EUROPEAN SPACE AGENCY
DIRECTORATE OF OPERATIONS AND INFRASTRUCTURE

SMP 2.0 Component Model

EGOS-SIM-GEN-TN-0101

Issue 1 Revision 2

28 October 2005

This Page is Intentionally left Blank

ABSTRACT

This technical note contains the specification of the component model for the SMP2 standard.

DOCUMENT APPROVAL

Prepared by	Organisation	Signature	Date
Peter Fritzen Peter Ellsiepen Anthony Walsh	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		

DOCUMENT STATUS SHEET

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

This Page is Intentionally left Blank

DOCUMENT CHANGE RECORD

DOCUMENT CHANGE RECORD Changes from SMP 2.0 Component Model Issue 1 Revision 1 to SMP 2.0 Component Model Issue 1 Revision 2			
DCI NO	N/A		
DATE	28 October 2005		
ORIGINATOR	SMP CCB		
APPROVED BY	Niklas Lindman		

1. PAGE	**2. PARAGRAPH**	**3. ISSUE**	**4. CHANGES MADE**
17	1.4	46	SMP Handbook and Alpha Specification moved from Applicable Documents to Reference Documents.
32	3.2.2.3	26	`ModelStateKind` enumeration added.
33	3.2.2.4	2, 26	`IModel` interface updated: `Initialize()` removed, `GetState()` and `Configure()` methods added, `InvalidModelState` exception added.
34	3.2.2.4.2	26	`InvalidModelState` exception added to `Publish()` method.
35	3.2.2.4.3	26	`Configure()` method added.
35	3.2.2.4.4	26	`InvalidModelState` exception added to `Connect()` method.
46	3.3.3.3.1	51	Explained that event sinks will be called synchronously in the order they have been added.
59	3.4.1.1.1	10	`GetOwner()` method added to `IEntryPoint`.
64	3.5.2.1	20	`Int32` replaced by `Int64` in IManagedReference.
67	3.5.2.2	21	`Int32` replaced by `Int64` in IManagedContainer.
80	3.6.1.1	2	Simulator states updated: Setup phase simplified.
82	3.6.1.2	23	`Get...()` methods added to get mandatory services without need for a type cast. `Configure()` method added.
87	3.6.1.2.12	2, 26	Semantics of `Publish()` method changed.
87	3.6.1.2.13	2, 26	`Configure()` method added.
88	3.6.1.2.15	2	`Initialize()` renamed to `Initialise()`, and semantics changed to call initialisation entry points.
91	3.6.1.2.22	-3	`AddInitEntryPoint()` method added to define entry points for `Initialising` phase.
93	3.6.1.3.1	24	Three parameters removed.
94	3.6.1.4	24	`GetSpecification()` and `GetImplemenation()` methods added .
99	4.1.1	22	`GetLogMessageKind()` method added, to allow for used-defined log message kinds.
109	4.1.3	11, 29, 53	`IScheduler` interface updated.

119	4.1.4	28	`EventId` type added. `Int32` replaced by `Int64`. `InvalidEventId` exception moved from `IEventManager` interface to `Services` namespace, as it is shared with `IScheduler` interface.
121	4.1.4.1.4	51	Explained that entry points will be called synchronously in the order they have been added.

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

1. INTRODUCTION

This document introduces the platform independent component model for the Simulation Model Portability (**SMP**) 2 standard. Every model developed for SMP2 (and every simulation service) has to follow a component based design. Therefore, this document provides the baseline for all components.

1.1 Purpose

The purpose of this document is to unambiguously define what an SMP2 component is, which components are parts of a simulation, and how optional component and model mechanisms shall be provided if available.

One of the major improvements that SMP2 brings to the Simulation Model Portability (**SMP**) standard is the concept of a component-based design. This allows breaking the functionality of a simulation down into individual components, which promotes re-use of components as well as for example easy updates of individual components without having to re-deliver a complete simulation. In SMP2, three major kinds of components are part of a running simulation: The simulation environment itself, its simulation services, and the models.

This document defines a consistent component model, which lays the foundations for each of the three component types.

1. For the **simulation environment**, all interfaces are documented.
2. For **simulation services**, the base interface as well as the interfaces to the individual simulation services are documented.
3. For **models**, the mandatory base interface is documented together with the optional component and model enhancements.
4. Finally, the generic **service acquisition mechanism** that involves all three types of components is standardised in this document.

1.2 Scope

The intended readerships of this document are tools developers, simulation environment furnishers, and possibly those wanting to provide a mapping of SMP2 to another platform. Model developers typically read the Handbook first [AD-1], and then jump to the Platform Mapping for their target platform, for example to the C++ Mapping [AD-4].

