EUROPEAN SPACE AGENCY
DIRECTORATE OF OPERATIONS AND INFRASTRUCTURE

SMP 2.0 C++ Mapping

EGOS-SIM-GEN-TN-0102

Issue 1 Revision 2

28 October 2005

This Page is Intentionally left Blank

ABSTRACT

This document contains the mapping of both the Component Model and the Metamodel to C++ for the SMP 2.0 standard.

© EUROPEAN SPACE AGENCY, 2005

DOCUMENT APPROVAL

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

Verified by	Organisation	Signature	Date
Christine Dingeldey	VEGA		28 October 2005

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

© EUROPEAN SPACE AGENCY, 2005

DOCUMENT STATUS SHEET

1. Issue	2. Revision	3. Date	4. Reason for Change
0	Beta 1	13 April 2004	New document structure
0	Beta 2	24 May 2004	Mapping of metamodel to C++ added.
0	RC 1	06 August 2004	Mapping updated for Release Candidate 1.
1	0	13 October 2004	Initial Release of SMP 2.0 Standard.
1	1	28 February 2005	First Update of SMP 2.0 Standard.
1	**2**	**28 October 2005**	Second Update of SMP 2.0 Standard.

DOCUMENT CHANGE RECORD

<table>
<tr><td colspan="3" rowspan="4">DOCUMENT CHANGE RECORD
Changes from
SMP 2.0 C++ Mapping Issue 1 Revision 1 to
SMP 2.0 C++ Mapping Issue 1 Revision 2</td><td>DCI NO</td><td>N/A</td></tr>
<tr><td>DATE</td><td>28 October 2005</td></tr>
<tr><td>ORIGINATOR</td><td>SMP CCB</td></tr>
<tr><td>APPROVED BY</td><td>Niklas Lindman</td></tr>
<tr><td>1. PAGE</td><td>2. PARAGRAPH</td><td>3. ISSUE</td><td colspan="2">4. CHANGES MADE</td></tr>
<tr><td>14</td><td>1.4</td><td>46</td><td colspan="2">SMP Handbook and Alpha Specification moved from Applicable Documents to Reference Documents.</td></tr>
<tr><td>15</td><td>2.2.1</td><td>8</td><td colspan="2">SGI/IRIX added.</td></tr>
<tr><td>21</td><td>3</td><td>See AD-3</td><td colspan="2">Updated according to changes of Component Model (see [AD-3]).</td></tr>
<tr><td>21</td><td>3.1</td><td>30</td><td colspan="2">Inheritance of exceptions from `Smp::Exception` added.</td></tr>
<tr><td>49</td><td>4</td><td>See AD-3</td><td colspan="2">Updated according to changes of Simulation Services (see [AD-3]).</td></tr>
<tr><td>57</td><td>5</td><td>See [AD-2]</td><td colspan="2">Updated according to changes of Metamodel (see [AD-2]).</td></tr>
<tr><td>60</td><td>5.3.2</td><td>4</td><td colspan="2">Static function/method for registration of user-defined value types added.</td></tr>
<tr><td>62</td><td>5.3.3</td><td>14</td><td colspan="2">`Feature` and `static` removed.</td></tr>
<tr><td>65</td><td>5.4.2.1</td><td>4</td><td colspan="2">Static method for registration of user-defined class added.</td></tr>
<tr><td>70</td><td>5.5</td><td></td><td colspan="2">`Package` added to Metamodel, and dedicated section 7 replaced.</td></tr>
<tr><td>70</td><td>5.5.1.1</td><td>5</td><td colspan="2">`Initialize()` method replaced by `Initialise$Package.Name$`, with additional `Initialise()` for shared object.</td></tr>
<tr><td>73</td><td>6</td><td>32</td><td colspan="2">Publication of `Property` added.</td></tr>
<tr><td>73</td><td>6.1</td><td>-3</td><td colspan="2">Registration of types changed to use `Uuid`.</td></tr>
<tr><td>81</td><td>6.1.5.5</td><td>31</td><td colspan="2">Additional `type` parameter for primitive type added.</td></tr>
<tr><td>82</td><td>6.1.5.6</td><td>31</td><td colspan="2">Additional `type` parameter for primitive type added.</td></tr>
<tr><td>86</td><td>6.2</td><td>-3</td><td colspan="2">Publication changed to use `Uuid`.</td></tr>
<tr><td>88</td><td>6.2.1.1.4</td><td>32</td><td colspan="2">Publication of `Property` added.</td></tr>
</table>

© EUROPEAN SPACE AGENCY, 2005

TABLE OF CONTENTS

ABSTRACT **3**

1. INTRODUCTION **13**
1.1 Purpose 13
1.2 Scope 13
1.3 Definitions, acronyms and abbreviations 13
1.4 References 14
1.4.1 Applicable Documents 14
1.4.2 Reference Documents 14
1.5 Overview 14

2. PLATFORM CONSIDERATIONS **15**
2.1 ANSI/ISO C++ 15
2.2 Data Types 15
2.2.1 Simple Types 15
2.2.2 Simple Union Type 16
2.2.2.1 SimpleTypeKind 16
2.2.2.2 SimpleTypeValue 16
2.2.2.3 AnySimple 17
2.2.2.4 AnySimpleArray 17
2.2.3 Universally Unique Identifiers 17
2.2.4 Strings 17
2.2.5 Collections 18
2.3 Interfaces 18
2.4 Inheritance 18
2.5 Deployment 18
2.6 Performance and Tradeoffs 18

3. COMPONENT MODEL **21**
3.1 Exceptions 21
3.1.1 InvalidObjectName 22
3.1.2 DuplicateName 22
3.1.3 InvalidAnyType 23
3.1.4 InvalidObjectType 23
3.2 Objects and Components 24
3.2.1 Objects 24
3.2.1.1 IObject 24
3.2.2 Components 24
3.2.2.1 IComponent 24
3.2.2.2 ComponentCollection 25
3.2.2.3 ModelStateKind 25
3.2.2.4 InvalidModelState 25
3.2.2.5 IModel 26
3.2.2.6 ModelCollection 26
3.2.2.7 IService 26
3.2.2.8 ServiceCollection 26
3.3 Component Mechanisms 27
3.3.1 Aggregation 27
3.3.1.1 IAggregate 27
3.3.1.2 IReference 27
3.3.1.3 ReferenceCollection 27
3.3.2 Composition 28
3.3.2.1 IComposite 28
3.3.2.2 IContainer 28
3.3.2.3 ContainerCollection 28
3.3.3 Events 29

3.3.3.1 IEventSink.....29
3.3.3.2 EventSinkCollection.....29
3.3.3.3 IEventSource.....30
3.3.3.4 EventSourceCollection.....31
3.3.4 Dynamic Invocation.....31
3.3.4.1 IDynamicInvocation.....31
3.3.4.2 IRequest.....33
3.3.5 Persistence.....34
3.3.5.1 IPersist.....35
3.3.5.2 IStorageReader.....35
3.3.5.2.1 Restore.....36
3.3.5.3 IStorageWriter.....36
3.3.5.3.1 Store.....36
3.4 Model Mechanisms.....37
3.4.1 Entry Points.....37
3.4.1.1 IEntryPoint.....37
3.4.1.2 EntryPointCollection.....37
3.5 Management Interfaces.....38
3.5.1 Managed Components.....38
3.5.1.1 IManagedObject.....38
3.5.1.2 IManagedComponent.....38
3.5.2 Managed Component Mechanisms.....39
3.5.2.1 IManagedReference.....39
3.5.2.2 IManagedContainer.....40
3.5.2.3 IEventConsumer.....40
3.5.2.4 IEventProvider.....41
3.5.3 Managed Model Mechanisms.....41
3.5.3.1 IManagedModel.....41
3.5.3.2 IEntryPointPublisher.....43
3.6 Simulation Environments.....43
3.6.1 Simulators.....43
3.6.1.1 SimulatorStateKind.....44
3.6.1.2 ISimulator.....45
3.6.1.3 IDynamicSimulator.....46
3.6.1.4 IFactory.....46
3.6.1.5 FactoryCollection.....47
3.6.2 Publication.....47

4. SIMULATION SERVICES.....49
4.1 Mandatory Services.....49
4.1.1 Logger.....49
4.1.1.1 ILogger.....49
4.1.1.2 Predefined Log Message Kinds.....50
4.1.2 Time Keeper.....50
4.1.2.1 ITimeKeeper.....50
4.1.2.2 TimeKind.....51
4.1.3 Scheduler.....52
4.1.3.1 IScheduler.....52
4.1.3.2 ITask.....53
4.1.4 Event Manager.....53
4.1.4.1 IEventManager.....53
4.1.4.2 Predefined Event Kinds.....55
4.2 Optional Services.....56
4.2.1 Resolver.....56
4.2.1.1 IResolver.....56

5. METAMODEL.....57
5.1 Overview.....57
5.1.1 Placeholders.....57

 © EUROPEAN SPACE AGENCY, 2005

5.1.2 Coloring and Font Schema 57
5.1.3 Generation of type Identification 57
5.1.4 Optional and Selectable Code 57
5.2 Core Elements 58
5.2.1 Simple Types 58
5.2.1.1 Identifier 58
5.2.1.2 Name 58
5.2.1.3 Description 58
5.2.1.4 UUID 58
5.2.2 XML Links 58
5.2.3 Elements 58
5.2.3.1 Element 58
5.2.3.2 Named Element 58
5.2.3.3 Document 58
5.2.4 Metadata 58
5.3 Core Types 59
5.3.1 Types 59
5.3.1.1 Visibility Element 59
5.3.1.2 Type 59
5.3.1.3 Language Type 59
5.3.1.4 Value Type 59
5.3.1.5 Value Reference 59
5.3.2 Value Types 60
5.3.2.1 Primitive Type 60
5.3.2.2 Enumeration 60
5.3.2.3 Integer 60
5.3.2.4 Float 60
5.3.2.5 Array 61
5.3.2.6 String 61
5.3.2.7 Structure 62
5.3.3 Typed Elements 62
5.3.3.1 Field 62
5.3.3.2 Operation 62
5.3.3.2.1 Parameter 63
5.3.4 Values 63
5.3.4.1 Value 63
5.3.4.2 Simple Value 63
5.3.4.3 Array Value 64
5.3.4.4 String Value 64
5.3.4.5 Structure Value 64
5.3.4.5.1 Field Value 64
5.3.5 Attributes 64
5.4 SMDL Catalogues 65
5.4.1 A Catalogue Document 65
5.4.1.1 Catalogue 65
5.4.1.2 Namespace 65
5.4.2 Classes 65
5.4.2.1 Class 65
5.4.2.2 Property 66
5.4.2.3 Association 66
5.4.3 Reference Types 67
5.4.3.1 Reference Type 67
5.4.3.2 Interface 67
5.4.3.3 Model 67
5.4.3.3.1 Entry Point 69
5.4.3.3.2 Container 69
5.4.3.3.3 Reference Collection 69
5.4.4 Events 69
5.4.4.1 Event Type 69

5.4.4.2 Event Source 69
5.4.4.3 Event Sink 69
5.5 SMDL Packages 70
5.5.1 A Package Document 70
5.5.1.1 Package 70
5.5.1.2 Implementation 71

6. PUBLICATION 73
6.1 Type Registry 73
6.1.1 IType 74
6.1.1.1 GetSimpleType 75
6.1.1.2 GetUuid 75
6.1.1.3 Publish 75
6.1.2 IEnumerationType 76
6.1.2.1 AddLiteral 77
6.1.3 IStructureType 77
6.1.3.1 AddField 78
6.1.4 IClassType 79
6.1.5 ITypeRegistry 79
6.1.5.1 AlreadyRegistered 79
6.1.5.2 NotRegistered 79
6.1.5.3 GetType for SimpleTypeKind 81
6.1.5.4 GetType for Uuid 81
6.1.5.5 AddFloat 81
6.1.5.6 AddInteger 82
6.1.5.7 AddEnumeration 83
6.1.5.8 AddArray 83
6.1.5.9 AddString 84
6.1.5.10 AddStructure 84
6.1.5.11 AddClass 85
6.1.6 Pre-defined Simple Types 85
6.2 Publication of Fields, Operations and Properties 86
6.2.1 IPublication 86
6.2.1.1 Publication using the Type Registry 86
6.2.1.1.1 GetTypeRegistry 87
6.2.1.1.2 Publish Field 87
6.2.1.1.3 Publish Operation 88
6.2.1.1.4 Publish Property 88
6.2.1.2 Direct Publication 89
6.2.1.2.1 Publish Array of simple type 89
6.2.1.2.2 Publish Array of user-defined type 90
6.2.1.2.3 Publish Structure 90
6.2.1.3 Overloaded Methods for fields of simple types 91
6.2.1.3.1 Publish Field of Char8 type 93
6.2.1.4 Convenience Methods 94
6.2.2 IPublishOperation 95
6.2.2.1 Publish Parameter 95

 © EUROPEAN SPACE AGENCY, 2005

LIST OF FIGURES AND TABLES

Figure 1-1: Three Parts of the SMP2 Platform Independent Model 14
Figure 3-1: Simulation Environment State Diagram with State Transition Methods 44
Figure 6-1: IType Interface 74
Figure 6-2: IEnumerationType Interface 76
Figure 6-3: IStructureType Interface 78

Table 5-1: Mapping of the Visibility attribute to ISO/ANSI C++ Access Control 59
Table 5-2: Type Modifier depending on type and direction 63

This Page is Intentionally left Blank

© EUROPEAN SPACE AGENCY, 2005

1. INTRODUCTION

This document presents the mapping of the SMP2 Platform Independent Model (**PIM**) into a Platform Specific Model (**PSM**) for the ISO/ANSI C++ platform as defined by the International Organization for Standardization (**ISO**) and the American National Standards Institute (**ANSI**).

1.1 Purpose

This document unambiguously specifies how the mechanism introduced in the SMP 2.0 Component Model (including Simulation Services) [AD-3] and SMP 2.0 Metamodel [AD-2] are mapped for the ANSI C++ programming language.

1.2 Scope

For all mechanisms defined already in the platform independent model, this document only provides a mapping to C++. For those features marked as platform specific in the PIM, a C++ version is presented and explained in detail.

1.3 Definitions, acronyms and abbreviations

AD	Applicable Document
ANSI	American National Standards Institute
DLL	Dynamic Link Library
DSO	Dynamic Shared Object
ESOC	European Space Operations Centre
ISO	International Organization for Standardization
N/A	Not Applicable
OS	Operating System
PIM	Platform Independent Model
PSM	Platform Specific Model
RD	Reference Document
STL	Standard Template Library
TBC	To be confirmed
TBD	To be defined

© EUROPEAN SPACE AGENCY, 2005

1.4 References

1.4.1 Applicable Documents

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

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

AD-2 SMP 2.0 Metamodel
EGOS-SIM-GEN-TN-0100, Issue 1.2, 28-Oct-2005

AD-3 SMP 2.0 Component Model
EGOS-SIM-GEN-TN-0101, Issue 1.2, 28-Oct-2005

1.4.2 Reference Documents

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

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

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

1.5 Overview

This document presents a mapping of the SMP2 Component Model, Simulation Services and Metamodel into the ISO/ANSI C++ programming language.

Generally, a platform mapping consists of three main parts:

1. A platform specific implementation of the SMP2 Component Model [AD-3, section 3].

2. A platform specific implementation of the SMP2 Simulation Services [AD-3, section 4].

3. A platform specific implementation of the SMP2 Metamodel elements [AD-2].

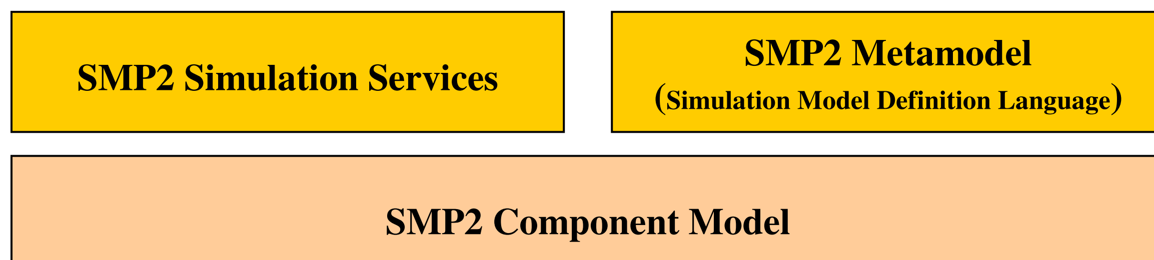

Figure 1-1: Three Parts of the SMP2 Platform Independent Model

© EUROPEAN SPACE AGENCY, 2005

2. PLATFORM CONSIDERATIONS

This section describes general considerations for the ANSI/ISO C++ platform mapping.

