of the registered type. |
| bassClassUuid | in | Uuid | baseClassUuid Universally unique identifier of the base class. |
| | return | IClassType | Interface to new type that allows adding fields. |

**Exceptions**

Smp::Publication::ITypeRegistry::AlreadyRegistered

#### 5.1.1.3 Add Enumeration Type

Add an enumeration type to the registry.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Name of the registered type. |
| description | in | String8 | Description of the registered type. |
| typeUuid | in | Uuid | Universally unique identifier of the registered type. |
| memorySize | in | Int16 | Size of an instance of this enumeration in bytes. Valid values are 1, 2, 4, 8 |
| | return | IType | Interface to new type. |

**Exceptions**

Smp::Publication::ITypeRegistry::AlreadyRegistered

#### 5.1.1.4 Add Float Type

Add a float type to the registry.

IManagedModel and IDynamicInvocation support fields, parameters and operations of Float types via the ST_Float32 and ST_Float64 primitive type, as a Float is mapped either to Float32 or Float64.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Name of the registered type. |
| description | in | String8 | Description of the registered type. |
| typeUuid | in | Uuid | Universally unique identifier of the registered type. |
| minimum | in | Float64 | Minimum value for float. |

| Name | Dir. | Type | Description |
|---|---|---|---|
| maximum | in | Float64 | Maximum value for float. |
| minInclusive | in | Bool | Flag whether the minimum value for float is valid or not. |
| maxInclusive | in | Bool | Flag whether the maximum value for float is valid or not. |
| unit | in | String8 | Unit of the type. |
| type | in | PrimitiveTypeKind | Primitive type to use for Float type. |
| | return | IType | Interface to new type. |

**Exceptions**

Smp::Publication::ITypeRegistry::AlreadyRegistered

#### 5.1.1.5 Add Integer Type

Add an integer type to the registry.

IManagedModel and IDynamicInvocation support fields, parameters and operations of Integer types via the ST_Int primitive types, as an Integer is mapped to one of Int8 / Int16 / Int32 / Int64 / UInt8 / UInt16 / UInt32.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Name of the registered type. |
| description | in | String8 | Description of the registered type. |
| typeUuid | in | Uuid | Universally unique identifier of the registered type. |
| minimum | in | Int64 | Minimum value for integer. |
| maximum | in | Int64 | Maximum value for integer. |
| unit | in | String8 | Unit of the type. |
| type | in | PrimitiveTypeKind | Primitive type to use for Integer type. |
| | return | IType | Interface to new type. |

**Exceptions**

Smp::Publication::ITypeRegistry::AlreadyRegistered

#### 5.1.1.6 Add String Type

Add a string type to the registry.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Name of the registered type. |
| description | in | String8 | Description of the registered type. |
| typeUuid | in | Uuid | Universally unique identifier of the registered type. |
| length | in | Int64 | Maximum length of the string. |
| | return | IType | Interface to new type. |

**Exceptions**

Smp::Publication::ITypeRegistry::AlreadyRegistered

#### 5.1.1.7 Add Structure Type

Add a structure type to the registry.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Name of the registered type. |
| description | in | String8 | Description of the registered type. |
| typeUuid | in | Uuid | Universally unique identifier of the registered type. |
| | return | IStructureType | Interface to new type that allows adding fields. |

**Exceptions**

Smp::Publication::ITypeRegistry::AlreadyRegistered

#### 5.1.1.8 Get Type

Returns a type by its simple type kind.

***Remark:*** This method can be used to map primitive types to the IType interface, to treat all types identically.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | IType | Interface to the requested type. |
| type | in | PrimitiveTypeKind | Primitive type the type is requested for. |

**Exceptions**

None

#### 5.1.1.9 Get Type

Returns a type by universally unique identifier.

***Remark:*** This method can be used to find out whether a specific type has been registered before.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | IType | Interface of the requested type, or null if no type with the registered Uuid could be found. |
| typeUuid | in | Uuid | Universally unique identifier for the requested type. |

**Exceptions**

None

#### 5.1.1.10 Already Registered

This exception is raised when trying to register a type with a Uuid that has already been registered.

**File**

#include "Smp/Publication/ITypeRegistry.h"

**Namespace**

Smp::Publication::ITypeRegistry

**Declaration of AlreadyRegistered**

```
/// This exception is raised when trying to register a type with a
/// Uuid that has already been registered.
class AlreadyRegistered : public Smp::Exception
{
public:
    /// Constructor for new exception.
    /// @param   name Name of new type that cannot be registered.
    /// @param   type Type that uses the same Uuid already
    AlreadyRegistered(
        Smp::String8 name,
        Smp::Publication::IType* type) throw();

    /// Copy constructor.
    AlreadyRegistered(
        AlreadyRegistered& ex) throw();

    /// Virtual destructor to release memory.
    virtual ~AlreadyRegistered();

    /// Name of new type that cannot be registered.
    Smp::String8 name;

    /// Type that uses the same Uuid already
    Smp::Publication::IType* type;
};
```

**Fields**

| Name | Type | Description |
|---|---|---|
| name | String8 | Name of new type that cannot be registered. |
| type | IType | Type that uses the same Uuid already |

### 5.1.2 IType

This base interface defines a type in the type registry.

**File**

#include "Smp/Publication/IType.h"

**Namespace**

Smp::Publication

**Declaration of IType**

```
/// Unique Identifier of type IType.
extern const Uuid Uuid_IType;

/// This base interface defines a type in the type registry.
class IType :
    public virtual Smp::IObject
{
public:

    /// Virtual destructor to release memory.
    virtual ~IType() {}

    /// Get primitive type that this type maps to, or PTK_None when the
    /// type cannot be mapped to a primitive type.
    /// @return  Primitive type kind that this type can be mapped to.
    virtual Smp::PrimitiveTypeKind GetPrimitiveType() const = 0;

    /// Get Universally Unique Identifier of type.
    /// @return  Universally Unique Identifier of type.
    virtual Smp::Uuid GetUuid() const = 0;

    /// Publish an instance of the type against a receiver.
    /// @param   receiver Receiver to publish against.
    /// @param   name Name of instance.
    /// @param   description Description of instance.
    /// @param   address Address of instance.
    /// @param   view View kind of instance.
    /// @param   state State flag of instance.
    /// @param   input Input flag of instance.
    /// @param   output Output flag of instance.
    virtual void Publish(
        Smp::IPublication* receiver,
        Smp::String8 name,
        Smp::String8 description,
        void* address,
        Smp::ViewKind view = VK_None,
        Smp::Bool state = true,
        Smp::Bool input = false,
        Smp::Bool output = false) = 0;
};
```

**Base Interfaces**

Smp::IObject

**Operations**

| Name | Description |
|---|---|
| GetPrimitiveType | Get primitive type that this type maps to, or PTK_None when the type cannot be mapped to a primitive type. |
| GetUuid | Get Universally Unique Identifier of type. |
| Publish | Publish an instance of the type against a receiver. |

#### 5.1.2.1 Get Primitive Type

Get primitive type that this type maps to, or PTK_None when the type cannot be mapped to a primitive type.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | PrimitiveTypeKind | Primitive type kind that this type can be mapped to. |

**Exceptions**

None

#### 5.1.2.2 Get Uuid

Get Universally Unique Identifier of type.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | Uuid | Universally Unique Identifier of type. |

**Exceptions**

None

#### 5.1.2.3 Publish

Publish an instance of the type against a receiver.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| receiver | in | IPublication | Receiver to publish against. |
| name | in | String8 | Name of instance. |
| description | in | String8 | Description of instance. |
| address | in | void* | Address of instance. |
| view | in | ViewKind | View kind of instance. |
| state | in | Bool | State flag of instance. |
| input | in | Bool | Input flag of instance. |
| output | in | Bool | Output flag of instance. |

**Exceptions**

None

### 5.1.3 IEnumeration Type

This interface defines a user defined enumeration.

**File**

#include "Smp/Publication/IEnumerationType.h"

**Namespace**

Smp::Publication

**Declaration of IEnumerationType**

```
/// Unique Identifier of type IEnumerationType.
extern const Uuid Uuid_IEnumerationType;

/// This interface defines a user defined enumeration.
class IEnumerationType :
    public virtual Smp::Publication::IType
{
public:

    /// Virtual destructor to release memory.
    virtual ~IEnumerationType() {}

    /// Add a literal to the Enumeration.
    /// @param   name Name of the literal.
    /// @param   description Description of the literal.
    /// @param   value Value of the literal
    virtual void AddLiteral(
        Smp::String8 name,
        Smp::String8 description,
        Smp::Int32 value) = 0;
};
```

**Base Interfaces**

Smp::Publication::IType

**Operations**

| Name | Description |
|---|---|
| AddLiteral | Add a literal to the Enumeration. |

#### 5.1.3.1 Add Literal

Add a literal to the Enumeration.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Name of the literal. |
| description | in | String8 | Description of the literal. |
| value | in | Int32 | Value of the literal |

**Exceptions**

None

### 5.1.4 IStructure Type

This interface defines a user defined structure.

**File**

#include "Smp/Publication/IStructureType.h"

**Namespace**

Smp::Publication

**Declaration of IStructureType**

```
/// Unique Identifier of type IStructureType.
extern const Uuid Uuid_IStructureType;

/// This interface defines a user defined structure.
class IStructureType :
    public virtual Smp::Publication::IType
{
public:

    /// Virtual destructor to release memory.
    virtual ~IStructureType() {}

    /// Add a field to the Structure.
    /// @param   name Name of field.
    /// @param   description Description of field.
    /// @param   uuid Uuid of field Type, which must be a value type,
    ///          but not String8.
    /// @param   offset Memory offset of field relative to Structure.
    /// @param   view View kind of field.
    /// @param   state State flag of field.
    /// @param   input Input flag of field.
    /// @param   output Output flag of field.
    virtual void AddField(
        Smp::String8 name,
        Smp::String8 description,
        Smp::Uuid uuid,
        Smp::Int64 offset,
        Smp::ViewKind view = VK_All,
        Smp::Bool state = true,
        Smp::Bool input = false,
        Smp::Bool output = false) = 0;
};
```

**Base Interfaces**

Smp::Publication::IType

**Operations**

| Name | Description |
|---|---|
| AddField | Add a field to the Structure. |

#### 5.1.4.1 Add Field

Add a field to the Structure.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Name of field. |
| description | in | String8 | Description of field. |
| uuid | in | Uuid | Uuid of field Type, which must be a value type, but not String8. |
| offset | in | Int64 | Memory offset of field relative to Structure. |
| view | in | ViewKind | View kind of field. |
| state | in | Bool | State flag of field. |
| input | in | Bool | Input flag of field. |
| output | in | Bool | Output flag of field. |

**Exceptions**

None

### 5.1.5 IClass Type

This interface defines a user defined class.

**File**

#include "Smp/Publication/IClassType.h"

**Namespace**

Smp::Publication

**Declaration of IClassType**

```
/// Unique Identifier of type IClassType.
extern const Uuid Uuid_IClassType;

/// This interface defines a user defined class.
class IClassType :
    public virtual Smp::Publication::IStructureType
{
};
```

**Base Interfaces**

Smp::Publication::IStructureType

**Operations**

None

### 5.1.6 Not Registered

This exception is raised when trying to publish a feature with a type Uuid that has not been registered.

**File**

#include "Smp/Publication/ITypeRegistry.h"

**Namespace**

Smp::Publication

**Declaration of NotRegistered**

```
/// This exception is raised when trying to publish a feature with a
/// type Uuid that has not been registered.
class NotRegistered : public Smp::Exception
{
public:
    /// Constructor for new exception.
    /// @param   uuid Uuid that does not correspond to a registered type.
    NotRegistered(
        Smp::Uuid uuid) throw();

    /// Copy constructor.
    NotRegistered(
        NotRegistered& ex) throw();

    /// Virtual destructor to release memory.
```

```
            virtual ~NotRegistered();

            /// Uuid that does not correspond to a registered type.
            Smp::Uuid uuid;
        };
```

**Base Exceptions**

Smp::Exception

**Fields**

| Name | Type | Description |
|---|---|---|
| uuid | Uuid | Uuid that does not correspond to a registered type. |

## 5.2 Publication of Fields, Operations and Properties

### 5.2.1 IPublish Operation

This interface provides functionality to publish operation parameters.