Rather than repeating the SMP2 concepts stated in the SMP2 Handbook [AD-1], this document provides the details of the SMP2 component model. It defines an Application Programmer Interface (**API**) using the platform independent Interface Definition Language (**IDL**) of the Common Object Request Broker Architecture (**CORBA**).

This document defines interfaces in a platform independent manner. It does not take implementation details for example of the C++ programming language into account. Further documents provide mappings of the Platform Independent Model (**PIM**) into a Platform Specific Model (**PSM**). See the SMP2 Handbook for details on platform independence and platform specific models [AD-1].

1.3 Definitions, acronyms and abbreviations

AD	Applicable Document
API	Application Programmer Interface
CORBA	Common Object Request Broker Architecture
ESOC	European Space Operations Centre
GMT	Greenwich Mean Time
HITL	Hardware-In-The-Loop
IDL	Interface Definition Language
JDK	Java Development Kit
MAEL	Mobile Aeronautics Education Laboratory
MJD	Modified Julian Date
N/A	Not Applicable
PIM	Platform Independent Model
PSM	Platform Specific Model
RD	Reference Document
SMDL	Simulation Model Definition Language
SMP	Simulation Model Portability
TBC	To be confirmed
TBD	To be defined
UML	Unified Modelling Language
UTC	Coordinated Universal Time

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
AD-4	SMP 2.0 C++ Mapping EGOS-SIM-GEN-TN-0102, Issue 1.2, 28-Oct-2005

1.4.2 Reference Documents

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

RD-1	Simulation Model Portability Handbook EWP-2080, Issue 1.1, 31-Oct-2000
RD-2	SMP2 Alpha Specification SIM-GST-TN-0045-TOS-GIC, Issue 1.0, 30-Dec-2003

1.5 Overview

This document introduces a platform independent component model used by SMP2 components. It introduces mandatory interfaces for components, models, services as well as additional, optional features. As this document does not define classes (i.e. an implementation), but only interfaces, all names in this document start with an upper-case "I", which stands for "Interface".

1.5.1 Objects

In order to allow harmonisation of components with other SMP2 elements, this document first defines what an object is (`IObject`). Although most elements of the SMP2 component model are components, these components expose some objects, which are derived directly from the `IObject` interface. These objects include entry points, event sources and event sinks, and containers and references. See section 3.2.1 for details on objects.

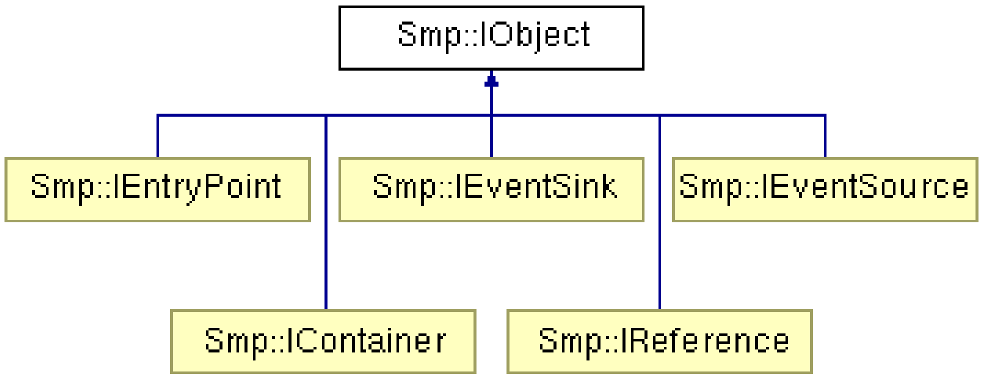

Figure 1-1: Object Types

1.5.2 Components

To allow treating all components in a similar way, it is then defined what a component is (i.e. the `IComponent` interface). The three most important interfaces derived from this base interface are `IModel` for models, `IService` for services, and `ISimulator` for the simulation environment (where `ISimulator` is not derived immediately from `IComponent`, but from `IComposite`). See section 3.2 for details on Objects and Components, and section 3.6 for details on Simulators.