2.1 ANSI/ISO C++

This document describes the mapping of SMP2 to the ISO/ANSI C++ platform. In the following, we refer to ISO/ANSI C++ simply as C++. Note that, strictly speaking, C++ is not a *platform* as such, but rather a programming language. Therefore, we require certain base functionality (i.e. an implementation of the SMP2 Component Model [AD-3]) that provides an environment in which SMP2 components can live, thus turning the C++ language into a component platform.

2.2 Data Types

C++ provides most simple types needed for SMP2, except for signed and unsigned 64 bit integer types, and for an "`any`" type (a variant). Further, C++ does not define a Universally Unique Identifier (**UUID**) type.

2.2.1 Simple Types

The following type definitions are used within this document.

```
namespace Smp
{
    /// Character type that is used as well by strings.
    typedef char                Char8;      ///<  8 bit character type.
    /// Bool type that is either <code>true</code> or <code>false</code>.
    typedef bool                Bool;       ///< Bool with true and false.
    // Integer types
    typedef signed char         Int8;       ///<  8 bit   signed integer type.
    typedef unsigned char       UInt8;      ///<  8 bit unsigned integer type.
    typedef signed short        Int16;      ///< 16 bit   signed integer type.
    typedef unsigned short      UInt16;     ///< 16 bit unsigned integer type.
    typedef signed int          Int32;      ///< 32 bit   signed integer type.
    typedef unsigned int        UInt32;     ///< 32 bit unsigned integer type.
    typedef int64               Int64;      ///< 64 bit   signed integer type.
    typedef uint64              UInt64;     ///< 64 bit unsigned integer type.
    // Floating point types
    typedef float               Float32;    ///< 32 bit floating-point type.
    typedef double              Float64;    ///< 64 bit floating-point type.
    // Date and time types
    typedef Int64               Duration;   ///< Duration in Nanoseconds.
    typedef Int64               DateTime;   ///< Relative to MJD2000+0.5.
}
```

Please note that the definition of `Int64` and `UInt64` makes use of two platform specific types. Currently, these are only defined for the Gnu compiler (`gcc`) on the Linux OS, for the C++ compiler on SGI/IRIX, and for Microsoft Visual C++ on the Microsoft Windows OS.

```
#ifdef WIN32
    typedef __int64             int64;      ///< 64 bit   signed integer type.
    typedef unsigned __int64    uint64;     ///< 64 bit unsigned integer type.
#endif

#ifdef __linux__
    typedef int64_t             int64;      ///< 64 bit   signed integer type.
    typedef uint64_t            uint64;     ///< 64 bit unsigned integer type.
#endif

#ifdef __sgi
    typedef int64_t             int64;      ///< 64 bit   signed integer type.
    typedef uint64_t            uint64;     ///< 64 bit unsigned integer type.
#endif
```

© EUROPEAN SPACE AGENCY, 2005

For other operating systems (SUN Solaris, HPUX, etc), similar mappings may be needed. It is not even guaranteed that every C++ implementation provides these two data types, as they are beyond the ANSI/ISO specification.

2.2.2 Simple Union Type

Some of the interfaces defined in the SMP2 Component Model make use of the `AnySimple` type in their CORBA IDL specification. In SMP2, this data type represents a type that can hold **any** of the simple types defined above.

2.2.2.1 SimpleTypeKind

This type is an enumeration of the available simple types.

```
namespace Smp
{
    /// Enumeration of simple type kinds (discriminator for AnySimple)
    enum SimpleTypeKind
    {
        ST_None,        ///< no type, e.g. for void.
        ST_Char8,       ///<  8 bit character type.
        ST_Bool,        ///<  1 bit Bool type.
        ST_Int8,        ///<  8 bit   signed integer type.
        ST_UInt8,       ///<  8 bit unsigned integer type.
        ST_Int16,       ///< 16 bit   signed integer type.
        ST_UInt16,      ///< 16 bit unsigned integer type.
        ST_Int32,       ///< 32 bit   signed integer type.
        ST_UInt32,      ///< 32 bit unsigned integer type.
        ST_Int64,       ///< 64 bit   signed integer type.
        ST_UInt64,      ///< 64 bit unsigned integer type.
        ST_Float32,     ///< 32 bit single floating-point type.
        ST_Float64,     ///< 64 bit double floating-point type.
        ST_Duration,    ///< Duration in nanoseconds.
        ST_DateTime     ///< Point in time in nanoseconds.
    };
}
```

2.2.2.2 SimpleTypeValue

This is a union that can hold a value of any of the simple types.

```
namespace Smp
{
    /// Union of simple type values (value for AnySimple)
    union SimpleTypeValue
    {
        Char8          char8Value;      ///<  8 bit character type.
        Bool           boolValue;       ///<  1 bit Bool type.
        Int8           int8Value;       ///<  8 bit   signed integer type.
        UInt8          uInt8Value;      ///<  8 bit unsigned integer type.
        Int16          int16Value;      ///< 16 bit   signed integer type.
        UInt16         uInt16Value;     ///< 16 bit unsigned integer type.
        Int32          int32Value;      ///< 32 bit   signed integer type.
        UInt32         uInt32Value;     ///< 32 bit unsigned integer type.
        Int64          int64Value;      ///< 64 bit   signed integer type.
        UInt64         uInt64Value;     ///< 64 bit unsigned integer type.
        Float32        float32Value;    ///< 32 bit single floating-point type.
        Float64        float64Value;    ///< 64 bit double floating-point type.
        Duration       durationValue;   ///< Duration in nanoseconds.
        DateTime       dateTimeValue;   ///< Point in time in nanoseconds.
    };
}
```


© EUROPEAN SPACE AGENCY, 2005

2.2.2.3 AnySimple

This type is a discriminated union with type and value.

```
namespace Smp
{
    /// Variant of simple type values.
    /// The AnySimple type is a Discriminated Union, with type and value.
    struct AnySimple
    {
        SimpleTypeKind  type;       ///< Contained simple type
        SimpleTypeValue value;      ///< Union of values and references
    };
}
```

2.2.2.4 AnySimpleArray

This type is an array of type `AnySimple`.

```
namespace Smp
{
    /// Array of AnySimple values.
    typedef AnySimple* AnySimpleArray;
};
```

2.2.3 Universally Unique Identifiers

For a unique identification of types (and hence models), SMP2 uses Universally Unique Identifiers with the format specified by the Open Group (http://www.opengroup.org).

```
namespace Smp
{
    /// Universally Unique Identifier
    /// @remarks The 8-4-4-4-12 format as specified by the Open Group is used.
    struct Uuid
    {
        UInt32 Data1;               ///< 8 hex nibbles
        UInt16 Data2;               ///< 4 hex nibbles
        UInt16 Data3;               ///< 4 hex nibbles
        UInt8  Data4[8];            ///< 4+12 hex nibbles
    };
}
```

In C++, constants of structure types can be defined, which allows pre-defining some identifiers used later.

2.2.4 Strings

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**). As SMP2 uses only fixed strings, either as input parameters, or as constant return values of operations (`GetName()` and `GetDescription()` methods of the `IObject` base interface), all strings of the PIM are mapped to constant character pointers (`const Char8*`) in this document.

```
namespace Smp
{
    typedef const Char8* String8;   /// String8 of 8 bit characters.
}
```

2.2.5 Collections

The C++ language itself does not provide a native mechanism for collections. Therefore, the Standard Template Library (**STL**) is used for the representation of collections, namely the `std::vector` template class.

2.3 Interfaces

C++ does not have a specific language mechanism to define interfaces. Therefore, the following mapping has been used:

- An Interface is mapped to a `class`.
- All methods within an interface are declared as `public`.
- All methods within an interface are declared as `virtual`.
- All methods within an interface are declared as **abstract** (“`= 0`”).

2.4 Inheritance

As C++ does not have the concept of an interface, it cannot explicitly provide a mechanism for interface inheritance. With the mapping of interfaces to classes with public, virtual, abstract methods, the standard C++ inheritance mechanism (also used for implementation inheritance) is used. As C++ provides multiple (implementation) inheritance, this can be used to represent multiple interface inheritance. In order to avoid problems when inheriting the same interface twice (“diamond inheritance”), inheritance of interfaces is always defined as `public virtual`.

2.5 Deployment

C++ is standardised on source code level, not on binary level. Therefore, the mechanisms introduced here may not work between different compilers, i.e. when using different compilers for different components involved in a simulation. The mapping defined in this document should allow a binary distribution of models, i.e. distributing models with their header files only, but not including their implementation source code. However, this must not be misunderstood as binary compatibility, which is not even guaranteed on the same Operating System (**OS**).

The context in which components are deployed in greatly influences the requirement on object lifetime management. For example, if components are packaged in different dynamic link libraries or shared libraries, object construction and destruction need to follow stricter rules then if all components originate from the same static library or executable. For example in Microsoft Windows, objects cannot be created in one DLL and destroyed in another one. Furthermore, if components are to be deployed in a multi-process or multi-machine scenario, method and property calls need to be marshalled in one way or another, which requires the support of a middleware, like for example CORBA. Exception handling also works differently in a distributed context compared to an in-process context.

The platform mapping introduced in this document does not yet cover deployment scenarios. Typically, a model developer makes use of existing middleware (CORBA, COM, J2EE, .NET) when deploying models. The scope of this mapping is important in order to be able to deploy successfully a C++ SMP2 system. Currently, it is assumed that all components will be deployed in the *same process*. If a distributed scenario is required, then it is the responsibility of the developers to handle the distribution aspects of the deployment (e.g. component creation, marshalling, exception handling, etc.).

2.6 Performance and Tradeoffs

The platform mapping has to be designed to be applicable for real-time scenarios. In this sense, ease of use and performance are often in conflict. For example, using a flexible invocation mechanism generally

 © EUROPEAN SPACE AGENCY, 2005

requires string or character array manipulation, which can be slow compared to direct access. In the design of this mapping to C++ we have taken care of separating the data flow used at runtime during the normal execution of the system from the set-up and scripting capabilities provided.

Therefore, we assume that model to model communication will be performed through native interfaces (e.g. the satellite talks to the environment through a dedicated `IEnvironment` interface that is bound at initialisation using the linking information contained in the SMDL Assembly). Similarly, the scheduler calls methods on the models through the `Execute` method of the dedicated `IEntryPoint` interface.

On the other hand, a flexible set-up mechanism, graphical user interfaces and scripting are made possible or improved through the flexible publication and dynamic invocation mechanisms.

© EUROPEAN SPACE AGENCY, 2005

This Page is Intentionally left Blank

© EUROPEAN SPACE AGENCY, 2005

3. COMPONENT MODEL

This section details the mapping of the platform independent Component Model to ANSI/ISO C++.

3.1 Exceptions

SMP2 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.

The SMP2 Component Model has been specified using the CORBA Interface Definition Language. This language does not support inheritance of exceptions. In C++, this is possible, and as it makes exception handling much easier to do so, the C++ mapping maps SMP2 exceptions to exceptions which derive from a common `Smp::Exception` base class.

```
namespace Smp
{
    /// Base class for all SMP exceptions.
    class Exception : public std::exception, public virtual IObject
    {
    private:
        /// Name of the exception that is returned by GetName.
        String8 exceptionName;

    protected:
        /// Description of the problem encountered.
        Char8* description;

        /// Protected constructor available to derived classes only.
        Exception(String8 _exceptionName) throw() :
            exceptionName(_exceptionName),
            description(NULL)
        {
        }

        /// Virtual destructor to release memory.
        virtual ~Exception() throw()
        {
            delete description;
        }

    public:
        /// Get name of exception class.
        String8 GetName() const
        {
            return exceptionName;
        }

        /// Get description of exception.
        String8 GetDescription() const
        {
            return description;
        }
    };
}
```

This base exception provides an implementation of the `IObject` interface (see below), so that for each SMP2 exception, at least the name and a description can be queried. Further, it only provides a protected constructor with an `exceptionName`, so that it is guaranteed that all derived exceptions set the name that is returned on `GetName()`.

The `description` text that is returned by `GetDescription()` is a protected field, and can hence be modified in derived classes. All other exceptions provide a textual description of the problem encountered using this field – but the source code that creates this description is omitted from this document for some of the exceptions (only "…" is shown).

3.1.1 InvalidObjectName

```
namespace Smp
{
    static const String8 InvalidObjectNameTemplate = "Invalid object name '%s'";

    /// Invalid object name.
    class InvalidObjectName : public Exception
    {
    public:
        /// Invalid object name passed to SetName().
        String8 objectName;

        InvalidObjectName(
            String8 _objectName) throw() :
        Exception("InvalidObjectName"),
        objectName(_objectName)
        {
            description = new Char8[strlen(InvalidObjectNameTemplate)
                + strlen(objectName)];

            sprintf(description, InvalidObjectNameTemplate,
                objectName);
        }
    };
}
```

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

3.1.2 DuplicateName

```
namespace Smp
{
    static const String8 DuplicateNameTemplate =
    "Name '%s' is already used by another object";

    /// Duplicate name.
    class DuplicateName : public Exception
    {
    public:
        /// Name that already exists in the collection.
        String8 name;

        DuplicateName(
            String8 _name) throw() :
        Exception("DuplicateName"),
        name(_name)
        {
            description = new Char8[strlen(DuplicateNameTemplate)
                + strlen(name)];

            sprintf(description, DuplicateNameTemplate,
                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.

 © EUROPEAN SPACE AGENCY, 2005

3.1.3 InvalidAnyType

```
namespace Smp
{
    static const String8 InvalidAnyTypeTemplate =
    "Invalid type found: Expected '%s', but found '%s'";

    /// Invalid type of AnySimple.
    class InvalidAnyType : public Exception
    {
    public:
        /// Type that is not valid.
        Smp::SimpleTypeKind invalidType;
        /// Type that was expected.
        Smp::SimpleTypeKind expectedType;

        InvalidAnyType(
            Smp::SimpleTypeKind _invalidType,
            Smp::SimpleTypeKind _expectedType) throw() :
        Exception("InvalidAnyType"),
            invalidType(_invalidType),
            expectedType(_expectedType)
        {
            description = new Char8[strlen(InvalidAnyTypeTemplate)
                    + strlen(TypeName[expectedType])
                    + strlen(TypeName[invalidType])];

            sprintf(description, InvalidAnyTypeTemplate,
                TypeName[expectedType],
                TypeName[invalidType]);
        }
    };
}
```

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

3.1.4 InvalidObjectType

```
namespace Smp
{
    static const String8 InvalidObjectTypeTemplate =
    "Object '%s' is not of valid type.";

    /// Invalid type of an Object.
    class InvalidObjectType : public std::exception
    {
    public:
        /// Object that is not valid type.
        Smp::IObject* invalidObject;

        InvalidObjectType(
            IObject* _invalidObject) throw() :
        Exception("InvalidObjectType"),
            invalidObject(_invalidObject)
        {
            Smp::String8 objectName = "NULL";

            if (invalidObject)
            {
                objectName = invalidObject->GetName();
            }

            description = new Char8[strlen(InvalidObjectTypeTemplate)
                + strlen(objectName)];

            sprintf(description, InvalidObjectTypeTemplate,
                objectName);
        }
    };
}
```

This exception is raised when passing an object of a wrong type to a method. This exception is based on some additional semantics, e.g. a container that has been defined as a container of receivers implementing an `IReceiver` interface.

3.2 Objects and Components

In SMP2, a simulation is composed out 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 the bases for components. They provide name and description.

3.2.1.1 IObject

```
namespace Smp
{
    /// Base interface for all objects.
    class IObject
    {
    public:
        /// Standard destructor.
        virtual ~IObject() { }

        /// Returns the name of the object ("property getter").
        virtual String8 GetName() const = 0;

        /// Returns the description of the object ("property getter").
        virtual String8 GetDescription() const = 0;
    };
}
```

This interface is the base interface for almost all other SMP2 interfaces. While most interfaces derive from `IComponent`, which itself is derived from `IObject`, some objects (including `IEntryPoint`, `IEventSink`, `IEventSource`, `IContainer` and `IReference`) are directly derived from `IObject`.

3.2.2 Components

Many elements in SMP2 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.

3.2.2.1 IComponent

```
namespace Smp
{
    // Forward declaration because of circular references.
    class IComposite;

    /// Base interface for all components.
    class IComponent : public virtual IObject
    {
    public:
        /// Returns the parent component of the component ("property getter").
        virtual IComposite* GetParent() const = 0;
    };
}
```

 © EUROPEAN SPACE AGENCY, 2005

All SMP2 components implement this base interface.

3.2.2.2 ComponentCollection

```
namespace Smp
{
    /// Collection of components.
    typedef std::vector<IComponent*> ComponentCollection;
}
```

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

3.2.2.3 ModelStateKind

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

Before going into `Initialising` state, the simulator has to ensure that every model is in `Connected` state.