**File**

#include "Smp/Publication/IPublishOperation.h"

**Namespace**

Smp::Publication

**Declaration of IPublishOperation**

```
        /// Unique Identifier of type IPublishOperation.
        extern const Uuid Uuid_IPublishOperation;

        /// This interface provides functionality to publish operation
        /// parameters.
        class IPublishOperation
        {
        public:

            /// Virtual destructor to release memory.
            virtual ~IPublishOperation() {}

            /// Publish a parameter of an operation.
            /// This method works for all types.
            /// @remarks Fields, parameters and operations of Enumeration types
            ///          are supported by IManagedModel and IDynamicInvocation
            ///          via one of the ST_Int8, ST_Int16, ST_Int32 or ST_Int64
            ///          primitive types, depending on their memory size.
            /// @param   name Parameter name.
            /// @param   description Parameter description.
            /// @param   typeUuid Uuid of parameter type.
            /// @param   direction Direction kind of parameter.
            /// @throws  Smp::Publication::NotRegistered
            virtual void PublishParameter(
                Smp::String8 name,
                Smp::String8 description,
```

```
                Smp::Uuid typeUuid,
                Smp::Publication::ParameterDirectionKind direction) throw (
                Smp::Publication::NotRegistered) = 0;
        };
```

**Base Interfaces**

None

**Operations**

| Name | Description |
|---|---|
| PublishParameter | Publish a parameter of an operation. |

#### 5.2.1.1 Publish Parameter

Publish a parameter of an operation.

This method works for all types.

***Remark:*** Fields, parameters and operations of Enumeration types are supported by IManagedModel and IDynamicInvocation via one of the ST_Int8, ST_Int16, ST_Int32 or ST_Int64 primitive types, depending on their memory size.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Parameter name. |
| description | in | String8 | Parameter description. |
| typeUuid | in | Uuid | Uuid of parameter type. |
| direction | in | ParameterDirectionKind | Direction kind of parameter. |

**Exceptions**

Smp::Publication::NotRegistered

### 5.2.2 IPublication

Interface that provides functionality to allow publishing members, including fields, properties and operations.

This interface is platform specific. For details see the SMP Platform Mappings.

**File**

#include "Smp/IPublication.h"

**Namespace**

Smp

**Declaration of IPublication**

```
    /// Unique Identifier of type IPublication.
    extern const Uuid Uuid_IPublication;

    /// Interface that provides functionality to allow publishing members,
    /// including fields, properties and operations.
    /// This interface is platform specific. For details see the SMP Platform
    /// Mappings.
```

```
class IPublication
{
public:

    /// Virtual destructor to release memory.
    virtual ~IPublication() {}

    /// Give access to the global type registry.
    /// The type registry is typically a singleton, and must not be null,
    /// to allow use of existing types, and registration of new types.
    /// @return  Reference to global type registry.
    virtual Smp::Publication::ITypeRegistry* GetTypeRegistry() const = 0;

    /// Publish a Char8 field.
    /// @param   name Field name.
    /// @param   description Field description.
    /// @param   address Field memory address.
    /// @param   view Show field in model tree.
    /// @param   state Include field in store/restore of simulation state.
    /// @param   input True if field is an input field, false otherwise.
    /// @param   output True if field is an output field, false otherwise.
    virtual void PublishField(
        Smp::String8 name,
        Smp::String8 description,
        Smp::Char8* address,
        Smp::ViewKind view = VK_None,
        Smp::Bool state = true,
        Smp::Bool input = false,
        Smp::Bool output = false) = 0;

    /// Publish a Bool field.
    /// @param   name Field name.
    /// @param   description Field description.
    /// @param   address Field memory address.
    /// @param   view Show field in model tree.
    /// @param   state Include field in store/restore of simulation state.
    /// @param   input True if field is an input field, false otherwise.
    /// @param   output True if field is an output field, false otherwise.
    virtual void PublishField(
        Smp::String8 name,
        Smp::String8 description,
        Smp::Bool* address,
        Smp::ViewKind view = VK_None,
        Smp::Bool state = true,
        Smp::Bool input = false,
        Smp::Bool output = false) = 0;

    /// Publish a Int8 field.
    /// @param   name Field name.
    /// @param   description Field description.
    /// @param   address Field memory address.
    /// @param   view Show field in model tree.
    /// @param   state Include field in store/restore of simulation state.
    /// @param   input True if field is an input field, false otherwise.
    /// @param   output True if field is an output field, false otherwise.
    virtual void PublishField(
        Smp::String8 name,
        Smp::String8 description,
        Smp::Int8* address,
        Smp::ViewKind view = VK_None,
        Smp::Bool state = true,
        Smp::Bool input = false,
        Smp::Bool output = false) = 0;

    /// Publish a Int16 field.
    /// @param   name Field name.
    /// @param   description Field description.
    /// @param   address Field memory address.
```

```
        /// @param   view Show field in model tree.
        /// @param   state Include field in store/restore of simulation state.
        /// @param   input True if field is an input field, false otherwise.
        /// @param   output True if field is an output field, false otherwise.
        virtual void PublishField(
            Smp::String8 name,
            Smp::String8 description,
            Smp::Int16* address,
            Smp::ViewKind view = VK_None,
            Smp::Bool state = true,
            Smp::Bool input = false,
            Smp::Bool output = false) = 0;

        /// Publish a Int32 field.
        /// @param   name Field name.
        /// @param   description Field description.
        /// @param   address Field memory address.
        /// @param   view Show field in model tree.
        /// @param   state Include field in store/restore of simulation state.
        /// @param   input True if field is an input field, false otherwise.
        /// @param   output True if field is an output field, false otherwise.
        virtual void PublishField(
            Smp::String8 name,
            Smp::String8 description,
            Smp::Int32* address,
            Smp::ViewKind view = VK_None,
            Smp::Bool state = true,
            Smp::Bool input = false,
            Smp::Bool output = false) = 0;

        /// Publish a UInt8 field.
        /// @param   name Field name.
        /// @param   description Field description.
        /// @param   address Field memory address.
        /// @param   view Show field in model tree.
        /// @param   state Include field in store/restore of simulation state.
        /// @param   input True if field is an input field, false otherwise.
        /// @param   output True if field is an output field, false otherwise.
        virtual void PublishField(
            Smp::String8 name,
            Smp::String8 description,
            Smp::UInt8* address,
            Smp::ViewKind view = VK_None,
            Smp::Bool state = true,
            Smp::Bool input = false,
            Smp::Bool output = false) = 0;

        /// Publish a UInt16 field.
        /// @param   name Field name.
        /// @param   description Field description.
        /// @param   address Field memory address.
        /// @param   view Show field in model tree.
        /// @param   state Include field in store/restore of simulation state.
        /// @param   input True if field is an input field, false otherwise.
        /// @param   output True if field is an output field, false otherwise.
        virtual void PublishField(
            Smp::String8 name,
            Smp::String8 description,
            Smp::UInt16* address,
            Smp::ViewKind view = VK_None,
            Smp::Bool state = true,
            Smp::Bool input = false,
            Smp::Bool output = false) = 0;

        /// Publish a UInt32 field.
        /// @param   name Field name.
        /// @param   description Field description.
        /// @param   address Field memory address.
```

```
        /// @param   view Show field in model tree.
        /// @param   state Include field in store/restore of simulation state.
        /// @param   input True if field is an input field, false otherwise.
        /// @param   output True if field is an output field, false otherwise.
        virtual void PublishField(
            Smp::String8 name,
            Smp::String8 description,
            Smp::UInt32* address,
            Smp::ViewKind view = VK_None,
            Smp::Bool state = true,
            Smp::Bool input = false,
            Smp::Bool output = false) = 0;

        /// Publish a UInt64 field.
        /// @param   name Field name.
        /// @param   description Field description.
        /// @param   address Field memory address.
        /// @param   view Show field in model tree.
        /// @param   state Include field in store/restore of simulation state.
        /// @param   input True if field is an input field, false otherwise.
        /// @param   output True if field is an output field, false otherwise.
        virtual void PublishField(
            Smp::String8 name,
            Smp::String8 description,
            Smp::UInt64* address,
            Smp::ViewKind view = VK_None,
            Smp::Bool state = true,
            Smp::Bool input = false,
            Smp::Bool output = false) = 0;

        /// Publish a Float32 field.
        /// @param   name Field name.
        /// @param   description Field description.
        /// @param   address Field memory address.
        /// @param   view Show field in model tree.
        /// @param   state Include field in store/restore of simulation state.
        /// @param   input True if field is an input field, false otherwise.
        /// @param   output True if field is an output field, false otherwise.
        virtual void PublishField(
            Smp::String8 name,
            Smp::String8 description,
            Smp::Float32* address,
            Smp::ViewKind view = VK_None,
            Smp::Bool state = true,
            Smp::Bool input = false,
            Smp::Bool output = false) = 0;

        /// Publish a Float64 field.
        /// @param   name Field name.
        /// @param   description Field description.
        /// @param   address Field memory address.
        /// @param   view Show field in model tree.
        /// @param   state Include field in store/restore of simulation state.
        /// @param   input True if field is an input field, false otherwise.
        /// @param   output True if field is an output field, false otherwise.
        virtual void PublishField(
            Smp::String8 name,
            Smp::String8 description,
            Smp::Float64* address,
            Smp::ViewKind view = VK_None,
            Smp::Bool state = true,
            Smp::Bool input = false,
            Smp::Bool output = false) = 0;

        /// Publish a field of any type via Uuid.
        /// @param   name Field name.
        /// @param   description Field description.
        /// @param   address Field memory address.
```

```
        /// @param   typeUuid Uuid of field type (determines the size).
        /// @param   view Show field in model tree.
        /// @param   state Include field in store/restore of simulation state.
        /// @param   input True if field is an input field, false otherwise.
        /// @param   output True if field is an output field, false otherwise.
        /// @throws  Smp::IPublication::InvalidFieldType
        /// @throws  Smp::Publication::NotRegistered
        virtual void PublishField(
            Smp::String8 name,
            Smp::String8 description,
            void* address,
            Smp::Uuid typeUuid,
            Smp::ViewKind view = VK_None,
            Smp::Bool state = true,
            Smp::Bool input = false,
            Smp::Bool output = false) throw (
            Smp::IPublication::InvalidFieldType,
            Smp::Publication::NotRegistered) = 0;

        /// Publish array of simple type.
        /// @param   name Field name.
        /// @param   description Field description.
        /// @param   count Size of array.
        /// @param   address Field memory address.
        /// @param   type Array item type.
        /// @param   view Show field in model tree.
        /// @param   state Include field in store/restore of simulation state.
        /// @param   input True if field is an input field, false otherwise.
        /// @param   output True if field is an output field, false otherwise.
        /// @throws  Smp::IPublication::InvalidFieldType
        virtual void PublishArray(
            Smp::String8 name,
            Smp::String8 description,
            Smp::Int64 count,
            void* address,
            Smp::PrimitiveTypeKind type,
            Smp::ViewKind view = VK_None,
            Smp::Bool state = true,
            Smp::Bool input = false,
            Smp::Bool output = false) throw (
            Smp::IPublication::InvalidFieldType) = 0;

        /// Publish a Int64 field.
        /// @param   name Field name.
        /// @param   description Field description.
        /// @param   address Field memory address.
        /// @param   view Show field in model tree.
        /// @param   state Include field in store/restore of simulation state.
        /// @param   input True if field is an input field, false otherwise.
        /// @param   output True if field is an output field, false otherwise.
        virtual void PublishField(
            Smp::String8 name,
            Smp::String8 description,
            Smp::Int64* address,
            Smp::ViewKind view = VK_None,
            Smp::Bool state = true,
            Smp::Bool input = false,
            Smp::Bool output = false) = 0;

        /// Publish array of any type.
        /// @param   name Array name.
        /// @param   description Array description.
        /// @return  Interface to publish item type against.
        virtual Smp::IPublication* PublishArray(Smp::String8 name, Smp::String8 de-
scription) = 0;

        /// Publish structure.
        /// @param   name Structure name.
```

```
        /// @param   description Structure description.
        /// @return  Reference to publish structure fields against.
        virtual Smp::IPublication* PublishStructure(Smp::String8 name, Smp::String8
description) = 0;

        /// Publish an operation with complex return type.
        /// The operation parameters (including an optional return parameter)
        /// may be published against the returned IPublishOperation interface
        /// after calling PublishOperation().
        /// @param   name Operation name.
        /// @param   description Operation description.
        /// @param   view Show field in model tree.
        /// @return  Reference to publish parameters against.
        /// @throws  Smp::Publication::NotRegistered
        virtual Smp::Publication::IPublishOperation* PublishOperation(
            Smp::String8 name,
            Smp::String8 description,
            Smp::ViewKind view = true) throw (
            Smp::Publication::NotRegistered) = 0;

        /// Publish a property.
        /// @param   name Property name.
        /// @param   description Property description.
        /// @param   typeUuid Uuid of type of property.
        /// @param   accessKind Access kind of Property.
        /// @param   view Show field in model tree.
        /// @throws  Smp::Publication::NotRegistered
        virtual void PublishProperty(
            Smp::String8 name,
            Smp::String8 description,
            Smp::Uuid typeUuid,
            Smp::AccessKind accessKind,
            Smp::ViewKind view = true) throw (
            Smp::Publication::NotRegistered) = 0;

        /// Get the field of given name that is an array of a simple type.
        /// This method raises an exception of type InvalidFieldName if called
        /// with a field name for which no corresponding field exists or for
        /// which the corresponding field is not an array of simple type.
        ///
        /// This method can only be used to get array fields with items of
        /// simple type.
        /// @param   fullName Fully qualified array field name (relative to the
        ///          model)
        /// @return  Array field.
        /// @throws  Smp::Management::IManagedModel::InvalidFieldName
        virtual Smp::IArrayField* GetArrayField(Smp::String8 fullName) const throw
(
            Smp::Management::IManagedModel::InvalidFieldName) = 0;

        /// Get the field of given name that is typed by a simple type.
        ///
        /// This method raises an exception of type InvalidFieldName if called
        /// with a field name for which no corresponding field exists or for
        /// which the corresponding field is not of simple type.
        /// For getting access to fields of structured or array types, this
        /// method may be called multiply, for example by specifying a field
        /// name "MyField.Position[2]" in order to access an array item of a
        /// structure.
        /// This method can only be used to get fields of simple type.
        /// @param   fullName Fully qualified field name (relative to the model).
        /// @return  Simple field.
        /// @throws  Smp::Management::IManagedModel::InvalidFieldName
        virtual Smp::ISimpleField* GetSimpleField(Smp::String8 fullName) const
throw (
            Smp::Management::IManagedModel::InvalidFieldName) = 0;

        /// Create request object.
```

```
        /// Returns a request object for the given operation that describes the
        /// parameters and the return value.
        /// Request object may be undefined if no operation with given name
        /// could be found.
        /// @param   operationName Name of operation.
        /// @return  Request object for operation.
        virtual Smp::IRequest* CreateRequest(Smp::String8 operationName) = 0;

        /// Delete request object.
        /// Destroys a request object that has been created with the
        /// CreateRequest() method before.
        /// The request object must not be used anymore after DeleteRequest has
        /// been called for it.
        /// @param   request Request object to delete.
        virtual void DeleteRequest(Smp::IRequest* request) = 0;
    };
```