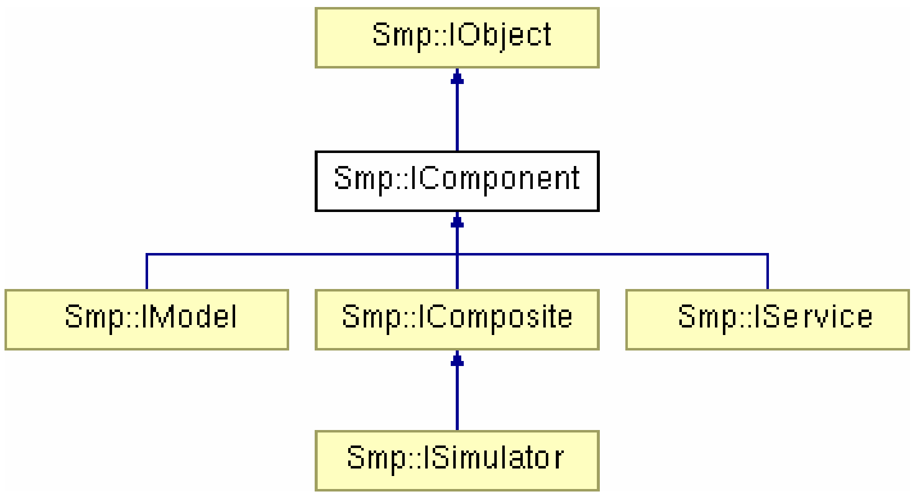

Figure 1-2: Component Types

1.5.3 Component Mechanisms

Further, SMP2 defines standard mechanisms how to enhance components, for example for component aggregation (`IAggregate`), component composition (`IComposite`), inter-component events (`IEventProvider` and `IEventConsumer`), dynamic invocation (`IDynamicInvocation`) and self-persistence (`IPersist`). See section 3.3 for details on Component Mechanisms.

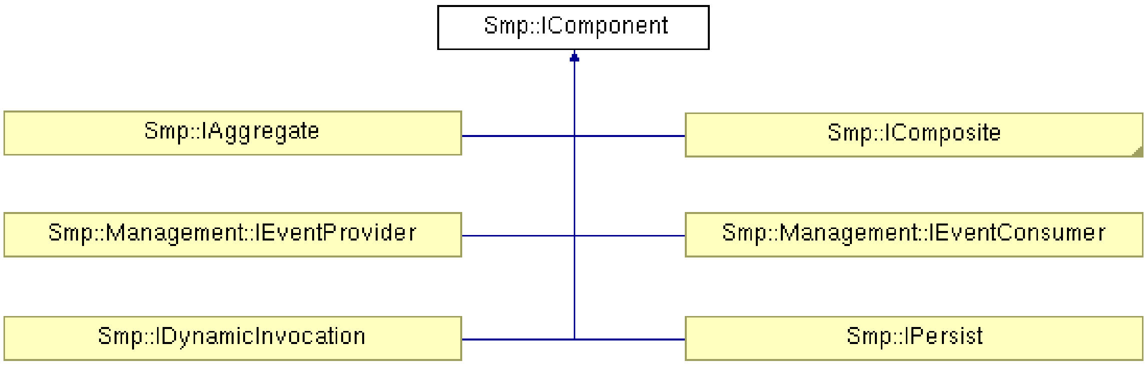

Figure 1-3: Component Mechanisms

1.5.4 Model Mechanisms

The most important component type of SMP2 is the Model. While implementing the mandatory `IModel` interface is sufficient to make a model SMP2 compliant, this does not allow using most of the SMP2 mechanisms. Therefore, optional extensions of the `IModel` interface do exist providing optional features for models that go beyond those for components. A fundamental one is that of an entry point publisher (`IEntryPointPublisher`). See section 3.4 for details on Model Mechanisms, and section 3.5.3 for Managed Model Mechanisms.

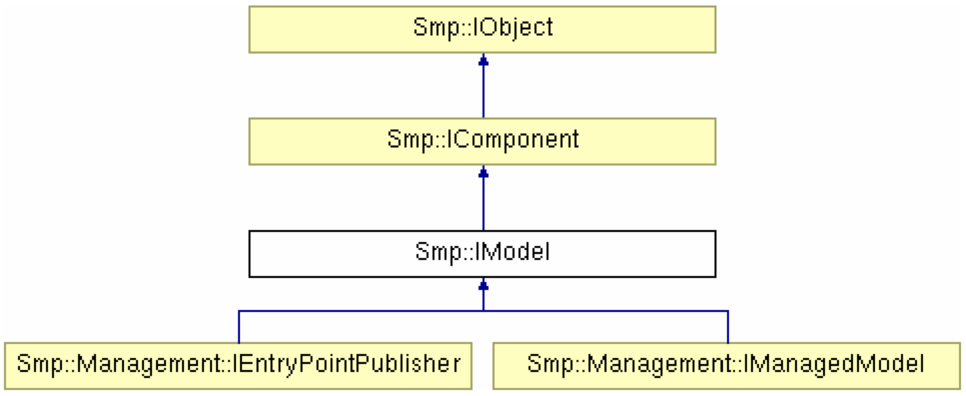
Smp::IObject
Smp::IComponent
Smp::IModel
Smp::Management::IEntryPointPublisher
Smp::Management::IManagedModel