```
namespace Smp
{
    enum ModelStateKind
    {
        MSK_Created,
        MSK_Publishing,
        MSK_Configured,
        MSK_Connected
    };
}
```

3.2.2.4 InvalidModelState

This exception is thrown by any of the model state transition methods when it is called in an invalid state of the model. The exception names both the invalid and the excepted state.

```
static const String8 InvalidModelStateTemplate =
    "Invalid state: Expected '%s', but is '%s'";

/// Invalid model state.
/// This exception is raised by a model when one of the
/// state transition methods is called in an invalid state.
class InvalidModelState : public Smp::Exception
{
public:
    /// Name of the state that is not valid.
    Smp::ModelStateKind invalidState;
    /// Name of the state that was expected.
    Smp::ModelStateKind expectedState;

    /// Constructor for new exception.
    InvalidModelState(
        Smp::ModelStateKind _invalidState,
        Smp::ModelStateKind _expectedState) throw() :
    Smp::Exception("InvalidModelState"),
        invalidState(_invalidState),
        expectedState(_expectedState)
    {
        ...
    }
};
```

3.2.2.5 IModel

```
namespace Smp
{
    // Forward declarations because of circular references.
    class IPublication;
    class ISimulator;

    /// Model base interface.
    class IModel : public virtual IComponent
    {
    public:
        /// Return model state.
        virtual ModelStateKind GetState() const = 0;

        /// Request for publication.
        virtual void Publish(IPublication* receiver) throw (
            Smp::IModel::InvalidModelState) = 0;

        /// Request for configuration.
        virtual void Configure(Smp::Services::ILogger* logger) throw (
            Smp:: IModel::InvalidModelState) = 0;

        /// Connect model to simulator.
        virtual void Connect(ISimulator* simulator) throw (
            Smp:: IModel::InvalidModelState) = 0;
    };
}
```

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`.

3.2.2.6 ModelCollection

```
namespace Smp
{
    /// Collection of models.
    typedef std::vector<IModel*> ModelCollection;
}
```

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

3.2.2.7 IService

```
namespace Smp
{
    /// Base interface for simulation services.
    class IService : public virtual IComponent
    {
    };
}
```

All SMP services implement this interface. It does not add any specific functionality.

3.2.2.8 ServiceCollection

```
namespace Smp
{
    /// Collection of services.
    typedef std::vector<IService*> ServiceCollection;
}
```

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

 © EUROPEAN SPACE AGENCY, 2005

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 SMP2 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

```
namespace Smp
{
    /// Aggregate component.
    class IAggregate : public virtual IComponent
    {
    public:
        /// Get all references.
        virtual const ReferenceCollection* GetReferences() const = 0;

        /// Get a reference by name.
        virtual IReference* GetReference(String8 name) const = 0;
    };
}
```

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

3.3.1.2 IReference

```
namespace Smp
{
    /// Reference to components.
    class IReference : public virtual IObject
    {
    public:
        /// Get all referenced components.
        virtual const ComponentCollection* GetComponents() const = 0;

        /// Get a referenced component by name.
        virtual IComponent* GetComponent(String8 name) const = 0;
    };
}
```

A reference allows querying for the referenced components.

3.3.1.3 ReferenceCollection

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

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

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 any direction.

3.3.2.1 IComposite

```
namespace Smp
{
    /// Composite component.
    class IComposite : public virtual IComponent
    {
    public:
        /// Get all containers.
        virtual const ContainerCollection* GetContainers() const = 0;

        /// Get a container by name.
        virtual IContainer* GetContainer(String8 name) const = 0;
    };
}
```

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

3.3.2.2 IContainer

```
namespace Smp
{
    /// Container of components.
    class IContainer : public virtual IObject
    {
    public:
        /// Get all contained components.
        virtual const ComponentCollection* GetComponents() const = 0;

        /// Get a contained component by name.
        virtual IComponent* GetComponent(String8 name) const = 0;
    };
}
```

A container allows querying for its children.

3.3.2.3 ContainerCollection

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

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

© EUROPEAN SPACE AGENCY, 2005

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 IEventSink

```
namespace Smp
{
    /// Event sink that can be subscribed to event source (IEventSource).
    class IEventSink : public virtual IObject
    {
    public:
        /// Event notification.
        virtual void Notify(IObject* sender, AnySimple arg) = 0;
    };
}
```

Provide notification method (event handler) that can be called by event publishers when an event is emitted.

3.3.3.2 EventSinkCollection

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

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

3.3.3.3 IEventSource

```
namespace Smp
{
    /// Event source that event sinks (IEventSink) can subscribe to.
    class IEventSource : public virtual IObject
    {
    public:
        /// Event sink is already subscribed.
        class AlreadySubscribed : public Exception
        {
        public:
            /// Event source the event sink is subscribed to.
            const IEventSource* eventSource;
            /// Event sink that is already subscribed.
            const IEventSink*   eventSink;

            AlreadySubscribed(
                const IEventSource* _eventSource,
                const IEventSink*   _eventSink) throw() :
            Smp::Exception("AlreadySubscribed"),
                eventSource(_eventSource),
                eventSink  (_eventSink) { ... }
        };

        /// Event sink is not subscribed.
        class NotSubscribed : public Exception
        {
        public:
            /// Event source the event sink is not subscribed to.
            const IEventSource* eventSource;
            /// Event sink that is not subscribed.
            const IEventSink* eventSink;

            NotSubscribed(
                const IEventSource* _eventSource,
                const IEventSink*   _eventSink) throw() :
            Smp::Exception("NotSubscribed"),
                eventSource(_eventSource),
                eventSink  (_eventSink) { ... }
        };

        /// Event sink is not compatible with event source.
        class InvalidEventSink : public Exception
        {
        public:
            /// Event source the event sink is subscribed to.
            const IEventSource* eventSource;
            /// Event sink that is not of valid type.
            const IEventSink*   eventSink;

            InvalidEventSink(
                const IEventSource* _eventSource,
                const IEventSink*   _eventSink) throw() :
            Smp::Exception("InvalidEventSink"),
                eventSource(_eventSource),
                eventSink  (_eventSink) { ... }
        };

        /// Event subscription.
        virtual void Subscribe(IEventSink* eventSink) throw (
            Smp::IEventSource::AlreadySubscribed,
            Smp::IEventSource::InvalidEventSink) = 0;

        /// Event unsubscription.
        virtual void Unsubscribe(IEventSink* eventSink) throw (
            Smp::IEventSource::NotSubscribed) = 0;
    };
}
```

Allow event consumers to subscribe or unsubscribe to/from an event.

© EUROPEAN SPACE AGENCY, 2005

3.3.3.4 EventSourceCollection

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

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

3.3.4 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`.

When a model publishes information about its available operations to the `IPublication` interface, this information can be used to create corresponding request objects. Therefore, the C++ Mapping of the `IPublication` interface contains two methods `CreateRequest()` and `DeleteRequest()` that a model can use to delegate the implementation of these two methods.

Like all features in this section, dynamic invocation is an optional feature.

3.3.4.1 IDynamicInvocation

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

```
namespace Smp
{
    /// Dynamic invocation.
    class IDynamicInvocation : public virtual IComponent
    {
    public:
        /// Invalid operation name.
        class InvalidOperationName : public Exception
        {
        public:
            /// Operation name of request passed to the Invoke() method.
            String8 operationName;

            InvalidOperationName(
                String8 _operationName) throw() :
            Smp::Exception("InvalidOperationName"),
                operationName (_operationName) { ... }
        };

        /// Invalid parameter count.
        class InvalidParameterCount : public Exception
        {
        public:
            /// Operation name of request passed to the Invoke() method.
            String8 operationName;
            /// Correct number of parameters of operation.
            const Int32 operationParameters;
            /// Wrong number of parameters of operation.
            const Int32 requestParameters;

            InvalidParameterCount(
                String8 _operationName,
                const Int32 _operationParameters,
                const Int32 _requestParameters) throw() :
            Smp::Exception("InvalidParameterCount"),
                operationName (_operationName),
                operationParameters(_operationParameters),
                requestParameters(_requestParameters) { ... }
        };

        /// Invalid parameter type.
        class InvalidParameterType : public Exception
        {
        public:
            /// Operation name of request passed to the Invoke() method.
            String8 operationName;
            /// Name of parameter of wrong type.
            String8 parameterName;

            InvalidParameterType(
                String8 _operationName,
                String8 _parameterName) throw() :
            Smp::Exception("InvalidParameterType"),
                operationName(_operationName),
                parameterName(_parameterName) { ... }
        };

        /// Create request.
        virtual IRequest *CreateRequest(String8 operationName) = 0;

        /// Dynamic invocation of operation.
        virtual void Invoke(IRequest *request) throw (
            Smp::IDynamicInvocation::InvalidOperationName,
            Smp::IDynamicInvocation::InvalidParameterCount,
            Smp::IDynamicInvocation::InvalidParameterType) = 0;

        /// Delete request.
        virtual void DeleteRequest(IRequest *request) = 0;
    };
}
```


© EUROPEAN SPACE AGENCY, 2005

3.3.4.2 IRequest

```
namespace Smp
{
    /// Request for dynamic invocation.
    class IRequest
    {
    public:
        /// Invalid parameter index.
        class InvalidParameterIndex : public Exception
        {
        public:
            /// Name of operation.
            String8 operationName;
            /// Invalid parameter index used.
            const Int32 parameterIndex;

            InvalidParameterIndex(
                String8 _operationName,
                const Int32  _parameterIndex) throw() :
            Smp::Exception("InvalidParameterIndex"),
                operationName (_operationName),
                parameterIndex(_parameterIndex) {}
        };

        /// Invalid value for parameter.
        class InvalidParameterValue : public Exception
        {
        public:
            /// Name of parameter value was assigned to.
            String8 parameterName;
            /// Value that was passed as parameter.
            const AnySimple value;

            InvalidParameterValue(
                String8 _parameterName,
                const AnySimple _value) throw() :
            Smp::Exception("InvalidParameterValue"),
                parameterName (_parameterName),
                value(_value) {}
        };

        /// Invalid value for return value.
        class InvalidReturnValue : public Exception
        {
        public:
            /// Name of operation the return value was assigned to.
            String8 operationName;
            /// Value that was passed as return value.
            const AnySimple value;

            InvalidReturnValue(
                String8 _operationName,
                const AnySimple _value) throw() :
            Smp::Exception("InvalidReturnValue"),
            operationName (_operationName),
            value(_value) {}
        };

        /// Operation is a void operation.
        class VoidOperation : public Exception
        {
        public:
            /// Name of operation.
            String8 operationName;

            VoidOperation(
                String8 _operationName) throw() :
            Smp::Exception("VoidOperation"),
            operationName (_operationName) {}
        };
```

© EUROPEAN SPACE AGENCY, 2005

```
        /// Virtual Destructor.
        virtual ~IRequest() { }

        /// Get operation name.
        virtual String8 GetOperationName() const = 0;

        /// Get parameter count.
        virtual Int32 GetParameterCount() const = 0;

        /// Get index of a parameter.
        virtual Int32 GetParameterIndex(String8 parameterName) const = 0;

        /// Set a parameter value.
        virtual void SetParameterValue(
            const Int32 index,
            const AnySimple value) throw (
            Smp::IRequest::InvalidParameterIndex,
            Smp::IRequest::InvalidParameterValue,
            Smp::InvalidAnyType) = 0;

        /// Get a parameter value.
        virtual AnySimple GetParameterValue(const Int32 index) const throw (
            Smp::IRequest::InvalidParameterIndex) = 0;

        /// Set the return value.
        virtual void SetReturnValue(const AnySimple value) throw (
            Smp::IRequest::InvalidReturnValue,
            Smp::IRequest::VoidOperation,
            Smp::InvalidAnyType) = 0;

        /// Get the return value.
        virtual AnySimple GetReturnValue() const throw (
            Smp::IRequest::VoidOperation) = 0;
    };
}
```

The request holds information that is passed between a client invoking an operation via the `IDynamicInvocation` interface and a component being invoked.

3.3.5 Persistence

Persistence of SMP2 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. 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. 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 features in this section, self-persistence of models and components is an optional feature, while external persistence (via the Store and Restore methods of the `ISimulator` interface) is a mandatory feature of every SMP2 simulation environment.

 © EUROPEAN SPACE AGENCY, 2005

3.3.5.1 IPersist

```
namespace Smp
{
    /// Self persistence for components.
    class IPersist : public virtual IComponent
    {
    public:
        /// Cannot restore from storage reader (IStorageReader).
        class CannotRestore : public std::exception
        {
        public:
            /// Error message indicating details of the problem.
            String8 message;

            CannotRestore(String8 _message) throw() :
            Smp::Exception("CannotRestore"),
            message(_message) { ... }
        };

        /// Cannot store to storage writer (IStorageWriter).
        class CannotStore : public std::exception
        {
        public:
            /// Error message indicating details of the problem.
            String8 message;

            CannotStore(String8 _message) throw() :
            Smp::Exception("CannotStore"),
            message(_message) { ... }
        };

        /// Restore component state from storage.
        /// @param    reader Interface that allows reading from storage.
        virtual void Restore(IStorageReader* reader) throw (
            Smp::IPersist::CannotRestore) = 0;

        /// Store component state to storage.
        /// @param    writer Interface that allows writing to storage.
        virtual void Store(IStorageWriter* writer) throw (
            Smp::IPersist::CannotStore) = 0;
    };
}
```

A component may implement this interface if it wants to have control over loading and saving of its state.

3.3.5.2 IStorageReader

```
namespace Smp
{
    /// Storage reader.
    class IStorageReader
    {
    public:
        /// Virtual destructor.
        virtual ~IStorageReader() { }

        /// Read data from storage.
        virtual void Restore(void* address, Int32 size) = 0;
    };
}
```

Provide functionality to restore data from storage. 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 component implementing `IPersist`.

© EUROPEAN SPACE AGENCY, 2005

3.3.5.2.1 Restore

```
void Restore(void *address, Int32 size);
```

Restore a memory block from storage.

Parameters:

address	Memory address of memory block.
size	Number of bytes to read from storage.

Returns:

Void.

Exceptions:

None.

Remarks:

The memory block is not interpreted, but read from storage in binary format.

3.3.5.3 IStorageWriter

```
namespace Smp
{
    /// Storage writer.
    class IStorageWriter
    {
    public:
        /// Virtual destructor.
        virtual ~IStorageWriter() { }

        /// Write data to storage.
        virtual void Store(void* address, Int32 size) = 0;
    };
}
```

Provide functionality to store data to storage. 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`.

3.3.5.3.1 Store

```
void Store(void *address, Int32 size);
```

Store a memory block to storage.

Parameters:

address	Memory address of memory block.
size	Number of bytes to write to storage.

Returns:

Void.

Exceptions:

None.

© EUROPEAN SPACE AGENCY, 2005

Remarks:

The memory block is not interpreted, but written to storage in binary format.

3.4 Model Mechanisms

While the IModel interface defines the mandatory functionality every SMP2 model has to provide, this section introduces additional mechanisms available for more advanced use. Entry points allow models exposing void functions to the scheduler or event manager services.

3.4.1 Entry Points

An entry point is a void function with no return value that can be exposed e.g. to the scheduler service.

3.4.1.1 IEntryPoint

```
namespace Smp
{
    /// Entry point for IScheduler or IEventManager.
    class IEntryPoint : public virtual IObject
    {
    public:
        /// Entry point owner.
        virtual IComponent* GetOwner(void) const = 0;

        /// Entry point execution.
        virtual void Execute(void) const = 0;
    };
}
```

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

3.4.1.2 EntryPointCollection

```
namespace Smp
{
    /// Collection of entry points.
    typedef std::vector<const IEntryPoint *> EntryPointCollection;
}
```

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

3.5 Management Interfaces

Managed interfaces allow external components to access all mechanisms by name. This includes the basic component features, 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 interface allowing reading and writing field values.

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 build from an XML document (namely an SMDL Assembly) during the `Creating` phase.

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 IManagedObject

```
namespace Smp
{
    namespace Management
    {
        /// Managed object.
        class IManagedObject : public virtual IObject
        {
        public:
            /// Defines the name of the managed object ("property setter").
            virtual void SetName(String8 name) throw (
                Smp::InvalidObjectName) = 0;