**Base Interfaces**

None

**Operations**

| Name | Description |
|---|---|
| CreateRequest | Create request object. |
| DeleteRequest | Delete request object. |
| GetArrayField | Get the field of given name that is an array of a simple type. |
| GetSimpleField | Get the field of given name that is typed by a simple type. |
| GetTypeRegistry | Give access to the global type registry. |
| PublishArray | Publish array of simple type. |
| PublishArray | Publish array of any type. |
| PublishField | Publish a Char8 field. |
| PublishField | Publish a Bool field. |
| PublishField | Publish a Int8 field. |
| PublishField | Publish a Int16 field. |
| PublishField | Publish a Int32 field. |
| PublishField | Publish a UInt8 field. |
| PublishField | Publish a UInt16 field. |
| PublishField | Publish a UInt32 field. |
| PublishField | Publish a UInt64 field. |
| PublishField | Publish a Float32 field. |
| PublishField | Publish a Float64 field. |
| PublishField | Publish a field of any type via Uuid. |
| PublishField | Publish a Int64 field. |
| PublishOperation | Publish an operation with complex return type. |
| PublishProperty | Publish a property |
| PublishStructure | Publish structure. |

#### 5.2.2.1 Create Request

Create request object.

Returns a request object for the given operation that describes the parameters and the return value.

Request object may be undefined if no operation with given name could be found.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| operationName | in | String8 | Name of operation. |
| | return | IRequest | Request object for operation. |

**Exceptions**

None

#### 5.2.2.2 Delete Request

Delete request object.

Destroys a request object that has been created with the CreateRequest() method before.

The request object must not be used anymore after DeleteRequest has been called for it.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| request | in | IRequest | Request object to delete. |

**Exceptions**

None

#### 5.2.2.3 Get Array Field

Get the field of given name that is an array of a simple type.

This method raises an exception of type InvalidFieldName if called with a field name for which no corresponding field exists or for which the corresponding field is not an array of simple type.

This method can only be used to get array fields with items of simple type.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| fullName | in | String8 | Fully qualified array field name (relative to the model) |
| | return | IArrayField | Array field. |

**Exceptions**

Smp::Management::IManagedModel::InvalidFieldName

#### 5.2.2.4 Get Simple Field

Get the field of given name that is typed by a simple type.

This method raises an exception of type InvalidFieldName if called with a field name for which no corresponding field exists or for which the corresponding field is not of simple type.

For getting access to fields of structured or array types, this method may be called multiply, for example by specifying a field name "MyField.Position[2]" in order to access an array item of a structure.

This method can only be used to get fields of simple type.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | ISimpleField | Simple field. |
| fullName | in | String8 | Fully qualified field name (relative to the model). |

**Exceptions**

Smp::Management::IManagedModel::InvalidFieldName

#### 5.2.2.5 Get Type Registry

Give access to the global type registry.

The type registry is typically a singleton, and must not be null, to allow use of existing types, and registration of new types.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| | return | ITypeRegistry | Reference to global type registry. |

**Exceptions**

None

#### 5.2.2.6 Publish Array

Publish array of simple type.

This method can only be used for arrays of simple type, as each simple type can be mapped to a primitive type. The memory layout of the array has to be without any padding, i.e. the array element with index i (0-based) is assumed to be stored at address + i*sizeof(primitiveType).

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Field name. |
| description | in | String8 | Field description. |
| count | in | Int64 | Size of array. |
| address | inout | void | Field memory address. |
| type | in | PrimitiveTypeKind | Array item type. |
| view | in | ViewKind | Show field in model tree. |
| state | in | Bool | Include field in store/restore of simulation state. |
| input | in | Bool | True if field is an input field, false otherwise. |
| output | in | Bool | True if field is an output field, false otherwise. |

**Exceptions**

Smp::IPublication::InvalidFieldType

#### 5.2.2.7 Publish Array

Publish array of any type.

This method can be used for arrays of any type. Individual array elements have to be added manually to the returned IPublication interface, where each array element can (and has to) be published with its own memory address.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Array name. |
| description | in | String8 | Array description. |
| | return | IPublication | Interface to publish item type against. |

**Exceptions**

None

#### 5.2.2.8 Publish Field

Publish a Char8 field.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Field name. |
| description | in | String8 | Field description. |
| address | inout | Char8 | Field memory address. |
| view | in | ViewKind | Show field in model tree. |
| state | in | Bool | Include field in store/restore of simulation state. |
| input | in | Bool | True if field is an input field, false otherwise. |
| output | in | Bool | True if field is an output field, false otherwise. |

**Exceptions**

None

#### 5.2.2.9 Publish Field

Publish a Bool field.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Field name. |
| description | in | String8 | Field description. |
| address | inout | Bool | Field memory address. |
| view | in | ViewKind | Show field in model tree. |
| state | in | Bool | Include field in store/restore of simulation state. |
| input | in | Bool | True if field is an input field, false otherwise. |
| output | in | Bool | True if field is an output field, false otherwise. |

**Exceptions**

None

#### 5.2.2.10 Publish Field

Publish a Int8 field.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Field name. |
| description | in | String8 | Field description. |
| address | inout | Int8 | Field memory address. |
| view | in | ViewKind | Show field in model tree. |
| state | in | Bool | Include field in store/restore of simulation state. |
| input | in | Bool | True if field is an input field, false otherwise. |
| output | in | Bool | True if field is an output field, false otherwise. |

**Exceptions**

None

#### 5.2.2.11 Publish Field

Publish a Int16 field.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Field name. |
| description | in | String8 | Field description. |
| address | inout | Int16 | Field memory address. |

| Name | Dir. | Type | Description |
|---|---|---|---|
| view | in | ViewKind | Show field in model tree. |
| state | in | Bool | Include field in store/restore of simulation state. |
| input | in | Bool | True if field is an input field, false otherwise. |
| output | in | Bool | True if field is an output field, false otherwise. |

**Exceptions**

None

#### 5.2.2.12 Publish Field

Publish a Int32 field.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Field name. |
| description | in | String8 | Field description. |
| address | inout | Int32 | Field memory address. |
| view | in | ViewKind | Show field in model tree. |
| state | in | Bool | Include field in store/restore of simulation state. |
| input | in | Bool | True if field is an input field, false otherwise. |
| output | in | Bool | True if field is an output field, false otherwise. |

**Exceptions**

None

#### 5.2.2.13 Publish Field

Publish a UInt8 field.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Field name. |
| description | in | String8 | Field description. |
| address | inout | UInt8 | Field memory address. |
| view | in | ViewKind | Show field in model tree. |
| state | in | Bool | Include field in store/restore of simulation state. |
| input | in | Bool | True if field is an input field, false otherwise. |
| output | in | Bool | True if field is an output field, false otherwise. |

**Exceptions**

None

#### 5.2.2.14 Publish Field

Publish a UInt16 field.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Field name. |
| description | in | String8 | Field description. |
| address | inout | UInt16 | Field memory address. |
| view | in | ViewKind | Show field in model tree. |
| state | in | Bool | Include field in store/restore of simulation state. |
| input | in | Bool | True if field is an input field, false otherwise. |
| output | in | Bool | True if field is an output field, false otherwise. |

**Exceptions**

None

#### 5.2.2.15 Publish Field

Publish a UInt32 field.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Field name. |
| description | in | String8 | Field description. |
| address | inout | UInt32 | Field memory address. |
| view | in | ViewKind | Show field in model tree. |
| state | in | Bool | Include field in store/restore of simulation state. |
| input | in | Bool | True if field is an input field, false otherwise. |
| output | in | Bool | True if field is an output field, false otherwise. |

**Exceptions**

None

#### 5.2.2.16 Publish Field

Publish a UInt64 field.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Field name. |
| description | in | String8 | Field description. |
| address | inout | UInt64 | Field memory address. |
| view | in | ViewKind | Show field in model tree. |
| state | in | Bool | Include field in store/restore of simulation state. |
| input | in | Bool | True if field is an input field, false otherwise. |
| output | in | Bool | True if field is an output field, false otherwise. |

**Exceptions**

None

#### 5.2.2.17 Publish Field

Publish a Float32 field.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Field name. |
| description | in | String8 | Field description. |
| address | inout | Float32 | Field memory address. |
| view | in | ViewKind | Show field in model tree. |
| state | in | Bool | Include field in store/restore of simulation state. |
| input | in | Bool | True if field is an input field, false otherwise. |
| output | in | Bool | True if field is an output field, false otherwise. |

**Exceptions**

None

#### 5.2.2.18 Publish Field

Publish a Float64 field.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Field name. |
| description | in | String8 | Field description. |
| address | inout | Float64 | Field memory address. |
| view | in | ViewKind | Show field in model tree. |
| state | in | Bool | Include field in store/restore of simulation state. |
| input | in | Bool | True if field is an input field, false otherwise. |
| output | in | Bool | True if field is an output field, false otherwise. |

**Exceptions**

None

#### 5.2.2.19 Publish Field

Publish a field of any type via Uuid.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Field name. |
| description | in | String8 | Field description. |
| address | inout | void | Field memory address. |

| Name | Dir. | Type | Description |
|---|---|---|---|
| typeUuid | in | Uuid | Uuid of field type (determines the size). |
| view | in | ViewKind | Show field in model tree. |
| state | in | Bool | Include field in store/restore of simulation state. |
| input | in | Bool | True if field is an input field, false otherwise. |
| output | in | Bool | True if field is an output field, false otherwise. |

**Exceptions**

Smp::IPublication::InvalidFieldType, Smp::Publication::NotRegistered

#### 5.2.2.20 Publish Field

Publish a Int64 field.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Field name. |
| description | in | String8 | Field description. |
| address | inout | Int64 | Field memory address. |
| view | in | ViewKind | Show field in model tree. |
| state | in | Bool | Include field in store/restore of simulation state. |
| input | in | Bool | True if field is an input field, false otherwise. |
| output | in | Bool | True if field is an output field, false otherwise. |

**Exceptions**

None

#### 5.2.2.21 Publish Operation

Publish an operation with complex return type.

The operation parameters (including an optional return parameter) may be published against the returned IPublishOperation interface after calling PublishOperation().