Figure 1-4: Model Mechanisms

1.5.5 Management Interfaces

To support dynamically configured simulations, where all information about the models, their links and initial values is read from an external source (typically XML files), SMP2 defines optional mechanisms for managed components. These allow setting their properties (name, description, and parent) from external applications, and provide mechanisms to query their elements (field values, entry points, event sources and sinks) by name. See section 3.5 for details on Management Interfaces.

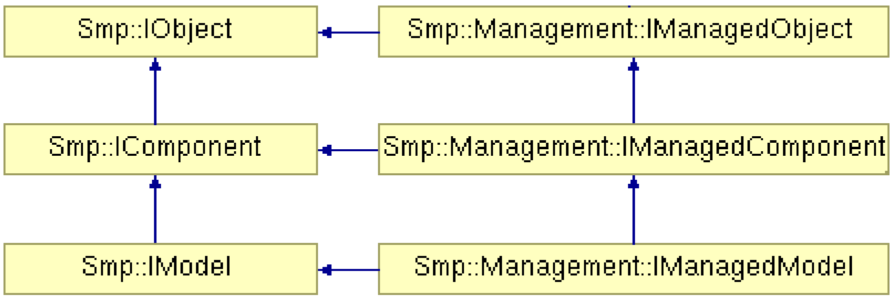
Smp::IObject
Smp::Management::IManagedObject
Smp::IComponent
Smp::Management::IManagedComponent
Smp::IModel
Smp::Management::IManagedModel

Figure 1-5: Management Interfaces

1.5.6 Simulation Environments

Finally, the Simulator or Simulation Environment is specified in this document. Its base interface `ISimulator` allows accessing both the simulation services and the models, but the simulation environment also has to provide a publication mechanism to the models. See section 3.6 for details on Simulation Environments.

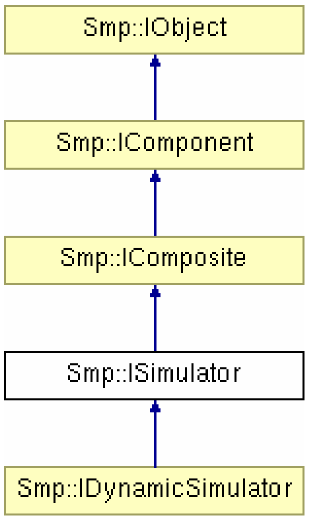

Figure 1-6: Simulation Environments

1.5.7 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. Furthermore, a mechanism of how models can acquire services is part of the standard. See section 4 for details on Simulation Services.

1.5.8 Mandatory and Optional Interfaces

This document defines mandatory and optional interfaces. A mandatory interface is one that every component *shall* support in order to inter-operate with other components (models or services). An optional interface is one that a component *may* implement. However, if the component does implement the corresponding functionality, then it shall implement the interface as defined in this specification.

1.5.9 Exceptions

SMP2 defines specific exceptions, and links them to the operations of the interfaces. All exceptions are expressed in CORBA IDL.

However, the rationale of the SMP2 Component Model is to use exceptions carefully, following these rules:

1. Exceptions are only used for *unexpected behaviour*. For expected behaviour (for example no service found with given name), the return value is used (typically a return value of null).

2. Exceptions are not used to indicate *that* an operation has failed, but to indicate *why* it failed. Most exceptions carry *useful, precise, complete* information. In some cases, the name of the exception clearly tells why an operation has failed.

3. Exceptions are typically used when different error sources exist, or when additional information about the error source (for example the expected number of parameters) is passed back to the caller.

Further exceptions may be added to the standard in later revisions based on a first reference implementation of SMP2.

This Page is Intentionally left Blank

2. PLATFORM CONSIDERATIONS

This document introduces a platform independent model of the SMP2 Component Model. However, in order to present this model, we need to use a language to describe for example data types, interfaces, inheritance, and collections. The two most appropriate options for a “platform independent” model are the Unified Modelling Language (**UML**), and the CORBA Interface Definition Language (**IDL**). We have decided to use CORBA IDL in this document, as it provides all mechanisms needed in order to specify the component model (except for a signed 8-bit integer type).