            /// Defines the description of the managed object ("property setter").
            virtual void SetDescription(String8 description) = 0;
        };
    }
}
```

A managed object additionally allows assigning name and description.

3.5.1.2 IManagedComponent

```
namespace Smp
{
    namespace Management
    {
        /// Managed component.
        class IManagedComponent :
            public virtual IManagedObject,
            public virtual IComponent
        {
        public:
            /// Defines the parent component ("property setter").
            virtual void SetParent(IComposite* parent) = 0;
        };
    }
}
```

A managed component additionally allows assigning the parent.

 © EUROPEAN SPACE AGENCY, 2005

3.5.2 Managed Component Mechanisms

The component mechanisms introduced in 3.3 (Component Mechanisms) do not provide full access for external components. 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 respective references. For event sources and event sinks, the managed interfaces provide access to the elements by name.

3.5.2.1 IManagedReference

```
namespace Smp
{
    namespace Management
    {
        /// Managed reference.
        class IManagedReference : public virtual IReference
        {
        public:
            /// Reference is full.
            class ReferenceFull : public Exception
            {
            public:
                /// Name of full reference.
                String8 referenceName;
                /// Number of components in the reference.
                const Int64  referenceSize;

                ReferenceFull(
                    String8 _referenceName,
                    const Int64  _referenceSize) throw() :
                Smp::Exception("ReferenceFull"),
                    referenceName(_referenceName),
                    referenceSize(_referenceSize) { ... }
            };

            class NotReferenced : public Exception
            {
            public:
                /// Name of reference.
                String8 referenceName;
                /// Component that is not referenced.
                const IComponent* component;

                NotReferenced(
                    String8 _referenceName,
                    const IComponent* _component) throw() :
                Smp::Exception("NotReferenced"),
                    referenceName(_referenceName),
                    component(_component) { ... }
            };

            /// Add component.
            virtual void AddComponent(IComponent* component) throw (
                Smp::Management::IManagedReference::ReferenceFull,
                Smp::InvalidObjectType) = 0;

            /// Get the number of referenced components.
            virtual Int64 Count() const = 0;
            /// Get lower bound of multiplicity.
            virtual Int64 Lower() const = 0;
            /// Get upper bound for number of components.
            virtual Int64 Upper() const = 0;

            /// Remove component.
            virtual void RemoveComponent(IComponent* component) throw (
                Smp::Management::IManagedReference::NotReferenced) = 0;
        };
    }
}
```

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

3.5.2.2 IManagedContainer

```
namespace Smp
{
    namespace Management
    {
        /// Managed container.
        class IManagedContainer : public virtual IContainer
        {
        public:
            /// Container is full.
            class ContainerFull : public Exception
            {
            public:
                /// Name of full container.
                String8 containerName;
                /// Number of components in the container, which is its Upper()
                /// limit when the container is full.
                const Int64  containerSize;

                ContainerFull(
                    String8 _containerName,
                    const Int64 _containerSize) throw() :
                Smp::Exception("ContainerFull"),
                    containerName(_containerName),
                    containerSize(_containerSize) { ... }
            };

            /// Add component.
            virtual void AddComponent(IComponent* component) throw (
                Smp::Management::IManagedContainer::ContainerFull,
                Smp::DuplicateName,
                Smp::InvalidObjectType) = 0;

            /// Get the number of contained components.
            virtual Int64 Count() const = 0;
            /// Get lower bound of multiplicity.
            virtual Int64 Lower() const = 0;
            /// Get upper bound for number of components.
            virtual Int64 Upper() const = 0;
        };
    }
}
```

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

3.5.2.3 IEventConsumer

```
namespace Smp
{
    namespace Management
    {
        /// Event consumer.
        class IEventConsumer : public virtual IComponent
        {
        public:
            /// Get all event sinks.
            virtual const EventSinkCollection* GetEventSinks() const = 0;

            /// Get an event sink by name.
            virtual IEventSink* GetEventSink(String8 name) const = 0;
        };
    }
}
```

Component that holds event sinks, which may be subscribed to other component's event sources.

 © EUROPEAN SPACE AGENCY, 2005

3.5.2.4 IEventProvider

```
namespace Smp
{
    namespace Management
    {
        /// Event publisher.
        class IEventProvider : public virtual IComponent
        {
        public:
            /// Get all event sources.
            virtual const EventSourceCollection *GetEventSources() const = 0;

            /// Get an event source by name.
            virtual IEventSource *GetEventSource(String8 name) const = 0;
        };
    }
}
```

Component that holds event sources, which allow other components to subscribe their event sinks.

3.5.3 Managed Model Mechanisms

The model mechanisms introduced in 3.3.4 (Dynamic Invocation) do not provide full access for external components. To overcome these limitations, managed interfaces are provided with full access to all functionality. For entry points, the managed interface provides access to the entry points by name. For fields, access by name is provided by an extended interface allowing reading and writing field values.

3.5.3.1 IManagedModel

```
namespace Smp
{
    namespace Management
    {
        /// Managed model.
        class IManagedModel : public virtual IManagedComponent,
                              public virtual IModel
        {
        public:
            /// Invalid field name.
            class InvalidFieldName : public Exception
            {
            public:
                /// Fully qualified field name that is invalid.
                String8 fieldName;

                InvalidFieldName(
                    String8 _fieldName) throw() :
                Smp::Exception("InvalidFieldName"),
                    fieldName(_fieldName) { ... }
            };

            /// Invalid value for field.
            class InvalidFieldValue : public Exception
            {
            public:
                /// Fully qualified field name the value was assigned to.
                String8 fieldName;
                /// Value that was passed as new field value.
                const AnySimple invalidValue;

                InvalidFieldValue(
                    String8 _fieldName,
                    const AnySimple _invalidValue) throw() :
                Smp::Exception("InvalidFieldValue"),
                    fieldName(_fieldName),
                    invalidValue(_invalidValue) { ... }
            };
```

```
            /// Invalid array size.
            class InvalidArraySize : public Exception
            {
            public:
                /// Name of field that has been accessed.
                String8 fieldName;
                /// Invalid array size.
                const Int64 givenSize;
                /// Real array size.
                const Int64 arraySize;

                InvalidArraySize(
                    String8 _fieldName,
                    const Int64 _givenSize,
                    const Int64 _arraySize) throw() :
                Smp::Exception("InvalidArraySize"),
                    fieldName(_fieldName),
                    givenSize(_givenSize),
                    arraySize(_arraySize) { ... }
            };

            /// Invalid value for field.
            class InvalidArrayValue : public Exception
            {
            public:
                /// Fully qualified field name the value was assigned to.
                String8 fieldName;
                /// Value that was passed as new field value.
                const AnySimpleArray invalidValue;

                InvalidArrayValue(
                    String8 _fieldName,
                    const AnySimpleArray _invalidValue) throw() :
                Smp::Exception("InvalidArrayValue"),
                    fieldName(_fieldName),
                    invalidValue(_invalidValue) { ... }
            };

            /// Get the value of a field which is typed by a system type.
            virtual AnySimple GetFieldValue(String8 fullName) throw (
                Smp::Management::IManagedModel::InvalidFieldName) = 0;

            /// Set the value of a field which is typed by a system type.
            virtual void SetFieldValue(String8 fullName, const AnySimple value) throw(
                Smp::Management::IManagedModel::InvalidFieldName,
                Smp::Management::IManagedModel::InvalidFieldValue) = 0;

            /// Get the value of an array field which is typed by a system type.
            virtual void GetArrayValue(
                String8 fullName,
                const AnySimpleArray values,
                const Int32 length) throw (
                Smp::Management::IManagedModel::InvalidFieldName,
                Smp::Management::IManagedModel::InvalidArraySize) = 0;

            /// Set the value of an array field which is typed by a system type.
            virtual void SetArrayValue(
                String8 fullName,
                const AnySimpleArray values,
                const Int32 length) throw (
                Smp::Management::IManagedModel::InvalidFieldName,
                Smp::Management::IManagedModel::InvalidArraySize,
                Smp::Management::IManagedModel::InvalidArrayValue) = 0;
        };
    }
}
```

A managed model is a managed component that additionally allows querying or modifying field values.

 © EUROPEAN SPACE AGENCY, 2005

3.5.3.2 IEntryPointPublisher

```
namespace Smp
{
    namespace Management
    {
        /// Entry point publisher.
        class IEntryPointPublisher : public virtual IModel
        {
        public:
            /// Get all entry points.
            virtual const EntryPointCollection* GetEntryPoints() const = 0;

            /// Get an entry point by name.
            virtual const IEntryPoint* GetEntryPoint(String8 name) const = 0;
        };
    }
}
```

An entry point publisher is a model that publishes entry points, which may be registered, for example, with the Scheduler or the Event Manager services.

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.

```
namespace Smp
{
    const String8 SMP_SimulatorModels   = "Models";
    const String8 SMP_SimulatorServices = "Services";
}
```

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.

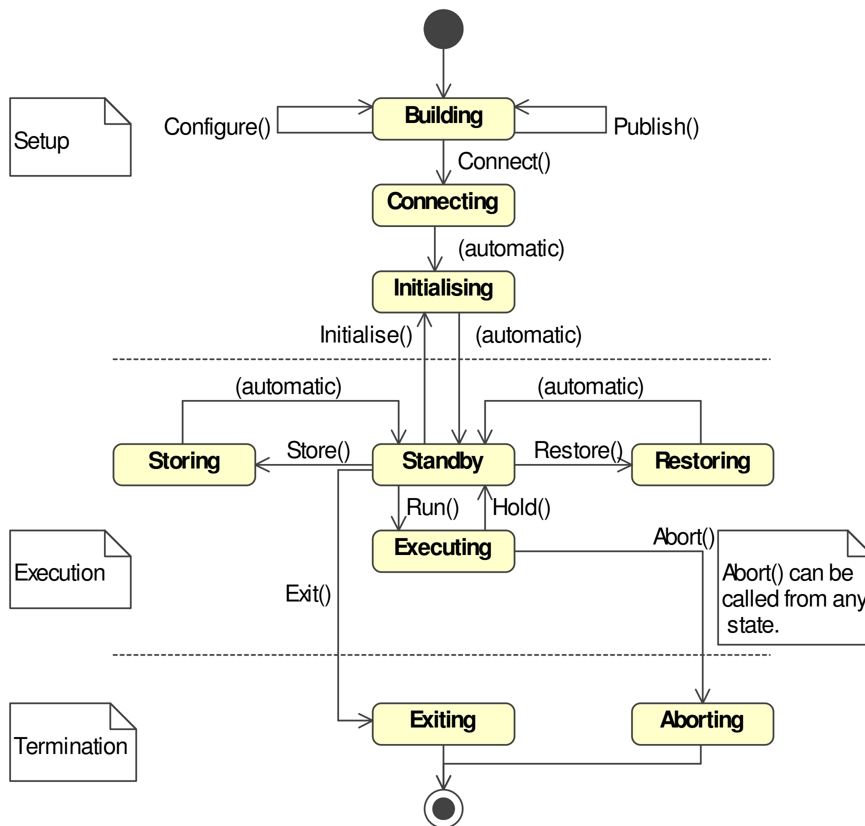

Figure 3-1: Simulation Environment State Diagram with State Transition Methods

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 3-1. 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.

3.6.1.1 SimulatorStateKind

```
namespace Smp
{
    /// Available simulator states.
    enum SimulatorStateKind
    {
        SSK_Building,
        SSK_Connecting,
        SSK_Initialising,
        SSK_Standby,
        SSK_Executing,
        SSK_Storing,
        SSK_Restoring,
        SSK_Exiting,
        SSK_Aborting
    };
}
```

Enumeration of the available states of the simulator.

 © EUROPEAN SPACE AGENCY, 2005

3.6.1.2 ISimulator

```
namespace Smp
{
    /// Interface to the Simulator.
    class ISimulator : public virtual IComposite
    {
    public:
        /// Get all models.
        virtual const ModelCollection* GetModels() = 0;
        /// Get a root model by name.
        virtual IModel* GetModel(String8 name) = 0;
        /// Add a root model.
        virtual void AddModel(IModel* model) throw (Smp::DuplicateName) = 0;

        /// Get all services.
        virtual const ServiceCollection* GetServices() = 0;
        /// Get a service by name.
        virtual IService* GetService(String8 name) = 0;
        /// Add a user-defined service.
        virtual void AddService(IService* service) throw (Smp::DuplicateName) = 0;

        /// Query for mandatory logger service.
        virtual Services::ILogger* GetLogger() const = 0;
        /// Query for mandatory scheduler service.
        virtual Services::IScheduler* GetScheduler() const = 0;
        /// Query for mandatory time keeper service.
        virtual Services::ITimeKeeper* GetTimeKeeper() const = 0;
        /// Query for mandatory event manager service.
        virtual Services::IEventManager* GetEventManager() const = 0;

        /// Simulator state.
        virtual SimulatorStateKind GetState() const = 0;

        /// Call Publish() method of models.
        virtual void Publish() = 0;

        /// Call Configure() method of models.
        virtual void Configure() = 0;

        /// Enter connecting state.
        virtual void Connect() = 0;

        /// Enter initialising state.
        virtual void Initialise() = 0;

        /// Enter standby state.
        virtual void Hold() = 0;

        /// Enter executing state.
        virtual void Run() = 0;

        /// Enter storing state.
        virtual void Store(String8 filename) = 0;

        /// Enter restoring state.
        virtual void Restore(String8 filename) = 0;

        /// Enter exiting state.
        virtual void Exit() = 0;

        /// Enter aborting state.
        virtual void Abort() = 0;

        /// Add initialisation entry point.
        virtual void AddInitEntryPoint(IEntryPoint* entryPoint) = 0;
    };
}
```

This interface gives access to the simulation environment state and services.

© EUROPEAN SPACE AGENCY, 2005

3.6.1.3 IDynamicSimulator

```
namespace Smp
{
    /// This interface gives access to a dynamic simulator.
    class IDynamicSimulator : public virtual ISimulator
    {
    public:
        /// Duplicate Uuid.
        class DuplicateUuid : public Exception
        {
        public:
            /// Name of factory that tried to register under this Uuid.
            String8 newName;
            /// Name of factory already registered under this Uuid.
            String8 oldName;

            DuplicateUuid(
                String8 _newName,
                String8 _oldName) throw() :
            Smp::Exception("DuplicateUuid"),
                newName(_newName),
                oldName(_oldName) { ... }
        };

        /// Register a global component factory with the simulator.
        virtual void RegisterFactory(IFactory *componentFactory) throw (
            Smp::IDynamicSimulator::DuplicateUuid) = 0;

        /// Create an instance of the given component implementation.
        virtual IComponent* CreateInstance(const Uuid implUuid) = 0;

        /// Get the factory of the given component implementation.
        virtual const IFactory* GetFactory(const Uuid implUuid) = 0;

        /// Get all factories of the given component specification.
        virtual FactoryCollection* GetFactories(const Uuid specUuid) = 0;
    };
}
```

This interface extends the `ISimulator` interface and adds methods to dynamically create components (typically models) from component factories. It makes use of the `IFactory` interface for factories.

3.6.1.4 IFactory

```
namespace Smp
{
    /// This interface is implemented by all component factories.
    class IFactory : public virtual IObject
    {
    public:
        /// Get specification identifier of factory.
        virtual Uuid GetSpecification() const = 0;

        /// Get implementation identifier of factory.
        virtual Uuid GetImplementation() const = 0;

        /// Create a new instance.
        virtual IComponent* CreateInstance() = 0;

        /// Delete an existing instance.
        virtual void DeleteInstance(IComponent* instance) = 0;
    };
}
```

This interface is implemented by all component factories.

 © EUROPEAN SPACE AGENCY, 2005

3.6.1.5 FactoryCollection

```
namespace Smp
{
    /// Collection of factories.
    typedef std::vector<IFactory*> FactoryCollection;
}
```

A factory collection is an ordered collection of factories, which allows iterating all members.

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 section 6 (Publication).

© EUROPEAN SPACE AGENCY, 2005

This Page is Intentionally left Blank

© EUROPEAN SPACE AGENCY, 2005

4. SIMULATION SERVICES

In order to facilitate the inter-operability between SMP2 compliant simulation environments (i.e. run-time simulation kernels), several *Simulation Services* are defined in the SMP2 specification. Some services are mandatory, where others are optional. A standardised mechanism of how models can acquire services is part of the standard.

4.1 Mandatory Services

Any SMP2 compliant simulation environment *shall* support the following standard services.

4.1.1 Logger

The logger service provides a method to send a log message to the simulation log file.

4.1.1.1 ILogger

```
namespace Smp
{
    namespace Services
    {
        /// Name of Logger service.
        const String8 SMP_Logger    = "Smp_Logger";

        /// Identifier of log message kind.
        typedef Int32 LogMessageKind;

        /// This interface gives access to the Logger.
        class ILogger : public virtual IService
        {
        public:
            /// Return identifier of log message kind by name.
            virtual LogMessageKind GetLogMessageKind(
                String8 messageKindName) = 0;

            /// Mechanism to log a messages.
            virtual void Log(
                const IObject *sender,
                String8 message,
                const LogMessageKind kind = LMK_Information) = 0;
        };
    }
}
```

All objects in a simulation can log messages using this service.

4.1.1.2 Predefined Log Message Kinds

```
namespace Smp
{
    namespace Services
    {
        // Predefined Log Message Kinds
        const LogMessageKind LMK_Information = 0;  ///< Information message.
        const LogMessageKind LMK_Event       = 1;  ///< Event message.
        const LogMessageKind LMK_Warning     = 2;  ///< Warning message.
        const LogMessageKind LMK_Error       = 3;  ///< Error message.
        const LogMessageKind LMK_Debug       = 4;  ///< Debug message.