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Operation name. |
| description | in | String8 | Operation description. |
| view | in | ViewKind | Show field in model tree. |
| | return | IPublishOperation | Reference to publish parameters against. |

**Exceptions**

Smp::Publication::NotRegistered

#### 5.2.2.22 Publish Property

Publish a property.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Property name. |
| description | in | String8 | Property description. |
| typeUuid | in | Uuid | Uuid of type of property. |
| accessKind | in | AccessKind | Access kind of Property. |
| view | in | ViewKind | Show field in model tree. |

**Exceptions**

Smp::Publication::NotRegistered

#### 5.2.2.23 Publish Structure

Publish structure.

**Parameters**

| Name | Dir. | Type | Description |
|---|---|---|---|
| name | in | String8 | Structure name. |
| description | in | String8 | Structure description. |
| | return | IPublication | Reference to publish structure fields against. |

**Exceptions**

None

#### 5.2.2.24 Invalid Field Type

Invalid field type.

This exception is raised when trying to publish a field with invalid type.

**File**

#include "Smp/IPublication.h"

**Namespace**

Smp::IPublication

**Declaration of InvalidFieldType**

```
        /// Invalid field type.
        /// This exception is raised when trying to publish a field with
        /// invalid type.
        /// @remarks This can happen, for example, when trying to publish a
        ///          field of the variable-length simple type String8.
        class InvalidFieldType : public Smp::Exception
        {
        public:
            /// Constructor for new exception.
            InvalidFieldType() throw();

            /// Copy constructor.
            InvalidFieldType(
```

```
                InvalidFieldType& ex) throw();

            /// Virtual destructor to release memory.
            virtual ~InvalidFieldType();

        };
```

***Remark:*** This can happen, for example, when trying to publish a field of the variable-length simple type String8.

**Fields**

None

### 5.2.3 Access Kind

The Access Kind of a property defines whether it has getter and setter.

**File**

#include "Smp/IPublication.h"

**Namespace**

Smp

**Declaration of AccessKind**

```
    /// Unique Identifier of type AccessKind.
    extern const Uuid Uuid_AccessKind;

    /// The Access Kind of a property defines whether it has getter and setter.
    enum AccessKind
    {
        /// Read/Write access, i.e. getter and setter.
        AK_ReadWrite,

        /// Read only access, i.e. only getter method.
        AK_ReadOnly,

        /// Write only access, i.e. only setter method.
        AK_WriteOnly
    };
```

**Table 8 - Enumeration Literals of AccessKind**

| Name | Description |
|---|---|
| AK_ReadWrite | Read/Write access, i.e. getter and setter. |
| AK_ReadOnly | Read only access, i.e. only getter method. |
| AK_WriteOnly | Write only access, i.e. only setter method. |

### 5.2.4 Parameter Direction Kind

The Parameter Direction Kind enumeration defines the possible parameter directions.

**File**

#include "Smp/Publication/IPublishOperation.h"

**Namespace**

Smp::Publication

**Declaration of ParameterDirectionKind**

```
/// Unique Identifier of type ParameterDirectionKind.
extern const Uuid Uuid_ParameterDirectionKind;

/// The Parameter Direction Kind enumeration defines the possible
/// parameter directions.
enum ParameterDirectionKind
{
    /// The parameter is read-only to the operation, i.e. its value
    /// must be specified on call, and cannot be changed inside the
    /// operation.
    PDK_In,

    /// The parameter is write-only to the operation, i.e. its value is
    /// unspecified on call, and must be set by the operation.
    PDK_Out,

    /// The parameter must be specified on call, and may be changed by
    /// the operation.
    PDK_InOut,

    /// The parameter represents the operation's return value.
    PDK_Return
};
```

**Table 9 - Enumeration Literals of ParameterDirectionKind**

| Name | Description |
|---|---|
| PDK_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. |
| PDK_Out | The parameter is write-only to the operation, i.e. its value is un-specified on call, and must be set by the operation. |
| PDK_InOut | The parameter must be specified on call, and may be changed by the operation. |
| PDK_Return | The parameter represents the operation's return value. |

# 6 Metamodel

This section describes the mapping of all relevant SMP Metamodel elements into ISO/ANSI C++. The exact definition of this mapping is essential in order to allow transforming modelling information stored in an SMDL Catalogue or Package into ISO/ANSI C++ code in a unique way.

## 6.1 Overview

The C++ mapping of the SMP Metamodel produces code fragments, complete files or directories depending on the kind of Element mapped. The following description of the SMP Metamodel mapping uses a template like notation to describe the code fragments or files in a generic way.

### 6.1.1 Placeholders

Non-general information coming out of the SMDL file, like names or types, is mapped using placeholders. Placeholders are encased with the '$' symbol.
**Example:**

`$Component.Name$`
This placeholder shall be replaced by the actual name of the component.

### 6.1.2 Coloring and Font Schema

The coloring schema of the mapping description uses blue for C++ keywords, black for the rest of the source code and green for a comment. The font of the mapped code is `Courier`.
**Example:**

```
// An Smp.Int32 is a 32 bit signed Integer type:
typedef signed int Int32;
```

### 6.1.3 Generation of Type Identification

The expression `TypeName()` delivers the fully qualified name of a type. When used in a header file, the fully qualified name shall be the type name plus preceding nesting namespaces.
**Example:**
`TypeName($Parameter.Type$)`
with a parameter of type `MyType` defined in a nested namespace of a namespace may be replaced in a header file by:
`::Namespace1::NestedNamespace2::MyType`

***Remark*:** With the `using` keyword, it is possible to write more readable source code, but at the risk of introducing ambiguities due to naming conflicts, as it is possible and valid to use the same type name within different namespaces. The namespace is indicated by the `using` command prefixed with all namespaces the namespace is nested in (if there are any). For the example above, this would look as follows:

```
// The used namespace chain the type is nested in is defined.
using namespace ::Namespace1::NestedNamespace2;
```

After that, `MyType` can be referenced without a namespace prefix.

### 6.1.4 Alternative Code

Alternative code fragments are separated by the '|' separator, where exactly one of the alternatives must be present.

**Example:**

```
// Set Visibility of Type
public:|protected:|private:
```

This means that one of three possible visibility keywords must be set.

### 6.1.5 Optional Code

Optional code is encased in brackets ('[' and ']').

**Example:**

```
// If embedded in a Class or Component set Visibility of Type
[public:|protected:|private:]
```

This means that a visibility keyword may be set (and then one of the three possibilities can be chosen).

## 6.2 C++ Specific Attributes

The C++ platform mapping of the Metamodel is supported by a number of C++ specific attributes. These have been defined to support C++ specific features, such as constructors and operators, which are not included in the Metamodel (as they may not map to other target platforms). However, to be able to use a model driven development process, where source code is auto-generated from a design captured in an SMDL Catalogue, it is essential to support these C++ specific features. By adding these attributes to the normative C++ Mapping, it can be ensured that all supporting tools (especially C++ Code Generators) will support these mechanisms in a consistent way.

### 6.2.1 C++ Attributes

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

#### 6.2.1.1 Abstract

The Abstract attribute specifies that an operation or property is abstract, i.e. that it must be overridden in a derived type. The default value for this attribute is false.

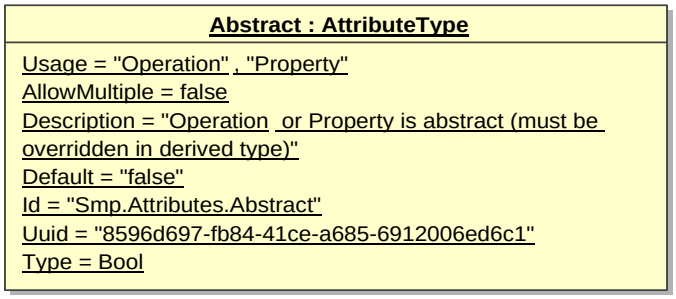


**Figure 6 - Abstract**

#### 6.2.1.2 Base Class

The BaseClass attribute specifies a name of a C++ class that shall be used as base class (implementation inheritance) for the Class or Model implementation. This can be used to inherit the implementation from a non-SMP C++ class into an SMP Class or Model implementation, for example to wrap an existing C++ implementation as SMP Class or Model. The default value for this attribute is the empty string, which corresponds to no base class.

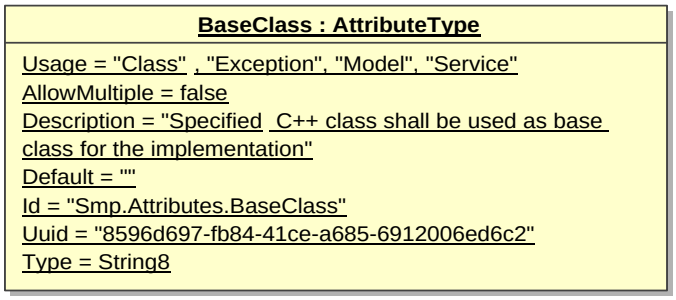


**Figure 7 - Base Class**

#### 6.2.1.3 By Reference

The ByReference attribute specifies that a parameter is passed by reference, i.e. as pointer in C++. The default value for this attribute is false.

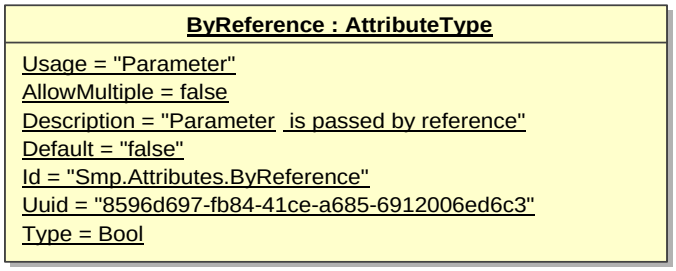


**Figure 8 - By Reference**

#### 6.2.1.4 Const

The Const attribute specifies that a feature is "constant" in the following sense:

- Association: The value of the referenced element is constant, i.e. it cannot be changed during runtime.
- Operation: The state of the containing type (e.g. Model) is constant, i.e. the operation must not change any field values during execution.
- Property: The state of the containing type (e.g. Model) is constant, i.e. the property getter must not change any field values during execution.
- Parameter: The value of the parameter is constant, i.e. the operation must not change it during execution. When applied to reference parameters (pointers in C++), the referenced element must not be changed.

The default value for this attribute is false.

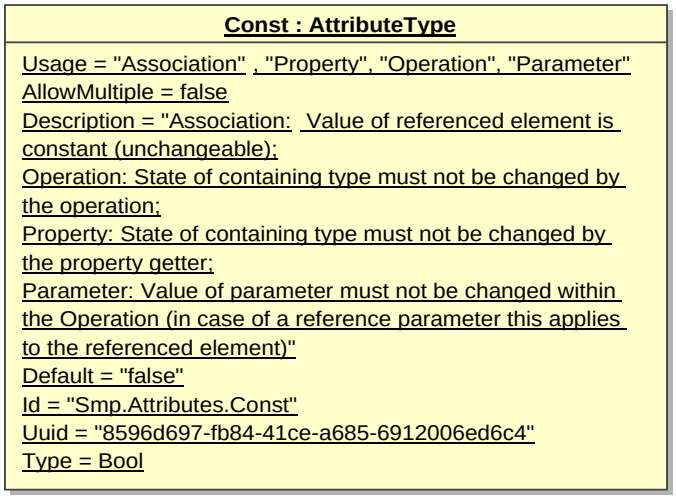


**Figure 9 - Const**

***Remark:*** This attribute cannot be applied to Fields since there is a first-level concept (Constant) for the purpose of specifying constant values.

#### 6.2.1.5 Constructor

The Constructor attribute specifies that the operation is mapped to a C++ constructor. The default value for this attribute is false, which corresponds to not mapping to a constructor.

A constructor must not have a return parameter.

The name of the constructor is ignored as the Class or Model name is used in C++.

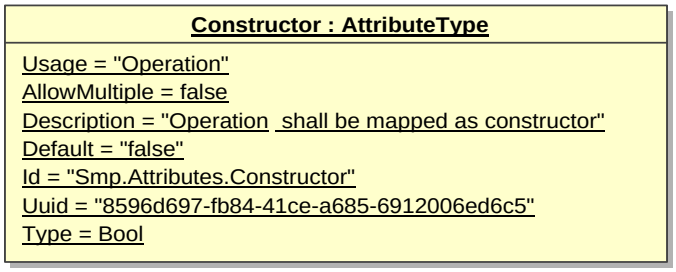


**Figure 10 - Constructor**

#### 6.2.1.6 Operator

The Operator attribute defines an operator kind for an operation. It can be used to specify that the operation is mapped to a C++ operator. The default value for this attribute is None, which corresponds to not mapping to an operator.