2.1 CORBA IDL

This document uses a subset of CORBA IDL to define the data types and interfaces that form the SMP2 Component Model. Individual platform mapping documents provide mappings of this component model to specific target platforms. These may coincide with the mappings CORBA IDL compilers generate for the target platform, but this is not guaranteed.

2.2 Data Types

CORBA IDL provides almost all simple types needed for SMP2, except for a signed 8-bit integer data type.

2.2.1 Simple Types

The following type definitions are used within this document.

```
module Smp
{
   /// Character type that is used as well by strings.
   typedef char                 Char8;     ///<  8 bit character type.

   /// Boolean type that is either true or false.
   typedef boolean              Bool;      ///<  Boolean with true and false.

   // Integer types
   typedef octet                Int8;      ///<  8 bit   signed integer type.
   typedef octet                UInt8;     ///<  8 bit unsigned integer type.
   typedef short                Int16;     ///< 16 bit   signed integer type.
   typedef unsigned short       UInt16;    ///< 16 bit unsigned integer type.
   typedef long                 Int32;     ///< 32 bit   signed integer type.
   typedef unsigned long        UInt32;    ///< 32 bit unsigned integer type.
   typedef long long            Int64;     ///< 64 bit   signed integer type.
   typedef unsigned long long UInt64;      ///< 64 bit unsigned integer type.

   // Floating point types
   typedef float                Float32;   ///< Single-precision float type.
   typedef double               Float64;   ///< Double-precision float type.

    // Date and time types
   typedef Int64                Duration;  ///< Duration in Nanoseconds.
   typedef Int64                DateTime;  ///< Relative to MJD2000+0.5.
};
```

These types correspond to the available primitive types defined in the SMP 2.0 Metamodel [AD-2]. By defining them in CORBA IDL, a Platform Mapping to native types is provided for the CORBA IDL Platform.

2.2.2 Simple Type Union

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. As the CORBA IDL any type is more powerful, the AnySimple type is not mapped to any, but to a discriminated union of the available simple types.

In addition, the type AnySimpleArray is defined to allow using arrays of AnySimple.

```
module 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 Boolean 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.
    };

    /// Union of simple type values
    union AnySimple switch (SimpleTypeKind)
    {
        case ST_Char8:    Char8    char8Value;    ///< Value for ST_Char8.
        case ST_Bool:     Bool     boolValue;     ///< Value for ST_Bool.

        case ST_Int8:     Int8     int8Value;     ///< Value for ST_Int8.
        case ST_UInt8:    UInt8    uint8Value;    ///< Value for ST_UInt8.
        case ST_Int16:    Int16    int16Value;    ///< Value for ST_Int16.
        case ST_UInt16:   UInt16   uint16Value;   ///< Value for ST_UInt16.
        case ST_Int32:    Int32    int32Value;    ///< Value for ST_Int32.
        case ST_UInt32:   UInt32   uint32Value;   ///< Value for ST_UInt32.
        case ST_Int64:    Int64    int64Value;    ///< Value for ST_Int64.
        case ST_UInt64:   UInt64   uint64Value;   ///< Value for ST_UInt64.

        case ST_Float32:  Float32  float32Value;  ///< Value for ST_Float32.
        case ST_Float64:  Float64  float64Value;  ///< Value for ST_Float64.

        case ST_Duration: Duration durationValue; ///< Value for ST_Duration.
        case ST_DateTime: DateTime datetimeValue; ///< Value for ST_DateTime.
    };

    /// Array of AnySimple values.
    typedef sequence<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).

```
module Smp
{
   struct Uuid
   {
      UInt32 Data1;              ///< 8 hex nibbles
      UInt16 Data2;              ///< 4 hex nibbles
      UInt16 Data3;              ///< 4 hex nibbles
      UInt8  Data4[8];           ///< 4+12 hex nibbles
   };
};
```

2.2.4 Strings

With the `string` type, CORBA IDL has a native string data type, which is used within this document as the platform mapping of String8. Strings are used as input parameters (typically a `name`, sometimes a `description`), and they can be queried by the `GetName()` and `GetDescription()` methods of the `IObject` base interface.