        // Names of predefined Log Message Kinds
        const String8 LMK_InformationName = "Information";
        const String8 LMK_EventName       = "Event";
        const String8 LMK_WarningName     = "Warning";
        const String8 LMK_ErrorName       = "Error";
        const String8 LMK_DebugName       = "Debug";
    }
}
```

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, e.g. 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.

4.1.2 Time Keeper

SMP2 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 and mission time. Further, the service provides Zulu time based on the clock of the computer.

4.1.2.1 ITimeKeeper

```
namespace Smp
{
    namespace Services
    {
        /// Name of Time Keeper service.
        const String8 SMP_TimeKeeper = "Smp_TimeKeeper";

        /// This interface gives access to the Time Keeper.
        class ITimeKeeper : public virtual IService
        {
        public:
            /// Return simulation time.
            virtual Duration GetSimulationTime() = 0;
            /// Return Epoch time.
            virtual DateTime GetEpochTime() = 0;
            /// Return Mission time.
            virtual Duration GetMissionTime() = 0;
            /// Return Zulu time.
            virtual DateTime GetZuluTime() = 0;

            /// Set Epoch time.
            virtual void SetEpochTime(const DateTime epochTime) = 0;
            /// Set Mission start.
            virtual void SetMissionStart(const DateTime missionStart) = 0;
            /// Set Mission time.
            virtual void SetMissionTime(const Duration missionTime) = 0;
        };
    }
}
```

Components can query for the time, and set the epoch or mission time.

 © EUROPEAN SPACE AGENCY, 2005

4.1.2.2 TimeKind

```
namespace Smp
{
    namespace Services
    {
        /// Enumeration of supported time types.
        enum TimeKind
        {
            TK_SimulationTime,  ///< Simulation time.
            TK_EpochTime,       ///< Epoch time.
            TK_MissionTime,     ///< Mission time.
            TK_ZuluTime         ///< Zulu time.
        };
    }
}
```

SMP2 supports four different kinds of time.

4.1.3 Scheduler

The scheduler service calls entry points of models based on events triggered by one of the four time kinds. In addition, tasks can be created, which can contain an ordered collection of entry points. This allows scheduling several entry points in one call to the scheduler.

4.1.3.1 IScheduler

```
namespace Smp
{
    namespace Services
    {
        /// Name of Scheduler service.
        const String8 SMP_Scheduler = "Smp_Scheduler";

        /// This interface gives access to the Scheduler.
        class IScheduler : public virtual IService
        {
        public:
            virtual void AddImmediateEvent(
                const IEntryPoint *entryPoint) = 0;
            virtual EventId AddSimulationTimeEvent(
                const IEntryPoint *entryPoint,
                const Duration simulationTime,
                const Duration cycleTime = 0,
                const Int64 count = 0) = 0;
            virtual EventId AddMissionTimeEvent(
                const IEntryPoint* entryPoint,
                const Duration missionTime,
                const Duration cycleTime = 0,
                const Int64 count = 0) = 0;
            virtual EventId AddEpochTimeEvent(
                const IEntryPoint* entryPoint,
                const DateTime epochTime,
                const Duration cycleTime = 0,
                const Int64 count = 0) = 0;
            virtual EventId AddZuluTimeEvent(
                const IEntryPoint* entryPoint,
                const DateTime zuluTime,
                const Duration cycleTime = 0,
                const Int64 count = 0) = 0;

            virtual void SetEventSimulationTime(
                const EventId event,
                const Duration simulationTime) throw ( InvalidEventId ) = 0;
            virtual void SetEventMissionTime(
                const EventId event,
                const Duration missionTime)    throw ( InvalidEventId ) = 0;
            virtual void SetEventEpochTime(
                const EventId event,
                const DateTime epochTime)      throw ( InvalidEventId ) = 0;
            virtual void SetEventZuluTime(
                const EventId event,
                const DateTime zuluTime)       throw ( InvalidEventId ) = 0;
            /// Set event cycle time.
            virtual void SetEventCycleTime(
                const EventId event,
                const Duration cycleTime)      throw ( InvalidEventId ) = 0;
            /// Set event count.
            virtual void SetEventCount(
                const EventId event,
                const Int64 count)             throw ( InvalidEventId ) = 0;

            virtual void RemoveEvent(const Int32 event) = 0;
        };
    }
}
```

Components can register (`Add`) and unregister (`Remove`) entry points for scheduling. Further, they can set (`Set`) individual attributes of events on the scheduler.

 © EUROPEAN SPACE AGENCY, 2005

4.1.3.2 ITask

```
namespace Smp
{
    namespace Services
    {
        /// Interface for Scheduler task.
        class ITask : public virtual IEntryPoint
        {
        public:
            /// Get all entry points.
            virtual EntryPointCollection GetEntryPoints() = 0;

            /// Add entry point.
            virtual void AddEntryPoint(const IEntryPoint *entryPoint) = 0;
        };
    }
}
```

This interface extends `IEntryPoint` to allow scheduling tasks. A Task is an ordered collection of entry points. Entry points in a task will be executed in the order they have been added.

4.1.4 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.

Remarks:

> 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`".

4.1.4.1 IEventManager

Both the event manager and the scheduler use and return event identifiers. Therefore, this type as well as the `InvalidEventId` exception is defined outside of these two interfaces, but within the `Services` namespace.

```
namespace Smp
{
    namespace Services
    {
        /// Identifier of global event of scheduler or event manager service.
        typedef Int64 EventId;

        /// Invalid event identifier.
        class InvalidEventId : public Smp::Exception
        {
        public:
            /// Invalid event identifier.
            const Int32 event;

            InvalidEventId(
                const Int32 _event) throw() :
            Smp::Exception("InvalidEventId"),
                event(_event) {}
        };
    }
}
```

© EUROPEAN SPACE AGENCY, 2005

The event manager itself provides methods to subscribe entry points to global events, and to unsubscribe them later. In addition, user-defined event types can be defined using the `GetEventId()` method.

```
        /// Name of Event Manager service.
        const String8 SMP_EventManager = "Smp_EventManager";

        /// This interface gives access to the Event Manager.
        class IEventManager : public virtual IService
        {
        public:
            /// Entry point is already subscribed.
            class AlreadySubscribed : public Smp::Exception
            {
            public:
                /// Name of event the entry point is already subscribed to.
                String8 eventName;
                /// Entry point that is already subscribed.
                const IEntryPoint* entryPoint;

                AlreadySubscribed(
                    String8 _eventName,
                    const IEntryPoint* _entryPoint) throw() :
                Smp::Exception("AlreadySubscribed"),
                    eventName(_eventName),
                    entryPoint(_entryPoint) { ... }
            };

            /// Entry point is not subscribed.
            class NotSubscribed : public Smp::Exception
            {
            public:
                /// Name of event the entry point is not subscribed to.
                String8 eventName;
                /// Entry point that is not subscribed.
                const IEntryPoint* entryPoint;

                NotSubscribed(
                    String8 _eventName,
                    const IEntryPoint* _entryPoint) throw() :
                Smp::Exception("NotSubscribed"),
                    eventName(_eventName),
                    entryPoint(_entryPoint) { ... }
            };

            /// Get event identifier.
            virtual EventId GetEventId(String8 eventName) = 0;

            /// Subscribe entry point.
            virtual void Subscribe(
                const EventId event,
                const IEntryPoint *entryPoint) throw (
                Smp::Services::InvalidEventId,
                Smp::Services::IEventManager::AlreadySubscribed) = 0;

            /// Unsubscribe entry point.
            virtual void Unsubscribe(
                const EventId event,
                const IEntryPoint *entryPoint) throw (
                Smp::Services::InvalidEventId,
                Smp::Services::IEventManager::NotSubscribed) = 0;

            /// Emit a global event.
            virtual void Emit(const EventId event) throw (
                Smp::Services::InvalidEventId) = 0;
        };
    }
}
```

Components can register entry points with events, define and emit events.

© EUROPEAN SPACE AGENCY, 2005

4.1.4.2 Predefined Event Kinds

```
namespace Smp
{
    namespace Services
    {
        const String8 SMP_LeaveConnecting    = "Smp_LeaveConnecting";
        const String8 SMP_EnterInitialising  = "Smp_EnterInitialising";
        const String8 SMP_LeaveInitialising  = "Smp_LeaveInitialising";
        const String8 SMP_EnterStandby       = "Smp_EnterStandby";
        const String8 SMP_LeaveStandby       = "Smp_LeaveStandby";
        const String8 SMP_EnterExecuting     = "Smp_EnterExecuting";
        const String8 SMP_LeaveExecuting     = "Smp_LeaveExecuting";
        const String8 SMP_EnterStoring       = "Smp_EnterStoring";
        const String8 SMP_LeaveStoring       = "Smp_LeaveStoring";
        const String8 SMP_EnterRestoring     = "Smp_EnterRestoring";
        const String8 SMP_LeaveRestoring     = "Smp_LeaveRestoring";
        const String8 SMP_EnterExiting       = "Smp_EnterExiting";
        const String8 SMP_EnterAborting      = "Smp_EnterAborting";

        const String8 SMP_EpochTimeChanged   = "Smp_EpochTimeChanged";
        const String8 SMP_MissionTimeChanged = "Smp_MissionTimeChanged";

        const EventId SMP_LeaveConnectingId    =  1; ///< Leave Connecting state.
        const EventId SMP_EnterInitialisingId  =  2; ///< Enter Initialising state.
        const EventId SMP_LeaveInitialisingId  =  3; ///< Leave Initialising state.
        const EventId SMP_EnterStandbyId       =  4; ///< Enter Standby state.
        const EventId SMP_LeaveStandbyId       =  5; ///< Leave Standby state.
        const EventId SMP_EnterExecutingId     =  6; ///< Enter Executing state.
        const EventId SMP_LeaveExecutingId     =  7; ///< Leave Executing state.
        const EventId SMP_EnterStoringId       =  8; ///< Enter Storing state.
        const EventId SMP_LeaveStoringId       =  9; ///< Leave Storing state.
        const EventId SMP_EnterRestoringId     = 10; ///< Enter Restoring state.
        const EventId SMP_LeaveRestoringId     = 11; ///< Leave Restoring state.
        const EventId SMP_EnterExitingId       = 12; ///< Enter Exiting state.
        const EventId SMP_EnterAbortingId      = 13; ///< Enter Aborting state.

        const EventId SMP_EpochTimeChangedId   = 14; ///< Epoch time has changed.
        const EventId SMP_MissionTimeChangedId = 15; ///< Mission time has changed.
    }
}
```

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.

4.2 Optional Services

An SMP2 compliant simulation environment *may* support the following mechanisms and services.

Note: This section lists optional services a simulation environment might choose to provide. These services are optional. However, if the simulation environment chooses to implement a functionality described in this section, then it *shall* use the specified semantics and the corresponding mapping into the target platform.

4.2.1 Resolver

Components can use the Resolver to resolve named references to components. References can either be specified using a fully qualified path, or using a path relative to some other component.

4.2.1.1 IResolver

```
namespace Smp
{
    namespace Services
    {
        /// Name of Resolver service.
        const String8 SMP_Resolver = "Smp_Resolver";

        /// This interface gives access to the Resolver.
        class IResolver : public virtual IService
        {
        public:
            /// Resolve reference to component via absolute path.
            virtual IComponent* ResolveAbsolute(String8 absolutePath) = 0;

            /// Resolve reference to component via relative path.
            virtual IComponent* ResolveRelative(
                String8 relativePath,
                IComponent* sender) = 0;
        };
    }
}
```

Components can use the Resolver to resolve references to other components by name.

 © EUROPEAN SPACE AGENCY, 2005

5. METAMODEL

This section describes the mapping of all relevant SMP2 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 into ISO/ANSI C++ code in a unique way.

5.1 Overview

The C++ mapping of the SMP2 Metamodel produces code fragments, complete files or directories depending on the kind of Element mapped. The following description of the SMP2 Metamodel mapping uses a template like notation to describe the code fragments or files in a generic way.

5.1.1 Placeholders

Non-general information coming out of the Catalogue, like names or types, is mapped by placeholders. Placeholders are encased with the '$' symbol. As an example the placeholder `$Component.Name$` shall be replaced by the actual name of the component.

5.1.2 Coloring and Font Schema

The coloring schema of the mapping description uses blue for C++ keywords, black for the rest of the compiled 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;
```

5.1.3 Generation of type Identification

The expression `TypeName()` delivers the fully qualified name of a type. Used in header files the qualified name will be the type name plus preceding nesting namespaces.

Example:

```
TypeName($Operation.Type$)
```

with an operation of return type `MyType` defined in a nested namespace of a namespace may be replaced in a header file by:

```
::Namespace1::NestedNamespace2::MyType
```

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 will be 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.

5.1.4 Optional and Selectable Code

Optional code is encased in brackets ('[' and ']').

Selectable code fragments are separated by the '|' separator.

© EUROPEAN SPACE AGENCY, 2005

Example:

```
// If embedded in a Class or Component set Visibility
[public:|protected:|private:]
```

This means that a visibility keyword may be set and one of the three possibilities can be chosen.

5.2 Core Elements

5.2.1 Simple Types

5.2.1.1 Identifier

The `Identifier`, respectively the `Id` attribute is not mapped into C++.

5.2.1.2 Name

The `Name` attribute is used for the name of a C++ equivalent of a mapped SMDL Type. Therefore C++ fields, operations, types such as structures and classes, and namespaces will be named according to the `Name` attribute of the corresponding SMDL element in a catalogue.

5.2.1.3 Description

The `Description` attribute is not mapped to C++, but may be generated as a comment in the description of a C++ equivalent of a mapped SMDL element.

5.2.1.4 UUID

The `UUID` attribute is not mapped to C++, but may be generated as a comment in the description of a C++ equivalent of a mapped SMDL Type.

5.2.2 XML Links

XML Links are used for XML files generated by SMDL. They are not mapped to C++.

5.2.3 Elements

5.2.3.1 Element

An `Element` is not mapped directly into C++, because it is not a stand-alone type, but a base type.

5.2.3.2 Named Element

A `NamedElement` is not mapped directly into C++, because it is not a stand-alone type, but a base type. Nevertheless stand-alone child types of a `NamedElement` will use the inherited `Name` attribute for the naming of their C++ equivalent (see 5.2.1.2 Name). The `Description` attribute is not mapped to C++, but may be generated as a comment (see 5.2.1.3 Description).

5.2.3.3 Document

The document class is used for XML files generated by SMDL. It is not mapped to C++.

5.2.4 Metadata

Metadata is not mapped to C++, as it is only used for tools supporting the SMDL Model Development or Model Integration processes.

© EUROPEAN SPACE AGENCY, 2005

5.3 Core Types

5.3.1 Types

5.3.1.1 Visibility Element

A VisibilityElement is not mapped directly into C++, because it is not a stand-alone type, but a base type. Nevertheless stand-alone child types of a VisibilityElement will map the inherited Visibility attribute to define their C++ access control as follows:

Table 5-1: Mapping of the Visibility attribute to ISO/ANSI C++ Access Control

Visibility attribute	C++ Access Control
public	public
protected	protected
private	private
package	public

5.3.1.2 Type

A Type is not mapped directly into C++, because it is not a stand-alone type, but a base type.

5.3.1.3 Language Type

A LanguageType is not mapped directly into C++, because it is not a stand-alone type, but a base type.

5.3.1.4 Value Type

A ValueType is either mapped as a typedef to a SimpleType, as a C++ struct, or as an enum. See 5.3.2 (Value Types) for details.