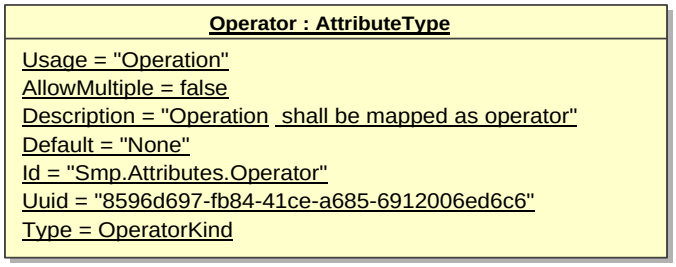


**Figure 11 - Operator**

#### 6.2.1.7 Operator Kind

This enumeration defines possible operator kinds.

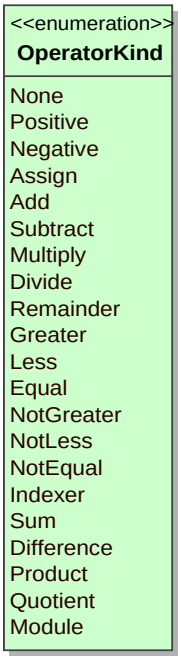


**Figure 12 - Operator Kind**

**Table 10 - Enumeration Literals of OperatorKind**

| Name | Description |
|---|---|
| None | Undefined |
| Positive | Positive value of instance.<br>C++ Example: +x |
| Negative | Negative value of instance.<br>C++ Example: -x |
| Assign | Assigns new value to instance.<br>C++ Example: x = a |
| Add | Adds value to instance.<br>C++ Example: x += a |
| Subtract | Subtracts value to instance.<br>C++ Example: x -= a |
| Multiply | Multiplies instance with value.<br>C++ Example: x *= a |
| Divide | Divides instance by value.<br>C++ Example: x /= a |
| Remainder | Remainder of instance for value.<br>C++ Example: x %= a |
| Greater | Compares whether instance is greater than value.<br>C++ Example: x > a |

| Name | Description |
|---|---|
| Less | Compares whether instance is less than value.<br>C++ Example: x < a |
| Equal | Compares whether instance is equal to value.<br>C++ Example: x == a |
| NotGreater | Compares whether instance is not greater than value.<br>C++ Example: x <= a |
| NotLess | Compares whether instance is not less than value.<br>C++ Example: x >= a |
| NotEqual | Compares whether instance is not equal to value.<br>C++ Example: x != a |
| Indexer | Returns indexed value of instance.<br>C++ Example: x[a] |
| Sum | Returns sum of two values.<br>C++ Example: a + b |
| Difference | Returns difference of two values.<br>C++ Example: a - b |
| Product | Returns product of two values.<br>C++ Example: a * b |
| Quotient | Returns quotient of two values.<br>C++ Example: a / b |
| Module | Returns remainder of two values.<br>C++ Example: a % b |

#### 6.2.1.8 Static

The Static attribute specifies that a feature is static, i.e. that it is defined on type/classifier scope. The default value for this attribute is false, which corresponds to instance scope.

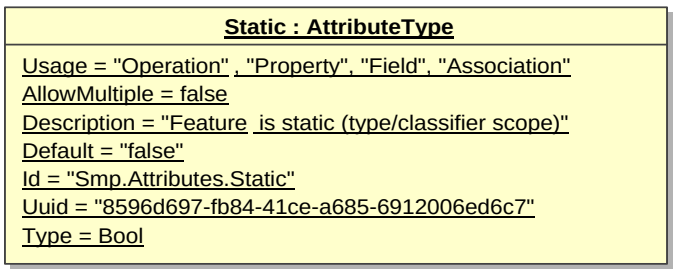


**Figure 13 - Static**

#### 6.2.1.9 Virtual

The Virtual attribute specifies that an operation or property is virtual, i.e. that it may be overridden in a derived type (polymorphism). The default value for this attribute is false.

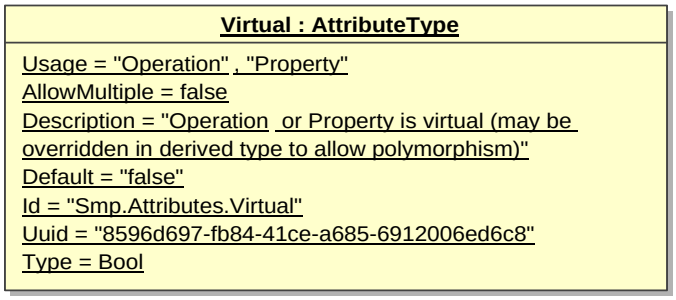


**Figure 14 - Virtual**

## 6.3 Core Elements

### 6.3.1 Basic Types

The Core Elements schema defines some basic types which are used for attributes and elements of other types.

#### 6.3.1.1 Identifier

The `Identifier` type, respectively the `Id` attribute of `NamedElement` is not mapped to C++.

#### 6.3.1.2 Name

The `Name` type, respectively the `Name` attribute of `NamedElement` is used for the name of a C++ equivalent of a mapped SMDL element. As an example, C++ fields, operations, types (such as structures and classes) and namespaces are named according to the `Name` attribute of the corresponding SMDL element in a catalogue.

#### 6.3.1.3 Description

The `Description` type, respectively the `Description` element of `NamedElement` is not mapped to C++, but may be generated as a comment in the description of a C++ equivalent of a mapped SMDL element.

#### 6.3.1.4 UUID

The `UUID` type is mapped to the equivalent `Uuid` type of the Component Model.

### 6.3.2 Elements

#### 6.3.2.1 Named Element

A `NamedElement` is not mapped directly into C++, because it is an abstract base type. Nevertheless types derived from `NamedElement` use the inherited `Name` at-

tribute for the naming of their C++ equivalent (see section 6.3.1.2 on Name). The `Description` element is not mapped to C++, but may be generated as a comment (see section 6.3.1.3 on Description).

#### 6.3.2.2 Document

The `Document` type is the abstract base type for XML files defined by SMDL. It is not mapped to C++, but the derived `Package` document type is (see section 6.6.1.1 on Package).

### 6.3.3 Metadata

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

#### 6.3.3.1 Metadata

`Metadata` is an abstract base type and not mapped to C++.

#### 6.3.3.2 Comment

A `Comment` is not mapped to C++, but may be generated as code comment.

#### 6.3.3.3 Documentation

`Documentation` is not mapped to C++, but may be generated as code documentation.

## 6.4 Core Types

### 6.4.1 Types

Types are used in different contexts. The most common type is a `LanguageType`, which has a mapping to the C++ programming language via the types derived from it. Typing is used as well for other mechanisms, e.g. for Attributes and Events.

#### 6.4.1.1 Visibility Element

A `VisibilityElement` is not directly mapped to C++, because it is an abstract base type. Types and features derived from `VisibilityElement` map the inherited `Visibility` attribute to define their C++ visibility (see Visibility Kind below).

#### 6.4.1.2 Visibility Kind

The `VisibilityKind` attribute is mapped to C++ according to the following table:

| Visibility Kind | C++ Visibility |
|---|---|
| private | private |
| protected | protected |
| package | public |
| public | public |

The C++ visibility coincides with the `Visibility` attribute except for `package`, which is not supported by C++.

#### 6.4.1.3 Type

`Type` is an abstract base type and not mapped to C++. However, the `Uuid` attribute of inherited types is mapped to a constant of type `Uuid`:

```
/// Unique Identifier of type $Type.Name$.
extern const Smp::Uuid Uuid_$Type.Name$;
```

The implementation has to be provided in another file, typically in a source file. The mapping may as well use a type derived from `Smp::Uuid`.

#### 6.4.1.4 Language Type

`LanguageType` is an abstract base type and not mapped to C++. All types derived from `LanguageType` have a mapping to C++. See section 6.4.2 for Value Types, section 6.4.1.6 for Value Reference, section 6.4.1.7 for Native Type and Platform Mapping, and section 6.5.2 for Reference Types.

#### 6.4.1.5 Value Type

`ValueType` is an abstract base type and not mapped to C++. See section 6.4.2 on Value Types for derived types.

#### 6.4.1.6 Value Reference

A `ValueReference` is mapped as a `typedef` of a pointer to a `ValueType`.

- In case the `ValueReference` belongs to a `ReferenceType` (i.e. is a nested type), the `Visibility` is defined.

```
// If embedded in a Reference Type set Visibility
[public:|protected:|private:]

// Define Value Reference as pointer to existing Value Type
typedef TypeName($ValueReference.Type$)* $ValueReference.Name$;
```

#### 6.4.1.7 Native Type and Platform Mapping

A `NativeType` is mapped as a `typedef` to an existing C++ type if (and only if) it has a `PlatformMapping` to the C++ platform. The following holds:

- A `PlatformMapping` is considered to be for the C++ platform if (and only if) its `Name` starts with "`cpp`".
- A `PlatformMapping` with `Name` equals to "`cpp`" is called the **default** C++ mapping.
- A `PlatformMapping` with a `Name` not equals to "`cpp`" is called a **conditional** C++ mapping, where the remainder of the `Name` is called the **Condition** (called `$PlatformMapping.Condition$` below).

A `NativeType` is mapped as follows:

- A `NativeType` with only a default C++ mapping is mapped as an `#include` and a `typedef`, where the include statement is added at the beginning of the file:

```
```

```
// Include header file if Platform Mapping has one
[#include "$PlatformMapping.Location$"]

// If embedded in a Reference Type set Visibility
[public:|protected:|private:]

// Use Namespace if Platform Mapping has one
typedef [$PlatformMapping.Namespace$::]$PlatformMapping.Type$ $NativeType.Name$;
```

- A `NativeType` with only conditional C++ mappings is mapped as a switch of conditional `typedef` statements, where the order of conditions is defined as the order of the `PlatformMapping` elements:

```
// Include header file if Platform Mapping has one
#if   defined ($PlatformMapping1.Condition$)
   [#include "$PlatformMapping1.Location$"]
#elif defined ($PlatformMapping2.Condition$)
   [#include "$PlatformMapping2.Location$"]
...
#endif

// If embedded in a Reference Type set Visibility
[public:|protected:|private:]

// Define Native Type via typedef
#if   defined ($PlatformMapping1.Condition$)
   typedef [$PlatformMapping1.Namespace$::]$PlatformMapping1.Type$ $NativeType.Name$;
#elif defined ($PlatformMapping2.Condition$)
   typedef [$PlatformMapping2.Namespace$::]$PlatformMapping2.Type$ $NativeType.Name$;
...
#endif
```

- A `NativeType` with both a default and conditional C++ mappings is mapped as a switch of conditional `typedef` statements with a final `#else` statement, where the order of conditions is defined as the order of the `PlatformMapping` elements (except for the default mapping, which is always generated at the end):

```
// Include header file if Platform Mapping has one
#if   defined ($PlatformMapping1.Condition$)
   [#include "$PlatformMapping1.Location$"]
#elif defined ($PlatformMapping2.Condition$)
   [#include "$PlatformMapping2.Location$"]
...
#else
   [#include "$PlatformMappingX.Location$"]
#endif

// If embedded in a Reference Type set Visibility
[public:|protected:|private:]

// Define Native Type via typedef
#if   defined ($PlatformMapping1.Condition$)
   typedef [$PlatformMapping1.Namespace$::]$PlatformMapping1.Type$ $NativeType.Name$;
#elif defined ($PlatformMapping2.Condition$)
```

```
   typedef [$PlatformMapping2.Namespace$::]$PlatformMapping2.Type$ $NativeType.Name$;
...
#else
   typedef [$PlatformMappingX.Namespace$::]$PlatformMappingX.Type$ $NativeType.Name$;
#endif
```

### 6.4.2 Value Types

#### 6.4.2.1 Simple Type

`SimpleType` is an abstract base type and not mapped to C++.

#### 6.4.2.2 Primitive Type

All primitive types are mapped to basic ISO/ANSI C++ types. As this mapping depends on hardware platform (e.g. 32 bit or 64 bit) and compiler, the actual mapping is provided in an include file called `Platform.h`.