```
module Smp
{
   typedef string String8;   ///< String of 8 bit characters.
};
```

2.2.5 Collections

With the `typedef sequence<...>` mechanism, CORBA IDL has a native mechanism to define collections, which is used within this document to define the collections of the SMP2 Component Model.

2.3 Interfaces

CORBA IDL provides a native `interface` definition mechanism. Therefore, there is no need to introduce a special semantics of how an interface is represented in IDL.

2.4 Inheritance

As CORBA IDL only defines interfaces of components, not their implementation, its concept of inheritance is always a concept of *interface inheritance*. CORBA IDL supports multiple inheritance of interfaces, although the SMP2 Component Model makes little use of this feature (except for managed interfaces).

This Page is Intentionally left Blank

3. COMPONENT MODEL

This section details the platform independent component model of SMP2. It contains all types used within the SMP2 Component Model. This includes simple types, derived types, and interfaces with exceptions.

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.

3.1.1 InvalidObjectName

```
module Smp
{
    exception InvalidObjectName
    {
        /// Name that is not valid.
        String8 objectName;
    };
};
```

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

3.1.2 DuplicateName

```
module Smp
{
    exception DuplicateName
    {
        /// Name that already exists in the collection.
        String8 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.

3.1.3 InvalidAnyType

```
module Smp
{
    exception InvalidAnyType
    {
        /// Type that is not valid.
        SimpleTypeKind invalidType;
        /// Type that was expected.
        SimpleTypeKind expectedType;
    };
};
```

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

3.1.4 InvalidObjectType

```
module Smp
{
    exception InvalidObjectType
    {
        /// Object that is not of valid type.
        IObject invalidObject;
    };
};
```

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 SMP2 are not components, but only objects.

3.2.1 Objects

Objects are the baseline for components. They provide name and description.

3.2.1.1 IObject

```
module Smp
{
   interface IObject
   {
      String8 GetName();
      String8 GetDescription();
   };
};
```

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`.

Inheritance Diagram:

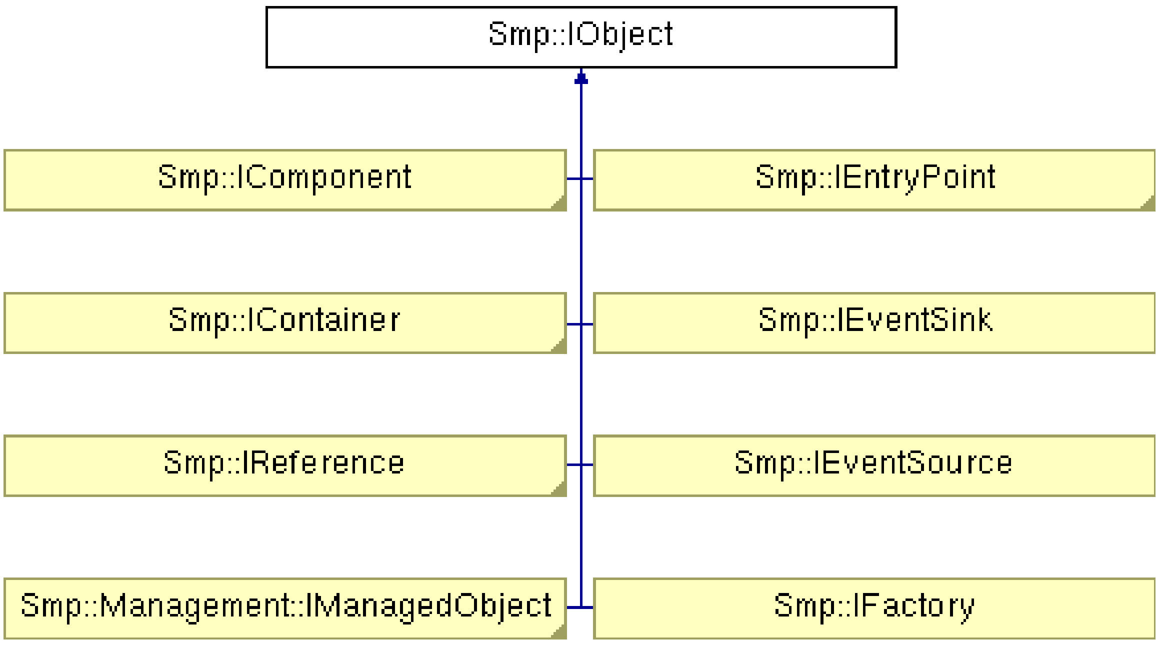

Figure 3-1: IObject Interface

Remarks:

The two methods of this interface ensure that all SMP2 objects can be shown with a name, and with an optional description.

3.2.1.1.1 GetName

```
String8 GetName();
```

This method returns the name of the object ("property getter"). Applications may display the name as user readable object identification.

Parameters:

None.

Returns:

Name of object.

Exceptions:

None.

Remarks:

Names

- must be unique within their context,
- must not be empty,
- must not exceed 32 characters in size,
- must start with a letter, and
- must only contain letters, digits, the underscore ("_") and brackets ("[" and "]").