5.3.1.5 Value Reference

A ValueReference is mapped as a typedef of a pointer to a ValueType

In case the Reference belongs to a Model (i.e. is a nested type), the Visibility is defined.

```
// If embedded in a Model set Visibility
[public:|protected:|private:]

typedef TypeName($ValueReference.Type$)* $ValueReference.Name$;
```

Examples:

```
// Reference to a Simple Type
public:
    typedef Smp::UInt8* RUInt8;

// Reference to an Integer Type named MyInteger
private:
    typedef MyInteger* RInteger;
```

5.3.2 Value Types

5.3.2.1 Primitive Type

All primitive types are mapped to basic ISO/ANSI C++ types, as shown in section 2.2.1.

5.3.2.2 Enumeration

An `Enumeration` is mapped as an ISO/ANSI C++ `enum` type. Each `EnumerationLiteral` is mapped as C++ `enum` literal with value assignment. In addition, a static function is defined that registers the enumeration and all its literals in the type registry.

In case the `Enumeration` belongs to a `Model` (i.e. is a nested type), the `Visibility` is defined. In this case, the static function becomes a static method of the model with the same visibility as the type.

```
// If embedded in a Model set Visibility
[public:|protected:|private:]

enum $Enum.Name$
{
    $Enum.Literal[1].Name$ = $Enum.Literal[1].Value$,
    $Enum.Literal[2].Name$ = $Enum.Literal[2].Value$,
    ...
};

static void _Register_$Enum.Name$(Smp::Publication::ITypeRegistry* registry)
{
    /// Register type in type registry
    ...
}
```

5.3.2.3 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` and `Minimum` attributes are not explicitly mapped, but can be shown as comments in the code. In addition, a static function is defined that registers the integer type with its limits in the type registry.

In case the `Integer` belongs to a `Model` (i.e. is a nested type), the `Visibility` is defined. In this case, the static function becomes a static method of the model with the same visibility as the type.

```
// If embedded in a Model set Visibility
[public:|protected:|private:]

typedef [$Integer.Type$|Smp::Int32] $Integer.Name$;

static void _Register_$Integer.Name$(Smp::Publication::ITypeRegistry* registry)
{
    /// Register type in type registry
    ...
}
```

5.3.2.4 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 static function is defined that registers the float type with its limits in the type registry.

© EUROPEAN SPACE AGENCY, 2005

In case the `Float` belongs to a `Model` (i.e. is a nested type), the `Visibility` is defined. In this case, the static function becomes a static method of the model with the same visibility as the type.

```
// If embedded in a Model set Visibility
[public:|protected:|private:]

typedef [$Float.Type$|Smp::Float64] $Float.Name$;

static void _Register_$Float.Name$(Smp::Publication::ITypeRegistry* registry)
{
    /// Register type in type registry
    ...
}
```

5.3.2.5 Array

An `Array` type is mapped as an ISO/ANSI C++ `struct`, not as a `typedef` of an `array`. This is because an array cannot be used as return type of an operation in C++. In addition, a static function is defined that registers the array type in the type registry.

In case the `Array` belongs to a `Model` (i.e. is a nested type), the `Visibility` is defined. In this case, the static function becomes a static method of the model with the same visibility as the type.

```
// If embedded in a Model set Visibility
[public:|protected:|private:]

struct $Array.Name$
{
       TypeName($Array.ItemType$) internalArray[$Array.Size$];
};

static void _Register_$Array.Name$(Smp::Publication::ITypeRegistry* registry)
{
    /// Register type in type registry
    ...
}
```

5.3.2.6 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 static function is defined that registers the string type in the type registry.

In case the `String` belongs to a `Model` (i.e. is a nested type), the `Visibility` is defined. In this case, the static function becomes a static method of the model with the same visibility as the type.

```
// If embedded in a Class or Component set Visibility
[public:|protected:|private:]

struct $String.Name$
{
       Smp::Char8 internalString[$String.Length$+1];
};

static void _Register_$String.Name$(Smp::Publication::ITypeRegistry* registry)
{
    /// Register type in type registry
    ...
}
```

5.3.2.7 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 `Model` (i.e. is a nested type), the `Visibility` is defined.

```
// If embedded in a Model set Visibility
[public:|protected:|private:]

struct $Structure.Name$
{
        // Define the Fields of the Structure here
        See 5.3.3.1 (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 Model set Visibility
[public:|protected:|private:]

struct $Structure.Name$;
```

5.3.3 Typed Elements

5.3.3.1 Field

A `Field` is mapped to C++ as a member variable of a C++ `struct` or a C++ `class`.

- In case the `Field` belongs to a `Class` or a `Model` the visibility is mapped.
- In case the `Field` belongs to a `Structure` the visibility is assumed to be public

```
// In case the field belongs to a Class or a Model set Visibility
[public:|protected:|private:]

// field of type value type.
TypeName($Field.Type$) $Field.Name$;
```

5.3.3.2 Operation

An `Operation` that is not an `Operator` is mapped as an ISO/ANSI C++ method, while an `Operator` is mapped as an ISO/ANSI C++ `operator`.

- If the `Operation` belongs to a `ReferenceType` (`Interface` or `Model`), it is declared `virtual` in order to allow overwriting it.
- If the `Operation` belongs to an `Interface`, it will be declared as pure virtual (“`=0`”).
- If the `Operation` belongs to a `Class` or `Model`, the specified `Visibility` is defined. For `Interface` and `Structure`, the `Visibility` is always `public`.
- If the `Operation` returns a `ReferenceType`, it will return by pointer (“`*`”), otherwise it will return by value.

© EUROPEAN SPACE AGENCY, 2005

```
// In case the Operation is embedded in a Class or Model
// the given Visibility will be set. Otherwise it will be public.
[public:|protected:|private:]

// In case the Operation is of type ReferenceType, it will return by pointer
// otherwise it will return by value.
// If the Operation is an Operator, it will be mapped to an operator.
[virtual] TypeName($Operation.Type$)[*] [operator]
                    $Operation.Name$(Parameter)[=0];
```

For the Parameter list, see 5.3.3.2.1 (Parameter).

5.3.3.2.1 Parameter

A Parameter is mapped to an argument of a C++ method or operator.

- An empty parameter list 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.

Table 5-2: Type Modifier depending on type and direction

Direction	In	Out	InOut
ValueType	const	*	*
ValueReference			
ReferenceType	const &	*	*

```
// For each Parameter:
[const] TypeName($Parameter.Type$)[&][*] $Parameter.Name$
```

5.3.4 Values

Default values of fields are mapped into C++ using the appropriate mechanisms to initialise fields in a class or structure. Their mapping into C++ is explained below.

5.3.4.1 Value

This base class is not directly mapped to C++.

5.3.4.2 Simple Value

A SimpleValue maps to a corresponding value in C++ (short, double, char, etc.).

Example:

```
Smp::Int32 myInt32Field = 123;
Smp::Char8 myChar8Field = 'x';
```

A SimpleValue for an enumeration (which is stored as Smp::Int32) maps to the Name of the EnumerationLiteral with the given Value.

Example:

```
enum MyEnum
{
```

```
    literal1 = 0,
    literal2 = 1
};

MyEnum myEnumField = literal1;
```

5.3.4.3 Array Value

An `ArrayValue` maps to an array of values.

Example:

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

MyPosition myArrayField = {{1.0, 2.0, 3.0}};
```

5.3.4.4 String Value

A `StringValue` maps to a string of characters (char).

Example:

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

MyString myStringField = {"Hello World"};
```

5.3.4.5 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'};
```

5.3.4.5.1 Field Value

A `FieldValue` maps to a value of a field of a structure, class or model, which consists of the name, the assignment operator ("="), and the value. See Structure Value above for an example for a structure field.

5.3.5 Attributes

`Attributes` are not mapped to C++.

 © EUROPEAN SPACE AGENCY, 2005

5.4 SMDL Catalogues

5.4.1 A Catalogue Document

5.4.1.1 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 elements in the catalogue.

5.4.1.2 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
}
```

To avoid problems with circular includes of files for the definition of composed types, it may be necessary to add forward declarations of `Structures`, `Classes`, `Interfaces` and `Models`. 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.

5.4.2 Classes

5.4.2.1 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 is contained in a namespaces the Namespaces will be
// opened before the class definition and closed afterward.
[namespace $namespace_1.Name$ {]

	// If the Class is nested in a Model, the visibility is set.
[public:|protected:|private:]

	// If the Class has a Base Class it will be inherited virtual and public.
	class $Class.Name$ [: virtual public $Class.BaseClass.Name$]
	{
		// The Constructor and virtual Destructor are declared with the
		// Visibility of the Class
	public:|protected:|private:
		$Class.Name$ (void);
		virtual ~$Class.Name$ (void);

		// If the Class has Fields, these will be declared:
		See 5.3.3.1 (Field)

		// If the Class has Operations, these will be declared virtual:
		See 5.3.3.2 (Operation)

		// If the Class has associations, these will be declared:
		See 5.4.2.3 (Association)

		// If the Class has Properties, these will be declared virtual:
		See 5.4.2.2 (Property)
```

```
            // Register type in type registry
            static void _Register(Smp::Publication::ITypeRegistry* registry);
      };
[}]
```

5.4.2.2 Property

A `Property` is mapped as one or two ISO/ANSI C++ methods: One for a possible setter and one for a possible getter.

- If the `Property` belongs to an `Interface`, `Class` or `Model`, it is declared `virtual` in order to allow overwriting it.
- If the `Property` belongs to an `Interface`, it will be declared as pure virtual (“`=0`”).
- If the `Property` belongs to a `Class` or `Model`, the specified `Visibility` is defined. For `Interface`, the `Visibility` is always `public`.
- If the `Property` returns a `ReferenceType`, it will return by pointer (“`*`”), otherwise it will return by value.

```
// In case the Property is embedded in a Class or Model
// the given Visibility will be set. Otherwise it will be declared public.
[public:|protected:|private:]

// In case the Property is ReadOnly or ReadWrite:
[virtual] TypeName($Property.Type$)[*] get_$Property.Name$()[=0];

// In case the Property is WriteOnly or ReadWrite:
[virtual] void set_$Property.Name$(TypeName($Property.Type$)[*] value)[=0];
```

***Note**: In case a Property has an attached field, a code generator may generate an inline definition in a header file giving access to this field.*

```
// In case the Property is embedded in a Class or Model the given Visibility
// will be set.
[public:|protected:|private:]

// In case the Property is ReadOnly or ReadWrite:
[virtual] TypeName($Property.Type$)[*] get_$Property.Name$()
{
      return $Property.AttachedField$;
}

// In case the Property is WriteOnly or ReadWrite:
      [virtual] void set_$Property.Name$(TypeName($Property.Type$)[*] value)
{
      $Property.AttachedField$ = value;
}
```

5.4.2.3 Association

An `Association` is mapped to C++ as a member variable of a C++ `class`.

- In case the `Association` belongs to a `Class` or a `Model` the visibility is mapped.
- In case the `Association` maps to a `ReferenceType`, it will be defined as a pointer (“`*`”).

```
// Set Visibility
```

 © EUROPEAN SPACE AGENCY, 2005

```
[public:|protected:|private:]

// Association of type reference.
TypeName($Association.Type$)[*] $Association.Name$;
```

5.4.3 Reference Types

5.4.3.1 Reference Type

A `ReferenceType` is mapped as a C++ `class`.

5.4.3.2 Interface

An `Interface` is mapped as an abstract C++ class, which means that every C++ method for any `Operation` or `Property` of the `Interface` is declared pure virtual.

```
// An Interface is always contained in a Namespaces that will
// be opened before the interface definition and closed afterward.
namespace $namespace_1.Name$
{
        // In case the Interface has Base Interfaces these will be inherited
        // virtual and public.
        class $Interface.Name$
                [: virtual public $Interface.BaseInterface_1.Name$,...]
        {
                // If the Interface has Operations, these will be declared virtual
                See 5.3.3.2 (Operation)

                // If the Interface has Properties, these will be declared virtual
                See 5.4.2.2 (Property)
        };
}
```

5.4.3.3 Model

A `Model` is mapped as a C++ `class`. For basic SMP2 compliance, the model is mapped as follows:

- The `Model` inherits the `IModel` interface.
- If the `Model` has containers, is inherits the `IComposite` interface.
- If the `Model` provides interfaces, it will inherit these interfaces.
- If the `Model` has a base model, it inherits its implementation.

For full SMP2 compliance, the following holds in addition:

- The `Model` inherits the `IManagedModel` interface.
- If the `Model` has references, is inherits the `IAggregate` interface.
- If the `Model` has event sources, it inherits the `IEventProvider` Interface.
- If the `Model` has event sinks, it inherits the `IEventConsumer` Interface.
- If the `Model` has entry points, it inherits the `IEntryPointPublisher` Interface.

```
// The namespaces the model is contained in is
// opened at the beginning of the definition and closed afterward:
namespace $namespace_1.Name$
{
	// Inheritance:
	// If the Model has a Base Class this is inherited virtual and public.
	// If the Model provides Interfaces these will be inherited
	// virtual and public.
	class $Component.Name$:
		virtual public Smp::IModel |
		virtual public Smp::Management::IManagedModel
		[, 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 $Model.Base.Name$]
		[, virtual public $Model.ProvidedInterface_1.Name$,...]
	{
		// In case the Model has nested types these are defined:
		See 5.3.2 (Value Types)

		// The Constructor and virtual Destructor are declared with the
		// Visibility of the Model
	public:|protected:|private:
		$Model.Name$ (void);
		virtual ~$Model.Name$ (void);

		// If the Model has Operations, these are declared virtual:
		See 5.3.3.2 (Operation)

		// If the Model has Fields, these are declared:
		See 5.3.3.1 (Field)

		// If the Model has associations, these are declared:
		See 5.4.2.3 (Association)

		// If the Model has Properties, these are declared virtual:
		See 5.4.2.2 (Property)

		// If the Model has Entry Points, they will be declared:
		See 5.4.3.3.1 (Entry Point)

		// If the Model has Event Sources, they will be declared:
		See 5.4.4.2 (Event Source)

		// If the Model has Event Sinks, they will be declared:
		See 5.4.4.3 (Event Sink)
	};
}
```


© EUROPEAN SPACE AGENCY, 2005

5.4.3.3.1 Entry Point

An `Entry Point` is mapped to the `IEntryPoint` interface. The model has to ensure that an implementation is available when other components call the handler of the entry point. The `Entry Point` is mapped public.

```
public:
    Smp::IEntryPoint* $EntryPoint.Name$;
```

5.4.3.3.2 Container

A `Container` is mapped to the `IContainer` interface. The model has to ensure that an implementation is available when other components access the container. The `Container` is mapped public.

```
public:
    Smp::IContainer* $Container.Name$;
```

5.4.3.3.3 Reference Collection

A `Reference` pointing to a `ReferenceCollection` is mapped to the `IReference` interface. The model has to ensure that an implementation is available when other components access the reference. The `Reference` is mapped public.

```
public:
    Smp::IReference* $Reference.Name$;
```

5.4.4 Events

5.4.4.1 Event Type

An event type itself is not mapped to C++, as the event notification mechanism uses an untyped argument to pass a value with each event. However, the event type is used to ensure that event sinks and event sources are only connected if they are of the same event type.

5.4.4.2 Event Source

An `EventSource` is mapped to the `IEventSource` interface. The model has to ensure that an implementation is available when other components access the event source. The `EventSource` is mapped public.

```
public:
    Smp::IEventSource* $EventSource.Name$;
```

5.4.4.3 Event Sink

An `EventSink` is mapped to the `IEventSink` interface. The model has to ensure that an implementation is available when other components access the event sink. The `EventSink` is mapped public.

```
public:
    Smp::IEventSink* $EventSink.Name$;
```

5.5 SMDL Packages

5.5.1 A Package Document

5.5.1.1 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 can 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::IDynamicSimulator* 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.

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.