#### 6.4.2.3 Enumeration

An `Enumeration` is mapped to an ISO/ANSI C++ `enum` type. In addition, a function is defined that registers the enumeration and all its literals in the type registry. In case the `Enumeration` belongs to a `ReferenceType` (i.e. is a nested type), the `Visibility` is defined. In this case, the function becomes a `static` method of the reference type with the same visibility as the type.

```
// If embedded in a Reference Type set Visibility
[public:|protected:|private:]

enum $Enum.Name$
{
   // Comma-separated mapping of Enumeration Literals
   See 6.4.2.4 (Enumeration Literal)
};

/// Register type in type registry
[static] void _Register_$Enum.Name$(Smp::Publication::ITypeRegistry* registry);
```

#### 6.4.2.4 Enumeration Literal

Each `EnumerationLiteral` is mapped as C++ `enum` literal with value assignment.

```
$EnumerationLiteral.Name$ = $EnumerationLiteral.Value$
```

#### 6.4.2.5 Integer

An `Integer` type is mapped as a `typedef` to the primitive type it references, or to `Smp::Int32` if it does not reference a type. The `Maximum`, `Minimum` and `Unit` attributes are not mapped, but can be shown as comments in the code. In addition, a function is defined that registers the integer type (with its limits and unit) in the type registry.

- In case the `Integer` belongs to a `ReferenceType` (i.e. is a nested type), the `Visibility` is defined. In this case, the function becomes a `static` method of the reference type with the same visibility as the type.

```
// If embedded in a Reference Type set Visibility
[public:|protected:|private:]

// Define Integer as typedef to existing Integer type, with default of Int32
typedef $Integer.Type$|Smp::Int32 $Integer.Name$;

/// Register type in type registry
[static] void _Register_$Integer.Name$(Smp::Publication::ITypeRegistry* registry);
```

#### 6.4.2.6 Float

A `Float` type is mapped as a `typedef` to either `Smp::Float32` or `Smp::Float64`. The `Maximum`, `Minimum`, `MaxInclusive`, `MinInclusive` and `Unit` attributes are not mapped, but can be shown as comments. In addition, a function is defined that registers the float type (with its limits and unit) in the type registry.

- In case the `Float` belongs to a `ReferenceType` (i.e. is a nested type), the `Visibility` is defined. In this case, the function becomes a `static` method of the reference type with the same visibility as the type.

```
// If embedded in a Reference Type set Visibility
[public:|protected:|private:]

// Define Integer as typedef to existing Float type, with default of Float64
typedef $Float.Type$|Smp::Float64 $Float.Name$;

/// Register type in type registry
[static] void _Register_$Float.Name$(Smp::Publication::ITypeRegistry* registry);
```

#### 6.4.2.7 String

A `String` type is mapped as an ISO/ANSI C++ `struct` with an internal fixed size array of type `Smp::Char8`. The size of the array is given by the `Length` attribute of the `String`, but extended by one to ensure the terminating null character fits into the string. In addition, a function is defined that registers the string type in the type registry.

- In case the `String` belongs to a `ReferenceType` (i.e. is a nested type), the `Visibility` is defined. In this case, the function becomes a `static` method of the reference type with the same visibility as the type.

```
// If embedded in a Reference Type set Visibility
[public:|protected:|private:]

// Define String as structure with array field
struct $String.Name$
{
  Smp::Char8 internalString[$String.Length$+1];
```

```
};

/// Register type in type registry
[static] void _Register_$String.Name$(Smp::Publication::ITypeRegistry* registry);
```

#### 6.4.2.8 Array

An `Array` type is mapped as an ISO/ANSI C++ `struct` (not as a `typedef` of an `array`)[1]. In addition, a function is defined that registers the array type in the type registry.

- In case the `Array` belongs to a `ReferenceType` (i.e. is a nested type), the `Visibility` is defined. In this case, the function becomes a `static` method of the reference type with the same visibility as the type.

```
// If embedded in a Reference Type set Visibility
[public:|protected:|private:]

// Define Array as structure with array field
struct $Array.Name$
{
  TypeName($Array.ItemType$) internalArray[$Array.Size$];
};

/// Register type in type registry
[static] void _Register_$Array.Name$(Smp::Publication::ITypeRegistry* registry);
```

#### 6.4.2.9 Structure

A `Structure` type is mapped as an ISO/ANSI C++ `struct`. Each field of the structure is mapped to a field in C++. In addition, a static method is defined that registers the structure type in the type registry.

- In case the `Structure` belongs to a `ReferenceType` (i.e. is a nested type), the `Visibility` is defined.

```
// If embedded in a Reference Type set Visibility
[public:|protected:|private:]

// Define Structure as struct with static _Register method
struct $Structure.Name$
{
  // Define the Fields of the Structure here
  See 6.4.3.2 (Field)

  // Register type in type registry
  static void _Register(Smp::Publication::ITypeRegistry* registry);
};
```

If a forward declaration is needed, this can be generated as follows:

```
// If embedded in a Reference Type set Visibility
```

[1] This is because an array cannot be used as return type of an operation in C++.

```
[public:|protected:|private:]

struct $Structure.Name$;
```

#### 6.4.2.10 Class

A `Class` is mapped as a C++ `class`. In addition, a static method is defined that registers the class type in the type registry.

- In case the `Class` belongs to a `ReferenceType` (i.e. is a nested type), the `Visibility` is defined.
- In case the `Class` has a `Base` class defined, it inherits from the specified base class.
- In case the `Class` has no constructor defined (i.e. no `Operation` with the `Constructor` attribute), a default constructor is generated.
- A virtual destructor is always generated.
- All features (`Constant`, `Field`, `Property`, `Association`, `Operation`) are generated.

The mapping can be tailored via the following attributes:

- In case the `Class` has a `BaseClass` attribute, it inherits from the specified base class (possibly in addition to the inheritance from the specified `Base` class, as C++ supports multiple implementation inheritance).

```
// If embedded in a Reference Type set Visibility
[public:|protected:|private:]

// If the Class has a Base Class it needs to inherit from it.
class $Class.Name$ [: public TypeName($Class.Base$)][, public $BaseClass.Name$]
{
  // Constructor and virtual Destructor are declared with Visibility of the Class
  public:|protected:|private:

  // Default constructor is generated if no other constructors are defined
  [$Class.Name$(void);]

  // Virtual destructor is always generated
  virtual ~$Class.Name$(void);

  // If the Class has Constants, these are declared:
  See 6.4.3.1 (Constant)

  // If the Class has Fields, these are declared:
  See 6.4.3.2 (Field)

  // If the Class has Properties, these are declared virtual:
  See 6.4.3.3 (Property)

  // If the Class has associations, these are declared:
  See 6.4.3.4 (Association)

  // If the Class has Operations, these are declared virtual:
  See 6.4.3.5 (Operation)
```

```
  // Register type in type registry
  static void _Register(Smp::Publication::ITypeRegistry* registry);
};
```

#### 6.4.2.11 Exception

An `Exception` is mapped to a C++ `class`. In addition, a static method is defined that registers the exception type (as a class type) in the type registry.

- In case the `Exception` belongs to a `ReferenceType` (i.e. is a nested type), the `Visibility` is defined.
- In case the `Exception` has a `Base` class defined, it inherits from the specified base class. Otherwise, it inherits from `Smp::Exception`.
- In case the `Exception` has no constructor defined (i.e. no `Operation` with the `Constructor` attribute), a default constructor is generated.
- A copy constructor is always generated, as this is needed to be able to catch exceptions by value.
- A virtual destructor is always generated.
- All features (`Constant`, `Field`, `Property`, `Association`, `Operation`) are generated.

The mapping can be tailored via the following attributes:

- In case the `Exception` has a `BaseClass` attribute, it inherits from the specified base class (in addition to the inheritance from the specified `Base` class or `Smp::Exception`, as C++ supports multiple implementation inheritance).

```
// If embedded in a Reference Type set Visibility
[public:|protected:|private:]

// If the Exception has a Base Exception it needs to inherit from it.
class $Exception.Name$ :
  public TypeName($Exception.Base$)|Smp::Exception[, public $BaseClass.Name$]
{
  // Constructors and virtual Destructor are declared with Visibility of the Exception
  public:|protected:|private:

  // Default constructor is generated if no other constructors are defined
  [$Exception.Name$(void) throw();]

  // Copy constructor is always generated, to allow catching exception by value
  $Exception.Name$($Exception.Name$& exception) throw();

  // Virtual destructor is always generated
  virtual ~$Exception.Name$(void);

  // If the Exception has Constants, these are declared:
  See 6.4.3.1 (Constant)

  // If the Exception has Fields, these are declared:
  See 6.4.3.2 (Field)

  // If the Exception has Properties, these are declared virtual:
  See 6.4.3.3 (Property)
```

```
    // If the Exception has associations, these are declared:
    See 6.4.3.4 (Association)

    // If the Exception has Operations, these are declared virtual:
    See 6.4.3.5 (Operation)

    // Register type in type registry
    static void _Register(Smp::Publication::ITypeRegistry* registry);
};
```

The mapping on an `Exception` is therefore almost identical to the mapping of a `Class`, but each `Exception` must inherit from another `Exception` (with the default `Smp::Exception`). Further, an `Exception` always needs a copy constructor, to allow catching exceptions by value.

### 6.4.3 Features

Features are either contained in a value type (`Structure`, `Class`, `Exception`), or in a reference type (`Interface`, `Model`, `Service`).

#### 6.4.3.1 Constant

A `Constant` is mapped to C++ as a `static const` member variable of a C++ `struct` or a C++ `class`.

- The `Visibility` is defined.

```
// Set Visibility
public:|protected:|private:

// Constant of value type.
static const TypeName($Constant.Type$) $Constant.Name$;
```

In addition, an implementation with the value of the `Constant` needs to be defined in a source file.

For the mapping of a `Value`, see section 6.4.4 on Values below.

***Remark***: In case the `Constant` belongs to a `Structure` the visibility should be `public`.

#### 6.4.3.2 Field

A `Field` is mapped to C++ as a member variable of a C++ `struct` or a C++ `class`.

- The `Visibility` is defined.

The mapping can be tailored via the following attributes:

- If the `Field` has the `Static` attribute, it is declared `static`.
- If the `Field` has the `Forcible` attribute, an additional field for the forced value, and a Boolean field for the forcing status are generated.

The `View` attribute is only taken into account in the publication of the field.

```
// Set Visibility
public:|protected:|private:

// Field of value type.
[static] TypeName($Field.Type$) $Field.Name$;
```

```
[// Forced value of field of value type.
[static] TypeName($Field.Type$) _forced_$Field.Name$;

// Flag whether field is forced.
[static] Smp::Bool _isForced_$Field.Name$;]
```

*Remark*: In case the `Field` belongs to a `Structure` the visibility should be `public`.

#### 6.4.3.3 Property

A `Property` is mapped as one or two ISO/ANSI C++ methods, one for a possible setter and one for a possible getter.

- The `Visibility` is defined.
- If the `Property` is of type readWrite or readOnly, a getter method is defined.
- If the `Property` is of type readWrite or writeOnly, a setter method is defined.
- If the `Property` belongs to an `Interface`, it is declared as pure `virtual` ("`= 0`").
- If the `Property` is of `ReferenceType`, it is declared as a pointer (unless it has the `ByReference` attribute).

The mapping can be tailored via the following attributes:

- If the `Property` has the `Abstract` attribute, it is declared pure `virtual` ("`= 0`").
- If the `Property` has the `Virtual` attribute, it is declared `virtual` in order to allow overwriting it.
- If the `Property` has the `Static` attribute, it is declared `static`.
- If the `Property` has the `ByReference` attribute, it is declared as a reference ("&").
- If the `Property` has the `Const` attribute, the property getter method is defined as `const`.

The `View` attribute is only taken into account in the publication of the property.

**Table 11 – Property Type Modifier depending on type and attribute**

| Attribute | none | ByReference | | |
|---|---|---|---|---|
| ValueType | | & | | |
| ValueReference | | & | | |
| ReferenceType | * | & | | |
| NativeType | | & | | |

```
// Set Visibility
public:|protected:|private:

// In case the Property is ReadOnly or ReadWrite:
[virtual] [static] TypeName($Property.Type$)[*|&] get_$Property.Name$()[ const][ = 0];

// In case the Property is WriteOnly or ReadWrite:
[virtual] [static] void set_$Property.Name$(TypeName($Property.Type$)[*|&] val)[ = 0];
```

***Remark***: A `Property` that is `Static` must neither by `Virtual` nor `Abstract`, as a `static` method cannot be `virtual`.

***Remark***: In case the `Property` belongs to an `Interface` the visibility should be `public`.

***Remark***: In case a `Property` has an attached field, a code generator may generate an implementation giving access to this field.

```
// In case the Property is ReadOnly or ReadWrite:
TypeName($Property.Type$)[*|&] TypeName($Property.Owner$)::get_$Property.Name$()[ const]
{
   return $Property.AttachedField.Name$;
}

// In case the Property is WriteOnly or ReadWrite:
void TypeName($Property.Owner$)::set_$Property.Name$(TypeName($Property.Type$)[*|&] val)
{
   $Property.AttachedField.Name$ = val;
}
```

#### 6.4.3.4 Association

An `Association` is mapped to C++ as a member variable of a C++ `class`.