3.2.1.1.2 GetDescription

```
String8 GetDescription();
```

This method returns the description of the object ("property getter"). Applications may display the description as additional information on the object.

Parameters:

None.

Returns:

Description of object.

Exceptions:

None.

Remarks:

Descriptions are optional and may be empty.

3.2.2 Components

Most all 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

```
module Smp
{
   interface IComponent : IObject
   {
      IComposite GetParent();
   };
};
```

All SMP2 components implement this base interface.

Inheritance Diagram:

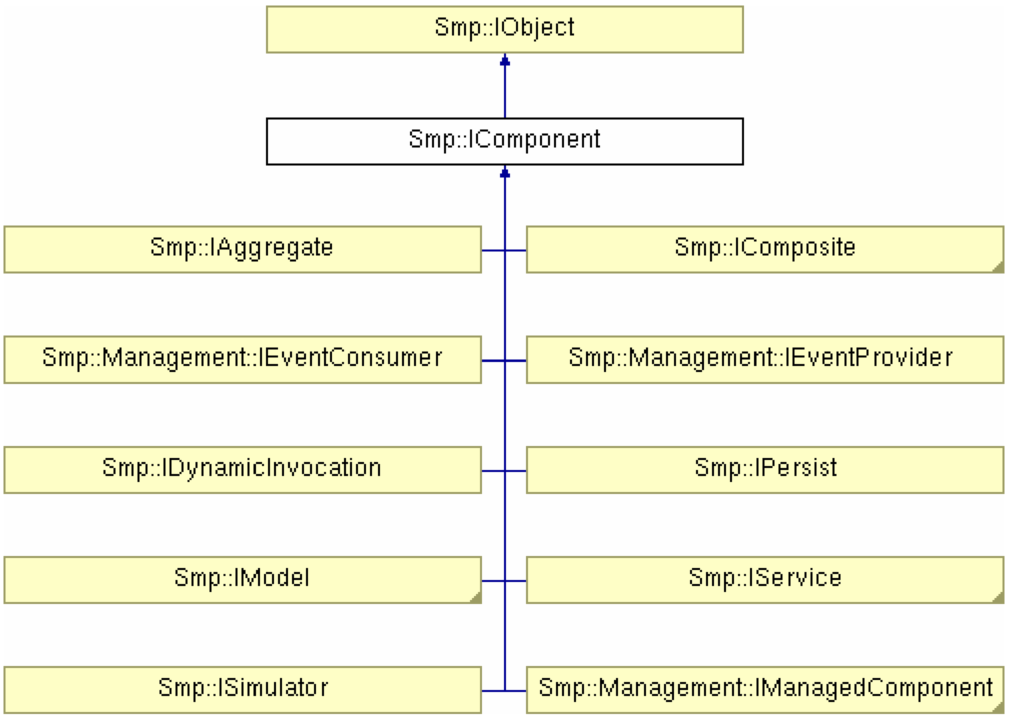

Figure 3-2: IComponent Interface

Remarks:

Services and models are the most typical components in SMP2.

3.2.2.1.1 GetParent

```
IComposite GetParent();
```

This method returns the parent component of the component (“property getter”). Components link to their parent to allow traversing the tree of components upwards to the root component, which is typically the simulation environment.

Parameters:

None.

Returns:

Parent component of component, or null if component has no parent.

Exceptions:

None.

Remarks:

Components that are part of a composition point to their parent via this property. Typically, only the simulator itself is a root component, so all other components should have a parent component.

3.2.2.2 ComponentCollection

```
module Smp
{
   typedef sequence<IComponent> ComponentCollection;
};
```

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

Remarks:

This type is platform specific. For details see the SMP2 Platform Mappings.

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.

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

Table 3-1: Model States

State name	Description	Enter	Leave
`Created`	The `Created` state is the initial state of a model. Model creation is done by an external mechanism, e.g. by Factories.	This state is entered automatically after the model has been created.	To leave it, call the `Publish()` state transition method.
`Publishing`	In `Publishing` state, the model is allowed to publish features. This includes publication of fields, operations and properties. In addition, the model is allowed to create other models.	This state is entered using the `Publish()` state transition.	To leave it, call the `Configure()` state transition method.
`Configured`	In `Configured` state, the model has been fully configured. This configuration may be done by external components, e.g. based on information stored in an SMDL Assembly, or internally by the model itself, e.g. by reading data from an external source.	This state is entered using the `Configure()` state transition.	To leave it, call the `Connect()` state transition method.
`Connected`	In `Connected` state, the model is connected to the simulator. In this state, neither publication nor creation of other models is allowed anymore.	This state is entered using the `Connect()` state transition	This is the final state of a model, and only left on termination.