```
#ifdef  WIN32
#define DLL_EXPORT __declspec(dllexport)
#else
#define DLL_EXPORT
#endif

extern "C"
{
    /// Initialise function for dynamic package.
    DLL_EXPORT bool Initialise(
        Smp::IDynamicSimulator* simulator,
        Smp::Publication::ITypeRegistry* typeRegistry);

    /// Finalise function for dynamic package.
    DLL_EXPORT bool Finalise();
}
```

Note that binary distribution of models is typically only possible when using the identical OS as well as the identical C++ compiler for all models as well as for the simulation environment.

 © EUROPEAN SPACE AGENCY, 2005

5.5.1.2 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.

When the `Implementation` points to a `Model` (via its `Type` link), a corresponding class factory has to be registered with the dynamic simulator (`IDynamicSimulator`) 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`.

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`) of the type.

© EUROPEAN SPACE AGENCY, 2005

This Page is Intentionally left Blank

© EUROPEAN SPACE AGENCY, 2005

6. PUBLICATION

SMP2 Models can publish their fields, operations and properties to a publication receiver (typically the simulation environment) 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 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.

6.1 Type Registry

The Simulation Environment has to provide a single Type Registry that provides the following operations to the models:

- A set of `Add...()` operations to register user-defined types.

- The `GetType()` operation to query for already registered types.

A model has to register the types of its fields, operations, parameters and properties before it can publish these features. 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. In case two models have fields of the same type, the type registry will take care of avoiding a double registration.

The type registry provides the following operations to add types to it:

`AddFloat` Registers a user-defined `Float` taking the minimum, maximum, the inclusive flags and the unit name as additional publication attributes.

`AddInteger` Registers a user-defined `Integer` taking the minimum and maximum values as additional publication attributes.

`AddEnumeration` 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.

`AddArray` Registers a user-defined `Array`, taking the item type, item size and the array size as additional publication attributes.

`AddString` Registers a user-defined `String`, taking the string size as additional publication attribute.

`AddStructure` Registers a user-defined `Structure`. The fields of the structure have to be added in subsequent calls to the returned `IStructureType` interface.

`AddClass` 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.

6.1.1 IType

The `IType` interface is the base interface for all types that can be registered in the type registry. It is returned when registering a `Float`, `Integer`, `Array` or `String` type. The `IType` interface provides a method to query the Universally Unique Identifier (UUID) for the type (`GetUuid()`), and a function to publish a field of this type (`Publish()`) against a publication receiver. As the pre-defined types are represented by instances of `IType` in the type registry as well, the interface provides a convenience method that returns the simple type represented by the type, or `ST_None` for user-defined types.

```
namespace Smp
{
    // Forward declaration for circular references.
    class IPublication;

    namespace Publication
    {
        /// This base interface defines a type in the type registry.
        class IType : public virtual Smp::IObject
        {
        public:
            /// Get simple type that this type describes.
            virtual Smp::SimpleTypeKind GetSimpleType() const = 0;

            /// Get Universally Unique Identifier of type.
            virtual const Smp::Uuid GetUuid() const = 0;

            /// Publish an instance of the type against a receiver.
            virtual void Publish(
                IPublication *receiver,
                Smp::String8 name,
                Smp::String8 description,
                void *address,
                const Smp::Bool view,
                const Smp::Bool state,
                const Smp::Bool input,
                const Smp::Bool output) = 0;
        };
    }
}
```

Inheritance Diagram:

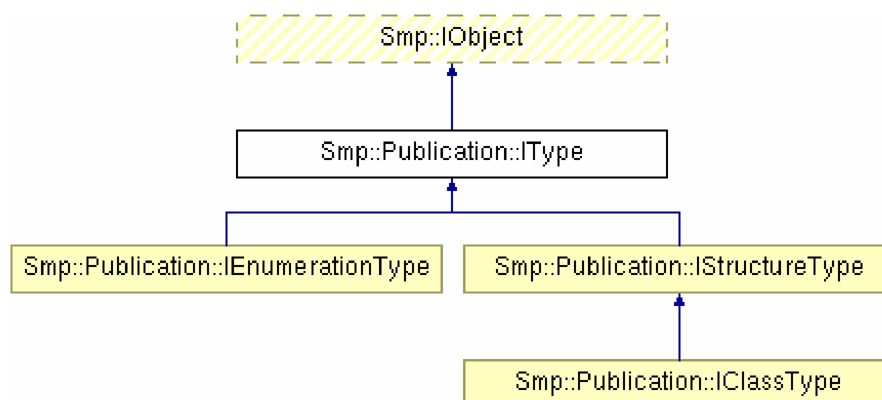

Figure 6-1: IType Interface

© EUROPEAN SPACE AGENCY, 2005

6.1.1.1 GetSimpleType

```
Smp::SimpleTypeKind GetSimpleType() const;
```

Returns the simple type represented by the type.

Parameters:

None.

Returns:

This method returns the simple type that is represented by the type, or ST_None if the type does not represent a simple type, but a user-defined type.

Exceptions:

None.

Remarks:

None.

6.1.1.2 GetUuid

```
const Smp::Uuid GetUuid() const;
```

Returns the Universally Unique Identifier of the type.

Parameters:

None.

Returns:

The Universally Unique Identifier of the type.

Exceptions:

None.

Remarks:

None.

6.1.1.3 Publish

```
virtual void Smp::Publish(
    Smp::IPublication* receiver,
    Smp::String8 name,
    Smp::String8 description,
    void* address,
    const Smp::Bool view,
    const Smp::Bool state,
    const Smp::Bool input,
    const Smp::Bool output);
```

Publishes a field of this type against a publication receiver.

Parameters:

receiver	Publication receiver to publish field against.
name	Name of the published field.
description	Description of the published field.
address	Address of the published field.
view	Defines whether the field shall be visible in the simulation environment.
state	Defines whether the field shall be stored and restored by the simulation environment.
input	Defines whether the field is an input field.
output	Defines whether the field is an output field.

Returns:

Void.

Exceptions:

None.

Remarks:

None.

6.1.2 IEnumerationType

The `IEnumerationType` interface inherits from the `IType` interface. It is returned after registering an enumeration type and provides a function to specify the literals of the registered enumeration type. As publication attributes the name, the description and the value of each literal have to be specified.

```
namespace Smp
{
    namespace Publication
    {
        /// This interface defines a user defined enumeration.
        class IEnumerationType : public virtual IType
        {
        public:
            virtual void AddLiteral(
                Smp::String8 name,
                Smp::String8 description,
                const Smp::Int32 value) = 0;
        };
    }
}
```

Inheritance Diagram:

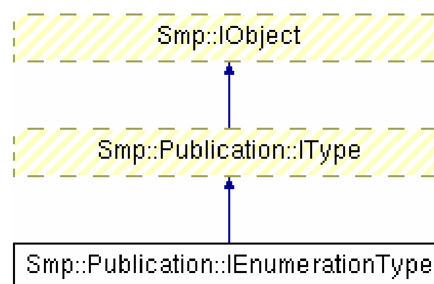

Figure 6-2: IEnumerationType Interface

 © EUROPEAN SPACE AGENCY, 2005

6.1.2.1 AddLiteral

```
void AddLiteral(Smp::String8 name,
                Smp::String8 description,
                const Smp::Int32 value);
```

Add a literal to a registered enumeration type.

Parameters:

name	Name of the published literal.
description	Description of the published literal.
value	Integer value of the published literal.

Returns:

Void.

Exceptions:

None.

Remarks:

This information allows showing enumerations using their associated literals rather than their integer values.

6.1.3 IStructureType

The `IStructureType` interface inherits from the `IType` interface. It is returned after registering a structure type and provides a function to specify the fields of the registered structure type. Before a field can be added the type of the field must be registered, and passed to the `AddField()` method.

```
namespace Smp
{
    namespace Publication
    {
        /// This interface defines a user-defined structure.
        class IStructureType : public virtual IType
        {
        public:
            virtual void AddField(
                Smp::String8 name,
                Smp::String8 description,
                const Smp::Uuid typeUuid,
                const Smp::Int64 offset,
                const Smp::Bool view = true,
                const Smp::Bool state = true,
                const Smp::Bool input = false,
                const Smp::Bool output = false) = 0;
        };
    }
}
```

Inheritance Diagram:

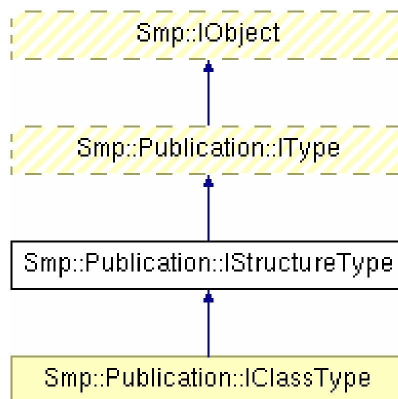

Figure 6-3: IStructureType Interface

6.1.3.1 AddField

```
void AddField(Smp::String8 name,
              Smp::String8 description,
              const Smp::Uuid typeUuid,
              const Smp::Int64 offset,
              const Smp::Bool view = true,
              const Smp::Bool state = true,
              const Smp::Bool input = false,
              const Smp::Bool output = false);
```

This method adds a field to a registered structure type.

Parameters:

name	Name of the published field.
description	Description of the published field.
typeUuid	Uuid of the type of the published field.
offset	Offset of field (in bytes) in the structure.
view	Defines whether the field shall be visible in the simulation environment.
state	Defines whether the field shall be stored by the simulation environment.
input	Defines whether the field is an input field.
output	Defines whether the field is an output field.

Returns:

Void.

Exceptions:

None.

Remarks:

None.

© EUROPEAN SPACE AGENCY, 2005

6.1.4 IClassType

The `IClassType` interface inherits from the `IStructureType` interface, but does not add new methods. It is only used to ensure that only classes can be specified as base classes in `AddClass()`.

```
namespace Smp
{
    namespace Publication
    {
        /// This interface defines a user defined class.
        class IClassType : public virtual IStructureType
        {
        };
    }
}
```

6.1.5 ITypeRegistry

This interface provides methods for the registration of user-defined types, and to find out whether a type with a given Uuid has been registered.

6.1.5.1 AlreadyRegistered

This exception is thrown when a Uuid is used that has been used for an existing type in the type registry already. This can happen as well when trying to register the same type again.

```
namespace Smp
{
    namespace Publication
    {
        class AlreadyRegistered : public Smp::Exception
        {
        public:
            /// Name of new type that cannot be registered.
            String8 name;
            /// Type that uses the same Uuid already
            IType* type;

            AlreadyRegistered(String8 _name, IType* _type);
        };
    }
}
```

6.1.5.2 NotRegistered

This exception is thrown when a Uuid is used in the publication process that has not been registered in the type registry before.

```
namespace Smp
{
    namespace Publication
    {
        class NotRegistered : public Smp::Exception
        {
        public:
            /// Uuid that does not correspond to a registered type.
            Uuid uuid;

            NotRegistered(Uuid _uuid);
        };
    }
}
```

```
namespace Smp
{
    namespace Publication
    {
        /// This interface defines a registration mechanism for user types.
        class ITypeRegistry
        {
        public:
            /// Returns a type by its simple type kind.
            virtual IType *GetType(const SimpleTypeKind type) const = 0;
            /// Returns a type by universally unique identifier.
            virtual IType *GetType(const Uuid typeUuid) const = 0;

            /// Add a float type to the registry.
            virtual const IType *AddFloatType(
                String8 name,
                String8 description,
                const Uuid typeUuid,
                const Float64 minimum,
                const Float64 maximum,
                const Bool minInclusive,
                const Bool maxInclusive,
                String8 unit,
                const SimpleTypeKind type = ST_Float64) throw (AlreadyRegistered) = 0;
            /// Add an integer type to the registry.
            virtual const IType *AddIntegerType(
                String8 name,
                String8 description,
                const Uuid typeUuid,
                const Int64 minimum,
                const Int64 maximum,
                const SimpleTypeKind type = ST_Int32) throw (AlreadyRegistered) = 0;
            /// Add an enumeration type to the registry.
            virtual IEnumerationType *AddEnumerationType(
                String8 name,
                String8 description,
                const Uuid typeUuid,
                const Int16 memorySize) throw (AlreadyRegistered) = 0;

            /// Add an array type to the registry.
            virtual const IType *AddArrayType(
                String8 name,
                String8 description,
                const Uuid typeUuid,
                const Uuid itemTypeUuid,
                const Int64 itemSize,
                const Int64 arrayCount) throw (AlreadyRegistered) = 0;

            /// Add a string type to the registry.
            virtual const IType *AddStringType(
                String8 name,
                String8 description,
                const Uuid typeUuid,
                const Int64 length) throw (AlreadyRegistered) = 0;

            /// Add a structure type to the registry.
            virtual IStructureType *AddStructureType(
                String8 name,
                String8 description,
                const Uuid typeUuid) throw (AlreadyRegistered) = 0;

            /// Add a class type to the registry.
            virtual IClassType *AddClassType(
                String8 name,
                String8 description,
                const Uuid typeUuid,
                const Uuid baseClassUuid) throw (AlreadyRegistered) = 0;
        };
    }
}
```


© EUROPEAN SPACE AGENCY, 2005

6.1.5.3 GetType for SimpleTypeKind

```
IType *GetType(const SimpleTypeKind type) const;
```

Returns a type by its simple type kind.

Parameters:

type Simple type the type is requested for.

Returns:

Interface to the requested type.

Exceptions:

None.

Remarks:

This method can be used to map simple types to the `IType` interface, to treat all types identically.

6.1.5.4 GetType for Uuid

```
IType *GetType(const Uuid typeUuid) const;
```

Returns a type by universally unique identifier.

Parameters:

typeUuid Universally unique identifier for the requested type.

Returns:

Interface of the requested type, or null if no type with the registered Uuid could be found.

Exceptions:

None.

Remarks:

This method can be used to find out whether a specific type has been registered before.

6.1.5.5 AddFloat

```
const IType *AddFloat(String8       name,
                      String8       description,
                      const Uuid    typeUuid,
                      const Float64 minimum,
                      const Float64 maximum,
                      const Bool    minInclusive,
                      const Bool    maxInclusive,
                      String8       unit,
                      const SimpleTypeKind type = ST_Float64)
      throw (AlreadyRegistered);
```

Add a float type to the registry.

© EUROPEAN SPACE AGENCY, 2005

Parameters:

name	Name of the registered type.
description	Description of the registered type.
typeUuid	Universally unique identifier of the registered type.
minimum	Minimum value for float.
maximum	Maximum value for float.
minInclusive	Flag whether the minimum value for float is valid or not.
maxInclusive	Flag whether the maximum value for float is valid or not.
unit	Unit of the type.
type	Primitive type to use for Float type.

Returns:

Interface to new type.

Exceptions:

This method throws an exception of type `AlreadyRegistered` if the typeUuid has been used by another type in the type registry already.

Remarks:

`IManagedModel` and `IDynamicInvocation` support fields, parameters and operations of `Float` types via the `ST_Float32` and `ST_Float64` simple types, as a `Float` is mapped to `Smp::Float32` or `Smp::Float64`.

6.1.5.6 AddInteger

```
const IType *AddInteger(String8 name,
                        String8 description,
                        const Uuid   typeUuid,
                        const Int64  minimum,
                        const Int64  maximum,
                        const SimpleTypeKind type = ST_Int32)
      throw (AlreadyRegistered);
```

Add an integer type to the registry.

Parameters:

name	Name of the registered type.
description	Description of the registered type.
typeUuid	Universally unique identifier of the registered type.
minimum	Minimum value for integer.
maximum	Maximum value for integer.
type	Primitive type that is used for Integer type.

Returns:

Interface to new type.

Exceptions:

This method throws an exception of type `AlreadyRegistered` if the typeUuid has been used by another type in the type registry already.

 © EUROPEAN SPACE AGENCY, 2005

Remarks:

IManagedModel and IDynamicInvocation support fields, parameters and operations of Integer types via the ST_Int?? simple type, as an Integer is mapped to one of the primitive types Smp::Int8, Smp::Int16, Smp::Int32, Smp::Int64, Smp::UInt8, Smp::UInt16 and Smp::UInt32.

6.1.5.7 AddEnumeration

```
IEnumerationType *AddEnumeration(String8 name,
                                 String8 description,
                                 const Uuid   typeUuid,
                                 const Int16  memorySize)
      throw (AlreadyRegistered);
```

Add an enumeration type to the registry.

Parameters:

name	Name of the registered type.
description	Description of the registered type.
typeUuid	Universally unique identifier of the registered type.
memorySize	Size of an instance of this enumeration in bytes. Valid values are 1, 2, 4, 8.

Returns:

Interface to new type that allows enumeration literals.

Exceptions:

This method throws an exception of type AlreadyRegistered if the typeUuid has been used by another type in the type registry already.

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 simple type, depending on their memory size.

6.1.5.8 AddArray

```
const IType *AddArray(String8 name,
                      String8 description,
                      const Uuid   typeUuid,
                      const Uuid   itemTypeUuid,
                      const Int64  itemSize,
                      const Int64  arrayCount)
      throw (AlreadyRegistered);
```

Add an array type to the registry.

Parameters:

name	Name of the registered type.
description	Description of the registered type.
typeUuid	Universally unique identifier of the registered type.
itemTypeUuid	Uuid of the type of the array items.
itemSize	Size of an array item in bytes.
arrayCount	Number of elements in the array.

© EUROPEAN SPACE AGENCY, 2005

Returns:

Interface to new type.

Exceptions:

This method throws an exception of type `AlreadyRegistered` if the typeUuid has been used by another type in the type registry already.

Remarks:

None.

6.1.5.9 AddString