- The `Visibility` is defined.
- If the `Association` is of a `ValueType` or `ReferenceType`, it is defined as a pointer (“*”).

The mapping can be tailored via the following attributes:

- If the `Association` has the `Static` attribute, it is declared `static`.
- If the `Association` has the `ByReference` attribute, it is defined as a reference (“&”).
- If the `Association` has the `Const` attribute, it is defined as `const`.

**Table 12 – Association Type Modifier depending on type and attribute**

| Type | ValueType | ValueReference | ReferenceType | NativeType |
|---|---|---|---|---|
| **Default** | * | | * | |
| **ByReference** | & | & | & | & |

```
// Set Visibility
public:|protected:|private:

// Association of type reference.
[const] [static] TypeName($Association.Type$)[*|&] $Association.Name$;
```

#### 6.4.3.5 Operation

An `Operation` is mapped to an ISO/ANSI C++ method.

- The `Visibility` is defined.
- If the `Operation` belongs to an `Interface`, it is declared as pure `virtual` (“`= 0`”).

- If the `Operation` has a `Parameter` of `return` direction, this parameter defines the return type of the operation (called `$Operation.ReturnType$` below).
- If the `Operation` returns a `ReferenceType`, it returns a pointer ("*"), otherwise it returns a value.

The mapping can be tailored via the following attributes:

- If the `Operation` has the `Abstract` attribute, it is declared pure `virtual` ("`= 0`").
- If the `Operation` has the `Static` attribute, it is declared `static`.
- If the `Operation` has a `return Parameter` with the `ByReference` attribute, it returns a reference ("&").
- If the `Operation` has the `Virtual` attribute, it is declared `virtual` in order to allow overwriting it.
- If the `Operation` has the `Const` attribute, it is defined as `const`.

The `View` attribute is only taken into account in the publication of the operation.

```
// Set Visibility
public:|protected:|private:

// Return type of Operation is defined by the return parameter, or void
[virtual] [static] void|TypeName($Operation.ReturnType$)[*|&] $Operation.Name$(
   // Comma-separated mapping of Parameters, except for return Parameter
   See 6.4.3.6 (Parameter)
)[ const][ = 0];
```

***Remark***: In case the `Operation` belongs to an `Interface` the visibility should be `public`.

- If the `Operation` has the `Operator` attribute, it is declared as an `operator`.

```
// Set Visibility
public:|protected:|private:

// Return type of Operator is defined by the return parameter, or void
[virtual] [static] void|TypeName($Operation.ReturnType$)[*|&] operator $Operator.Kind$(
   // Comma-separated mapping of Parameters, except for return Parameter
   See 6.4.3.6 (Parameter)
)[ const][ = 0];
```

- If the `Operation` has the `Constructor` attribute, it is declared as a constructor.

```
// Set Visibility
public:|protected:|private:

// Constructor must neither be const, nor virtual, nor have a return Parameter
$Operation.Owner.Name$(Parameters);
```

***Remark***: In case the `Operation` has the `Constructor` attribute, it should not have the `Const`, `Virtual` or `Static` attribute, or a `Parameter` of `return` direction.

#### 6.4.3.6 Parameter

A `Parameter` is mapped to an argument of a C++ operation, operator or constructor, or to the return type of an operation or operator.

- An empty parameter list (i.e. at most a return parameter) is mapped to `void`.
- A non-empty parameter list is mapped to a comma-separated list of arguments with optional type modifier, type and name for each argument (but excluding an optional return parameter).
- Based on `Type` and `Direction`, the `Parameter` is passed by value, as a pointer, or as a reference, according to the following table:

**Table 13 - Parameter Modifier depending on type and direction**

| Direction | in | out | inout | return |
|---|---|---|---|---|
| ValueType | const | * | * | |
| ValueReference | const | | | |
| ReferenceType | const & | * | * | * |
| NativeType | const | * | * | |

The mapping can be tailored via the following attributes:

- If the `Parameter` has the `ByReference` attribute, it is passed by reference ("&").
- If the `Parameter` has the `Const` attribute, it is passed as `const`.

```
// Parameter can be passed by value, by reference, or as a pointer
[const] TypeName($Parameter.Type$)[*|&] $Parameter.Name$
```

### 6.4.4 Values

Default values of fields, and values of constants are mapped to C++ using the appropriate mechanisms to initialise fields in a class or structure.

#### 6.4.4.1 Value

This base class is not mapped to C++.

#### 6.4.4.2 Simple Value

A `SimpleValue` maps to a corresponding value in C++. It is always used together with an owner, such as a `Constant`, `Field`, or `Parameter`.

##### 6.4.4.2.1 Bool Value

```
Smp::Bool $BoolValue.Owner.Name$ = false|true;
```

##### 6.4.4.2.2 Char8Value

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

```
Smp::Char8 $Char8Value.Owner.Name$ = $Char8Value.Value$;
```

##### 6.4.4.2.3 Date Time Value

```
Smp::DateTime $BoolValue.Owner.Name$ = $DateTimeValue.Value$;
```

#### 6.4.4.2.4 Duration Value

```
Smp::Duration $DurationValue.Owner.Name$ = $DurationValue.Value$;
```

#### 6.4.4.2.5 Enumeration Value

```
TypeName($EnumerationValue.Owner.Type) $EnumerationValue.Owner.Name$
    = $EnumerationValue.Literal$;
```

An `EnumerationValue` maps to the enumeration `Literal` of the `Enumeration-Value`, not to the `Value`.

#### 6.4.4.2.6 Float32Value

```
Smp::Float32 $Float32Value.Owner.Name$ = $Float32Value.Value$;
```

#### 6.4.4.2.7 Float64Value

```
Smp::Float64 $Float64Value.Owner.Name$ = $Float64Value.Value$;
```

#### 6.4.4.2.8 Int16Value

```
Smp::Int16 $Int16Value.Owner.Name$ = $Int16Value.Value$;
```

#### 6.4.4.2.9 Int32Value

```
Smp::Int32 $Int32Value.Owner.Name$ = $Int32Value.Value$;
```

#### 6.4.4.2.10 Int64Value

```
Smp::Int64 $Int64Value.Owner.Name$ = $Int64Value.Value$;
```

#### 6.4.4.2.11 Int8Value

```
Smp::Int8 $Int8Value.Owner.Name$ = $Int8Value.Value$;
```

#### 6.4.4.2.12 String8Value

```
Smp::String8 $String8Value.Owner.Name$ = $String8Value.Value$;
```

#### 6.4.4.2.13 UInt16Value

```
Smp::UInt16 $UInt16Value.Owner.Name$ = $UInt16Value.Value$;
```

#### 6.4.4.2.14 UInt32Value

```
Smp::UInt32 $UInt32Value.Owner.Name$ = $UInt32Value.Value$;
```

#### 6.4.4.2.15 UInt64Value

```
Smp::UInt64 $UInt64Value.Owner.Name$ = $UInt64Value.Value$;
```

#### 6.4.4.2.16 UInt8Value

```
Smp::UInt8 $UInt8Value.Owner.Name$ = $UInt8Value.Value$;
```

**Examples:**

```
Smp::Int32 myInt32Field = 123;
Smp::Char8 myChar8Field = 'x';
```

### 6.4.4.3 Simple Array Value

A `SimpleArrayValue` maps to an array of simple values.

**Example:**

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

```
MyPosition myArrayField = {{1.0, 2.0, 3.0}};
```

#### 6.4.4.4 Array Value

An `ArrayValue` maps to an array of values.

#### 6.4.4.5 Structure Value

A `StructureValue` maps to a value of a structure, which is a comma-separated list of field values enclosed in "`{`" and "`}`".

**Example:**

```
struct MyStruct
{
    Smp::Int64 size;
    Smp::Char8 text;
};

MyStruct myStructField = { 1, 'x' };
```

### 6.4.5 Attributes

An `Attribute` is not mapped to C++.

## 6.5 Smdl Catalogue

### 6.5.1 Catalogue

#### 6.5.1.1 Catalogue

A `Catalogue` itself is not mapped to C++, but its name may be used as the name of a root directory for files generated for namespaces in the catalogue.

#### 6.5.1.2 Namespace

A `Namespace` is mapped to C++ `namespace`. As no code is generated for a namespace, every type contained within a namespace will be wrapped by a C++ `namespace` definition. As namespaces may contain nested namespaces, a C++ `namespace` may contain further namespaces.

```
namespace $Namespace.Name$
{
  // Define nested namespaces and types
}
```

***Remark***: To avoid problems with circular includes of files for the definition of composed types, it may be necessary to add a forward declaration of `Structure`, `Class`, `Interface` and `Model`. Further, it is recommended to define each `Structure`, `Class`, `Interface` or `Model` in a dedicated header file, and to provide the implementation of a `Structure`, `Class` or `Model` in a dedicated source file.

### 6.5.2 Reference Types

#### 6.5.2.1 Reference Type

`ReferenceType` is an abstract base type. Each type derived from `ReferenceType` is mapped to a C++ `class`.

#### 6.5.2.2 Component

`Component` is an abstract base type. Each type derived from `Component` is mapped to a C++ `class`.

#### 6.5.2.3 Interface

An `Interface` is mapped to an abstract C++ `class`, which means that every C++ method for any `Operation` or `Property` of the `Interface` is declared pure `virtual` ("`= 0`").

- An `Interface` always has a virtual destructor with an empty implementation.
- If the `Interface` has base interfaces, `virtual` inheritance is used.

```
// In case the Interface has Base Interfaces these will be inherited virtual and public.
class $Interface.Name$
  [: virtual public TypeName($Interface.Base1$), ...]
{
  // Virtual destructor is always generated
  virtual ~$Interface.Name$(void) {}
```

```
  // If the Interface has Properties, these are declared pure virtual:
  See 6.4.3.3 (Property)

  // If the Interface has Operations, these are declared pure virtual:
  See 6.4.3.5 (Operation)
};
```

#### 6.5.2.4 Model

A `Model` is mapped to a C++ `class`. For basic SMP compliance, the model is mapped as follows:

- The `Model` implements the `IModel` interface.
- If the `Model` has at least one `Container`, it implements the `IComposite` interface.
- If the `Model` provides an `Interface`, it implements this interfaces via public, virtual inheritance.
- If the `Model` has a `Base` model, it inherits its implementation via public, non-virtual inheritance.

The mapping can be tailored via the following attributes:

- If the `Model` has a `BaseClass` attribute, it inherits from the specified base class.
- If the `Model` has a `Fallible` attribute, it implements the `IFallibleModel` interface.

```
class $Model.Name$ :
  [public $BaseClass.Name$,]
  [public TypeName($Model.Base$),]
  [virtual public TypeName($Model.Interface1$), ...]
  [virtual public Smp::IComposite,]
   virtual public Smp::IModel|Smp::IFallibleModel
{
  // In case the Model has nested types these are defined:
  See 6.4.1 (Types)

  // Constructor and virtual Destructor are declared with Visibility of the Model
  public:|protected:|private:

  // Default Constructor is always generated.
  $Model.Name$(void);

  // Virtual Destructor is always generated.
  virtual ~$Model.Name$(void);

  // If the Model has Constants, these are declared:
  See 6.4.3.1 (Constant)

  // If the Model has Fields, these are declared:
  See 6.4.3.2 (Field)

  // If the Model has Properties, these are declared virtual:
  See 6.4.3.3 (Property)

  // If the Model has associations, these are declared:
  See 6.4.3.4 (Association)
```

```
   // If the Model has Operations, these are declared virtual:
   See 6.4.3.5 (Operation)

   // If the Model has Entry Points, they will be declared:
   See 6.5.3.1.1 (Entry Point)

   // If the Model has Containers, they will be declared:
   See 6.5.3.2.1 (Container)

   // If the Model has References, they will be declared:
   See 6.5.3.2.2 (Reference)

   // If the Model has Event Sources, they will be declared:
   See 6.5.3.3.2 (Event Source)

   // If the Model has Event Sinks, they will be declared:
   See 6.5.3.3.3 (Event Sink)
};
```

Each of the interfaces that the Model has to implement could be implemented by inheritance from another class that implements the interface.
For full SMP compliance, the following holds in addition:

- The `Model` implements the `IManagedModel` interface.
- If the `Model` has at least one `Reference`, it implements the `IAggregate` interface.
- If the `Model` has at least one `EventSource`, it implements the `IEventProvider` Interface.
- If the `Model` has at least one `EventSink`, it implements the `IEventConsumer` Interface.
- If the `Model` has at least one `EntryPoint`, it implements the `IEntryPointPublisher` Interface.

```
// If the Model has a Base Model it uses public (non-virtual) implementation inheritance.
// If the Model provides Interfaces, it uses public, virtual inheritance.
class $Model.Name$ :
  [public $BaseClass.Name$,]
  [public TypeName($Model.Base$),]
  [virtual public TypeName($Model.Interface1$), ...]
  [virtual public Smp::IComposite,]
  [virtual public Smp::IAggregate,]
  [virtual public Smp::Management::IEventProvider,]
  [virtual public Smp::Management::IEventConsumer,]
  [virtual public Smp::Management::IEntryPointPublisher,]
  [virtual public Smp::IFallibleModel,]
   virtual public Smp::Management::IManagedModel
{
   ...
}
```

#### 6.5.2.5 Service

A `Service` is mapped to a C++ `class`.

- The `Service` implements the `IService` interface.
- If the `Service` provides an `Interface`, it implements this interfaces.
- If the `Service` has a `Base` component, it inherits its implementation.

The mapping can be tailored via the following attributes:

- If the `Service` has a `BaseClass` attribute, it inherits from the specified base class.

```
// If the Service has a Base Component it uses public (non-virtual) inheritance.
// If the Service provides Interfaces, it uses public, virtual inheritance.
class $Service.Name$ :
  [public $BaseClass.Name$,]
  [public TypeName($Service.Base$),]
  [virtual public TypeName($Service.Interface1$), ...]
   virtual public Smp::IService
{
   // In case the Service has nested types these are defined:
   See 6.4.1 (Types)

   // Constructor and virtual Destructor are declared with Visibility of the Service
   public:|protected:|private:

   // Default Constructor is always generated.
   $Service.Name$(void);

   // Virtual Destructor is always generated.
   virtual ~$Service.Name$(void);

   // If the Service has Constants, these are declared:
   See 6.4.3.1 (Constant)

   // If the Service has Fields, these are declared:
   See 6.4.3.2 (Field)

   // If the Service has Properties, these are declared virtual:
   See 6.4.3.3 (Property)

   // If the Service has Associations, these are declared:
   See 6.4.3.4 (Association)

   // If the Service has Operations, these are declared virtual:
   See 6.4.3.5 (Operation)

   // If the Service has Entry Points, they will be declared:
   See 6.5.3.1.1 (Entry Point)

   // If the Service has Event Sources, they will be declared:
   See 6.5.3.3.2 (Event Source)

   // If the Service has Event Sinks, they will be declared:
   See 6.5.3.3.3 (Event Sink)
};
```

***Remark***: Each of the interfaces that the Service has to implement could be implemented by inheritance from another class that implements the interface.

### 6.5.3 Modelling Mechanisms

#### 6.5.3.1 Class based Design

##### 6.5.3.1.1 Entry Point

An `EntryPoint` is mapped to a reference to the `IEntryPoint` interface. The `EntryPoint` is mapped `public`.

```
public:
  // Expose entry point as pointer to IEntryPoint interface
  Smp::IEntryPoint* $EntryPoint.Name$;
```

The model has to ensure that an implementation is available when other components call the handler of the entry point.

***Remark***: An entry point can as well be mapped to a type that implements the `Smp::IEntryPoint` interface.

#### 6.5.3.2 Component based Design

##### 6.5.3.2.1 Container

A `Container` is mapped to a pointer to the `IContainer` interface. The `Container` is mapped `public`.

```
public:
  // Expose container as pointer to IContainer interface
  Smp::IContainer* $Container.Name$;
```

The model has to ensure that an implementation is available when other components access the container.

***Remark***: A container can as well be mapped to a type that implements the `Smp::IContainer` interface.

##### 6.5.3.2.2 Reference

A `Reference` is mapped to a pointer to the `IReference` interface. The `Reference` is mapped `public`.

```
public:
  // Expose reference as pointer to IReference interface
  Smp::IReference* $Reference.Name$;
```

The model has to ensure that an implementation is available when other components access the reference.

***Remark***: A reference can as well be mapped to a type that implements the `Smp::IReference` interface.

#### 6.5.3.3 Event based Design

##### 6.5.3.3.1 Event Type

An event type itself is not mapped to C++, as the event notification mechanism uses an argument of `AnySimple` type to pass a value with each event.

***Remark***: The event type can be used to ensure that event sinks and event sources are only connected if they are of the same event type.

#### 6.5.3.3.2 Event Source

An `EventSource` is mapped to a pointer to the `IEventSource` interface. The `EventSource` is mapped public.

```
public:
  // Expose event source as pointer to IEventSource interface
  Smp::IEventSource* $EventSource.Name$;
```

The model has to ensure that an implementation is available when other components access the event source.

***Remark***: An event source can as well be mapped to a type that implements the `Smp::IEventSource` interface.

#### 6.5.3.3.3 Event Sink

An `EventSink` is mapped to a pointer to the `IEventSink` interface. The `EventSink` is mapped public.

```
public:
  // Expose event sink as pointer to IEventSink interface
  Smp::IEventSink* $EventSink.Name$;
```

The model has to ensure that an implementation is available when other components access the event sink.

***Remark***: An event sink can as well be mapped to a type that implements the `Smp::IEventSink` interface.

### 6.5.4 Catalogue Attributes

#### 6.5.4.1 View and ViewKind

The `View` attribute, together with the `ViewKind` enumeration, is used to specify the view argument for the publication call of a `Field`, `Property` or `Operation`. In case that a field, property or operation is published, this publication call shall use the value assigned to the `View` attribute, or `VK_None` as default.

## 6.6 Smdl Package

### 6.6.1 Package

A package describes all metamodel elements that are needed in order to define how implementations of types defined in catalogues are packaged.

#### 6.6.1.1 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.

The C++ mapping of a `Package` is a static or dynamic library providing an implementation for each of the `Implementation` elements. To make these types available for later use, many of them need to be registered: For each model, a factory needs to be registered, while value types are registered in the type registry. This registration of types shall be done from a standardised `Initialise()` function. In a corresponding `Finalise()` function, memory has to be released. To avoid duplicate symbols in the linker, these functions shall contain the name of the package as well.

```
extern "C"
{
   /// Initialise function for static package.
   bool Initialise$Package.Name$(
      Smp::ISimulator* simulator,
      Smp::Publication::ITypeRegistry* typeRegistry);

   /// Finalise function for static package.
   bool Finalise$Package.Name$();
}
```

For a dynamic library (Dynamic Link Library (**DLL**) on the Microsoft Windows OS, or Dynamic Shared Object (**DSO**) on the Unix OS), two additional functions that do not include the name of the package shall be defined as well, as global functions in dynamic libraries do not create naming conflicts at link time. These `Initialise()` and corresponding `Finalise()` functions shall call the functions including the package name. As a package may reference other packages as a `Dependency`, which indicates that a type referenced from an implementation in the package requires a type implemented in the referenced package, the initialise and finalise functions of these dependencies shall be called as well.

```
#ifdef  WIN32
#define DLL_EXPORT __declspec(dllexport)
#else
#define DLL_EXPORT
#endif

extern "C"
{
```

```
    /// Initialise function for dynamic package.
    DLL_EXPORT bool Initialise(
        Smp::ISimulator* simulator,
        Smp::Publication::ITypeRegistry* typeRegistry);

    /// Finalise function for dynamic package.
    DLL_EXPORT bool Finalise();
}
```

There are no rules on the order in which packages are initialised, as the type registration process via Universally Unique Identifiers (UUIDs) does not introduce dependencies on the order. However, the initialise and finalise functions may get called several times during initialisation (e.g. when referenced from more than one package), so the implementation needs to ensure that types are only registered once, and memory is released only once as well.

#### 6.6.1.2 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.

**Implementation of a Model**

When the `Implementation` points to a `Model` (via its `Type` link), a corresponding class factory has to be registered with the managed simulator (`IManagedSimulator`) using the `RegisterFactory()` method. This class factory uses the `Uuid` of the `Model` (as specification identifier) as well as the `Uuid` of the `Implementation` (as implementation identifier). This allows registering more than one implementation for a `Model` definition in a `Catalogue`.

This will only work when the simulator passed to the `Initialise()` function is a managed simulator, i.e. it implements the optional `IManagedSimulator` interface.

**Implementation of a Service**

When the `Implementation` points to a `Service` (via its `Type` link), an instance of the `Service` shall be created and added to the simulator (`ISimulator`) using the `AddService()` method.

**Implementation of a Value Type**

When the `Implementation` points to a `ValueType` (via its `Type` link), the corresponding user-defined value type has to be registered in the type registry (`ITypeRegistry`). This is done by calling the global register function (for `Enumeration`, `Integer`, `Float`, `Array`, `String`) or method (`Structure`, `Class`, `Exception`) of the type.

### 6.6.2 Bundle Format

For distribution of a binary package, this C++ Mapping defines a C++ specific Bundle Format. A Bundle is an archive (e.g. a tar file on Linux, or a zip file on Windows) which can provide any number of folders and files that make up the

bundle. The structure of folders and files within the bundle, and the names of folders and files are not standardised, and can include both SMP artefacts (Smdl Catalogues, Assemblies, Schedules, Packages and Configurations) and other files not standardised by SMP (including, but not limited to, source code files, libraries, and documentation).

The added value of a Bundle is an additional Bundle Manifest with the following name:

```
SMP.MF
```

This Manifest is an ASCII file (aligned with the OSGi manifest format) which contains key-value pairs in the following format:

```
Key: Value
```

- `Key` and `Value` are separated by a colon.
- They `Key` must only contain alpha-numerical characters, underscore ("_") and dash ("-").
- The `Value` starts at the first non-whitespace character after the colon (":"), and is terminated by the end of line

All valid Keys are allowed, but a number of Keys have a defined meaning:

| Key | Meaning |
|---|---|
| Bundle-Copyright | Copyright statement for the bundle. |
| Bundle-ContactAddress | Full address of a person or company that can be contacted. |
| Bundle-DocURL | URL where documentation for the bundle can be retrieved from. |
| Bundle-Description | Textual description of the bundle and its content. |
| Bundle-ManifestVersion | A bundle manifest may express the version of the OSGi manifest header syntax in the Bundle-ManifestVersion header. If specified, the bundle manifest version must be'2'. |
| Bundle-Name | The Bundle-Name header defines a readable name for this bundle. This should be a short, human-readable name that can contain spaces. |
| Bundle-SymbolicName | The Bundle-SymbolicName manifest header is a mandatory header. The bundle symbolic name and bundle version allow a bundle to be uniquely identified in the Framework. That is, a bundle with a given symbolic name and version is treated as equal to another bundle with the same (case sensitive) symbolic name and exact version.<br>The installation of a bundle with a Bundle-SymbolicName and Bundle-Version identical to an existing bundle must fail. |
| Bundle-Vendor | The Bundle-Vendor header contains a human-readable description of the bundle vendor. |
| Bundle-Version | Bundle-Version is an optional header; the default value is 0.0.0. |

| Key | Meaning |
|---|---|
| | A version consists of major, minor and micro version components. If the minor or micro version components are not specified, they have a default value of 0.<br><br>Versions are comparable. Their comparison is done numerically and sequentially on the major, minor, and micro components. A version is considered equal to another version if the major, minor, and micro components are equal. |
| Require-Bundle | The Require-Bundle header specifies the required exports from another bundle. This is a comma-separated list of required bundles, where each bundle is at least specified by its symbolic name, optionally followed by a specific version:<br><br><Bundle-SymbolicName>[; Bundle-Version="<Bundle-Version>"] |
| Compiler-Name | Name of the compiler that has been used to compile the source code. |
| Compiler-Version | Version of the compiler that has been used to compile the source code. |
| OS-Name | Name of the Operating System. |
| OS-Version | Version of the Operating System. |

### 6.6.3 Binary Distribution

The mapping of a Package to C++ defines which symbols a static or dynamic library of SMP2 Types has to expose. This has been done to enable binary distribution of models, where only the header files (for the compiler) and the libraries (for the linker) have to be provided, but no implementation source code. Nevertheless, binary compatibility depends on a number of other constraints, which may even vary between operating systems and compilers. The following guidelines help to reduce problems, but are by no means complete:

- The operating system on which the Simulation Environment runs should be used to compile the models.
- The same C++ compiler that has been used to compile the Simulation Environment (the component providing the implementation of the ISimulator interface) should be used to compile the models.
- The compiler settings that have been used to compile the Simulation Environment (the component providing the implementation of the ISimulator interface) should be used to compile the models.

The objective of these guidelines is to ensure that resolving of symbols and Run-Time Type Information (**RTTI**) is compatible between the Simulation Environment and the models, and that both depend on the same libraries of the same operating system (e.g. same glibc library).