```
const IType *AddString(String8 name,
                       String8 description,
                       const Uuid   typeUuid,
                       const Int64  length)
      throw (AlreadyRegistered);
```

Add a string type to the registry.

Parameters:

name	Name of the registered type.
description	Description of the registered type.
typeUuid	Universally unique identifier of the registered type.
length	Maximum length of the string.

Returns:

Interface to new type.

Exceptions:

This method throws an exception of type `AlreadyRegistered` if the typeUuid has been used by another type in the type registry already.

Remarks:

None.

6.1.5.10 AddStructure

```
IStructureType *AddStructure(String8 name,
                             String8 description,
                             const Uuid   typeUuid)
      throw (AlreadyRegistered);
```

Add a structure type to the registry.

Parameters:

name	Name of the registered type.
description	Description of the registered type.
typeUuid	Universally unique identifier of the registered type.

Returns:

Interface to new type that allows adding fields.

 © EUROPEAN SPACE AGENCY, 2005

Exceptions:

This method throws an exception of type `AlreadyRegistered` if the typeUuid has been used by another type in the type registry already.

Remarks:

None.

6.1.5.11 AddClass

```
IClassType *AddClass(String8 name,
                     String8 description,
                     const Uuid typeUuid,
                     const Uuid baseClassUuid)
      throw (AlreadyRegistered);
```

Add a class type to the registry.

Parameters:

name	Name of the registered type.
description	Description of the registered type.
typeUuid	Universally unique identifier of the registered type.
baseClassUuid	Uuid of the base type of the registered class type, or a Uuid with all fields set to 0 for none.

Returns:

Interface to new type that allows adding fields.

Exceptions:

This method throws an exception of type `AlreadyRegistered` if the typeUuid has been used by another type in the type registry already.

Remarks:

None.

6.1.6 Pre-defined Simple Types

```
namespace Smp
{
    namespace Publication
    {
        static const Uuid Uuid_Void     = {0,0,0,{ ' ',' ',' ',' ','V','o','i','d'} };
        static const Uuid Uuid_Char8    = {0,0,0,{ ' ',' ',' ','C','h','a','r','8'} };
        static const Uuid Uuid_Bool     = {0,0,0,{ ' ',' ',' ',' ','B','o','o','l'} };
        static const Uuid Uuid_Int8     = {0,0,0,{ ' ',' ',' ',' ','I','n','t','8'} };
        static const Uuid Uuid_Int16    = {0,0,0,{ ' ',' ',' ','I','n','t','1','6'} };
        static const Uuid Uuid_Int32    = {0,0,0,{ ' ',' ',' ','I','n','t','3','2'} };
        static const Uuid Uuid_Int64    = {0,0,0,{ ' ',' ',' ','I','n','t','6','4'} };
        static const Uuid Uuid_UInt8    = {0,0,0,{ ' ',' ',' ','U','I','n','t','8'} };
        static const Uuid Uuid_UInt16   = {0,0,0,{ ' ',' ','U','I','n','t','1','6'} };
        static const Uuid Uuid_UInt32   = {0,0,0,{ ' ',' ','U','I','n','t','3','2'} };
        static const Uuid Uuid_UInt64   = {0,0,0,{ ' ',' ','U','I','n','t','6','4'} };
        static const Uuid Uuid_Float32  = {0,0,0,{ ' ','F','l','o','a','t','3','2'} };
        static const Uuid Uuid_Float64  = {0,0,0,{ ' ','F','l','o','a','t','6','4'} };
        static const Uuid Uuid_DateTime = {0,0,0,{ 'D','a','t','e','T','i','m','e'} };
        static const Uuid Uuid_Duration = {0,0,0,{ 'D','u','r','a','t','i','o','n'} };
    }
}
```

For each of the pre-defined simple types, a corresponding Universally Unique Identifier has been defined. This allows to query for a simple type by its Uuid, using the `GetType()` method of the `ITypeRegistry`.

Note:

The type registry must ensure that the pre-defined simple types are registered using their pre-defined Uuids.

6.2 Publication of Fields, Operations and Properties

6.2.1 IPublication

The `IPublication` interface provides a set of functions to publish fields of simple or used-defined type, arrays, structures, and operations. Further, it supports models in implementing the `IManagedModel` and `IDynamicInvocation` interfaces by providing an implementation for most of their functions. As the `IPublication` interface is so complex, its functions are described in three sections.

6.2.1.1 Publication using the Type Registry

This section describes methods available to publish fields and operations of models by referencing types from the type registry. These methods should be used to publish features with complex types. The complex types are registered upfront in the type registry and are used in these methods by their Uuid. By using these convenience methods the effort of publishing the structure of the used complex types is taken over by the complex types themselves.

```
namespace Smp
{
    class IPublication
    {
    public:
        /// Give access to the global type registry.
        virtual Publication::ITypeRegistry* GetTypeRegistry() const = 0;

        /// Publish field of any type.
        virtual void PublishField(
            String8 name,
            String8 description,
            void* address,
            const Uuid typeUuid,
            const Bool view = true,
            const Bool state = true,
            const Bool input = false,
            const Bool output = false) throw (
            Smp::Publication::NotRegistered) = 0;

        /// Publish an operation with complex return type.
        virtual Publication::IPublishOperation* PublishOperation(
            String8 name,
            String8 description,
            const Uuid returnTypeUuid) throw (
            Smp::Publication::NotRegistered) = 0;

        /// Publish a property.
        virtual void PublishProperty(
            String8 name,
            String8 description,
            const Uuid typeUuid,
            const AccessKind accessKind) throw (
            Smp::Publication::NotRegistered) = 0;

        // Further methods are detailed below
        // ...
    }
}
```

 © EUROPEAN SPACE AGENCY, 2005

6.2.1.1.1 GetTypeRegistry

```
Publication::ITypeRegistry* GetTypeRegistry() const;
```

Provide access to the global type registry.

Parameters:

None.

Returns:

Interface to type registry.

Exceptions:

None.

Remarks:

None.

6.2.1.1.2 Publish Field

```
void PublishField(
            String8 name,
            String8 description,
            const void* address,
            const Uuid typeUuid,
            const Bool view = true,
            const Bool state = true,
            const Bool input = false,
            const Bool output = false) throw (Smp::Publication::NotRegistered);
```

Publish a field of a registered type, including its memory address.

Parameters:

name	Name of the published field.
description	Description of the published field.
address	Address of the published field.
typeUuid	Uuid of the type of the published field.
view	Defines whether the field shall be visible in the simulation environment.
state	Defines whether the field shall be stored and restored by the simulation environment.
input	Defines whether the field is an input field.
output	Defines whether the field is an output field.

Returns:

Void.

Exceptions:

This method throws an exception of type `NotRegistered` when no type has been registered with the given Uuid.

Remarks:

None.

© EUROPEAN SPACE AGENCY, 2005

6.2.1.1.3 Publish Operation

```
Publication::IPublishOperation *PublishOperation(
            String8 name,
            String8 description,
            const Uuid returnTypeUuid) throw (Smp::Publication::NotRegistered);
```

Publish an operation that has a return type of a registered type.

Parameters:

name	Name of the published operation.
description	Description of the published operation.
returnTypeUuid	Uuid of return type of the published operation.

Returns:

Interface to `IPublishOperation` to publish the parameters of the operation to.

Exceptions:

This method throws an exception of type `NotRegistered` when no type has been registered with the given Uuid.

Remarks:

None.

6.2.1.1.4 Publish Property

```
void PublishProperty(
            String8 name,
            String8 description,
            const Uuid typeUuid,
            const AccessKind accessKind) throw (Smp::Publication::NotRegistered);
```

Publish a property of a registered type, including its access kind.

Parameters:

name	Name of the published field.
description	Description of the published field.
typeUuid	Uuid of the type of the published property.
accessKind	Access kind of the published property.

Returns:

Void.

Exceptions:

This method throws an exception of type `NotRegistered` when no type has been registered with the given Uuid.

Remarks:

None.

 © EUROPEAN SPACE AGENCY, 2005

6.2.1.2 Direct Publication

```
namespace Smp
{
    /// Publication receiver.
    class IPublication
    {
    public:
        // Further methods are detailed above
        // ...

        /// Publish array of simple type.
        virtual void PublishArray(
            String8 name,
            String8 description,
            const Int64 count,
            void* address,
            const SimpleTypeKind type,
            const Bool view = true,
            const Bool state = true,
            const Bool input = false,
            const Bool output = false) = 0;

        /// Publish array of any type.
        virtual IPublication* PublishArray(
            String8 name,
            String8 description) = 0;

        /// Publish structure.
        virtual IPublication* PublishStructure(
            String8 name,
            String8 description) = 0;

         // Further methods are detailed below
         // ...
     }
}
```

This section describes the methods available to publish fields of models without referencing types from the type registry. These methods only use the pre-defined simple types (via the `SimpleTypeKind` enumeration), and mechanisms for arrays (including strings, which are arrays of characters) and structures (including classes, which are derived from structures).

These methods are typically not used by models, but by used-defined types in the type registry.

6.2.1.2.1 Publish Array of simple type

```
void PublishArray(
            String8 name,
            String8 description,
            const Int64 count,
            void* address,
            const SimpleTypeKind type,
            const Bool view = true,
            const Bool state = true,
            const Bool input = false,
            const Bool output = false);
```

Publish a field that is an array of a simple type, including its memory address.

Parameters:

name	Name of the published array field.
description	Description of the published array field.
count	Array size of the published array field.

address	Address of the published array field.
type	Type of the published array field (i.e. of each array element).
view	Defines whether the field shall be visible in the simulation environment.
state	Defines whether the field shall be stored and restored by the simulation environment.
input	Defines whether the field is an input field.
output	Defines whether the field is an output field.

Returns:

Void.

Exceptions:

None.

Remarks:

This method works for all simple type kinds except for `ST_None`, which does not name a type. Strings can be published using the simple type kind `ST_Char8`.

6.2.1.2.2 Publish Array of user-defined type

```
IPublication* PublishArray(
            String8 name,
            String8 description);
```

Publish a field that is an array of a user-defined type.

Parameters:

name	Name of the published array field.
description	Description of the published array field.

Returns:

Interface to publication receiver to publish the elements of the array to.

Exceptions:

None.

Remarks:

The elements of the array have to be published separately using the returned `IPublication` interface. This method only creates an empty node for the array.

6.2.1.2.3 Publish Structure

```
IPublication* PublishStructure(
            String8 name,
            String8 description);
```

Publish a field that is of a structure type.

Parameters:

name	Name of the published array field.
description	Description of the published array field.

 © EUROPEAN SPACE AGENCY, 2005

Returns:

Interface to publication receiver to publish the fields of the structure to.

Exceptions:

None.

Remarks:

The field of the structure have to be published separately using the returned `IPublication` interface. This method only creates an empty node for the structure.

As classes are derived from structure, they can be published with this method as well.

6.2.1.3 Overloaded Methods for fields of simple types

When using the `PublishField()` method to publish a field, a type has to be specified, together with a void pointer to the memory address of the field. This implementation is not type-safe, as the compiler cannot perform type checking. Therefore, the C++ implementation provides overloaded methods for most types, which use a typed memory address (rather than a void pointer), and omit the specification of the type (as it is known by the pointer type). However, due to the fact that `Duration` and `DateTime` are both mapped to `Int64` internally, no overloaded methods for these types can be provided.

© EUROPEAN SPACE AGENCY, 2005

This section only details one of the available methods, as they only differ in the type of the memory address. Except for DateTime and Duration (which are mapped to `Int64`), an overloaded, type-safe method exists for each primitive type.

```
namespace Smp
{
    class IPublication
    {
    public:
        // Further methods are detailed above
        // ...

        /// Publish 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(
            String8 name,
            String8 description,
            Char8* address,
            const Bool view = true,
            const Bool state = true,
            const Bool input = false,
            const Bool output = false) = 0;

        /// Publish a Bool field.
        // ... use "Bool *address" for third parameter

        /// Publish Int8 field.
        // ... use "Int8 *address" for third parameter

        /// Publish Int16 field.
        // ... use "Int16 *address" for third parameter

        /// Publish Int32 field.
        // ... use "Int32 *address" for third parameter

        /// Publish Int64 field.
        // ... use "Int64 *address" for third parameter

        /// Publish UInt8 field.
        // ... use "UInt8 *address" for third parameter

        /// Publish UInt16 field.
        // ... use "UInt16 *address" for third parameter

        /// Publish UInt32 field.
        // ... use "UInt32 *address" for third parameter

        /// Publish UInt64 field.
        // ... use "UInt64 *address" for third parameter

        /// Publish Float32 field.
        // ... use "Float32 *address" for third parameter

        /// Publish Float64 field.
        // ... use "Float64 *address" for third parameter

        // Further methods are detailed below
        // ...
    }
}8
```


© EUROPEAN SPACE AGENCY, 2005

6.2.1.3.1 Publish Field of Char8 type

```
void PublishField(
            String8 name,
            String8 description,
            Char8* address,
            const Bool view = true,
            const Bool state = true,
            const Bool input = false,
            const Bool output = false);
```

Publish a field of Char8 type, including its memory address.

Parameters:

name	Name of the published Char8 field.
description	Description of the published Char8 field.
address	Address of the published Char8 field.
view	Defines whether the field shall be visible in the simulation environment.
state	Defines whether the field shall be stored and restored by the simulation environment.
input	Defines whether the field is an input field.
output	Defines whether the field is an output field.

Returns:

Void.

Exceptions:

None.

Remarks:

As this method uses a typed pointer, it does not need to specify the type of the field.

© EUROPEAN SPACE AGENCY, 2005

6.2.1.4 Convenience Methods

The convenience methods allow an easy implementation of the `IManagedModel` and `IDynamicInvocation` interfaces. As all information needed for the implementation of these methods is published to the publication receiver, their implementation can be delegated to the publication receiver. The semantics of the four convenience methods has not changed, so the reader is referred to the corresponding interfaces for detailed descriptions of these methods.

```
namespace Smp
{
    class IPublication
    {
    public:
        // Further methods are detailed above
        // ...

        /// Get the value of a field which is typed by a system type.
        virtual AnySimple GetFieldValue(String8 fullName) = 0;

        /// Set the value of a field which is typed by a system type.
        virtual void SetFieldValue(String8 fullName, AnySimple value) = 0;

        /// Get the value of an array field which is typed by a system type.
        virtual void GetArrayValue(
            String8 fullName,
            const AnySimpleArray values,
            const Int32 length) throw (
            Smp::Management::IManagedModel::InvalidFieldName,
            Smp::Management::IManagedModel::InvalidArraySize) = 0;

        /// Set the value of an array field which is typed by a system type.
        virtual void SetArrayValue(
            String8 fullName,
            const AnySimpleArray values,
            const Int32 length) throw (
            Smp::Management::IManagedModel::InvalidFieldName,
            Smp::Management::IManagedModel::InvalidArraySize,
            Smp::Management::IManagedModel::InvalidArrayValue) = 0;

        /// Create request object.
        virtual IRequest* CreateRequest(String8 operationName) = 0;

        /// Delete request object.
        virtual void DeleteRequest(IRequest* request) = 0;
    }
}
```

 © EUROPEAN SPACE AGENCY, 2005

6.2.2 IPublishOperation

The `IPublishOperation` interface provides functions to add parameters to a previously published operation. As publication attributes the parameter name, description and type have to be specified.

```
namespace Smp
{
    namespace Publication
    {
        // Forward declaration because of circular references.
        class IType;

        /// Publish operation parameters.
        class IPublishOperation
        {
        public:
            /// Virtual destructor.
            virtual ~IPublishOperation() { }

            /// Publish a parameter of an operation.
            virtual void PublishParameter(
                String8 name,
                String8 description,
                const Smp::Uuid typeUuid) throw (
                Smp::Publication::NotRegistered) = 0;
        };
    }
}
```

6.2.2.1 Publish Parameter

```
void PublishParameter(
                  String8 name,
                  String8 description,
                  const Smp::Uuid typeUuid) throw (
                  Smp::Publication::NotRegistered);
```

Publish a parameter of an operation.

Parameters:

name	Name of the published parameter.
description	Description of the published parameter.
typeUuid	Uuid of the type of the published parameter.

Returns:

Void.

Exceptions:

This method throws an exception of type `NotRegistered` when no type has been registered with the given Uuid.

Remarks:

This method works for all types except for `Uuid_Void`, which does not identify a type.

This Page is Intentionally left Blank

© EUROPEAN SPACE AGENCY, 2005