3.2.2.4 IModel

```
module Smp
{
   interface IModel : IComponent
   {
      /// Invalid model state.
      /// This exception is raised by a model when one of the
      /// state transition commands is called in an invalid state.
      exception InvalidModelState
      {
         /// State that is not valid.
         ModelStateKind invalidState;
         /// State that was expected.
         ModelStateKind expectedState;
      };

      ModelStateKind GetState();
      void Publish(in IPublication receiver)      raises (InvalidModelState);
      void Configure(in Services::ILogger logger) raises (InvalidModelState);
      void Connect(in ISimulator simulator)       raises (InvalidModelState);
   };
};
```

All SMP2 models implement this interface. As models interface to the simulation environment, they have a dependency to it via the two interfaces **`IPublication`** (see 3.6.2.1) and **`ISimulator`** (see 3.6.1.2).

Inheritance Diagram:

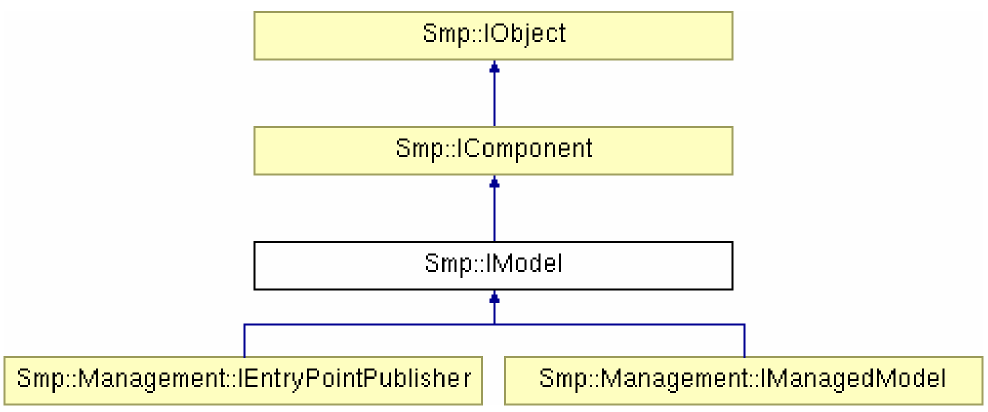
Smp::IObject
Smp::IComponent
Smp::IModel
Smp::Management::IEntryPointPublisher
Smp::Management::IManagedModel

Figure 3-3: IModel Interface

Remarks:

This is the only mandatory interface models have to implement. All other functionality derived from it is optional.

3.2.2.4.1 GetState

```
ModelStateKind GetState();
```

Return the state the model is currently in.

Parameters:

None.

Returns:

Current model state.

Exceptions:

None.

Remarks:

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

3.2.2.4.2 Publish

```
void Publish(in IPublication receiver) raises (InvalidModelState);
```

Request the model to publish its fields and operations against the provided **`IPublication`** interface.

This method can only be called once for each model, and only when the model is in the `Created` state. When this operation is called, the model immediately enters the `Publishing` state, before it publishes any of its features.

Parameters:

receiver	Publication receiver.

Returns:

Void.

Exceptions:

This method raises an `InvalidModelState` exception if the model is not in `Created` state.

Remarks:

The simulation environment typically calls this method in the `Building` state.

3.2.2.4.3 Configure

```
void Configure(in Services::ILogger logger) raises (InvalidModelState);
```

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

This method can only be called once for each model, and only when the model is in `Publishing` state. The model can still publish further features in this call, and can even create other models, but at the end of this call, it needs to enter the `Configured` state.

Parameters:

logger	Logger service for logging of error messages during configuration.

Returns:

Void.

Exceptions:

This method raises an `InvalidModelState` exception if the model is not in `Publishing` state.

Remarks:

The simulation environment typically calls this method in the `Building` state.

3.2.2.4.4 Connect

```
void Connect(in ISimulator simulator) raises (InvalidModelState);
```

Allow the model to connect to the simulator.

This method can only be called once for each model, and only when the model is in the `Configured` state. When this operation is called, the model immediately enters the `Connected` state, before it uses any of the simulator methods and services.

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

Parameters:

simulator	Simulation Environment that hosts the model.

Returns:

Void.

Exceptions:

This method raises an `InvalidModelState` exception if the model is not in `Configured` state.

Remarks:

The simulation environment typically calls this method in the `Connecting` state.

3.2.2.5 ModelCollection

```
module Smp
{
    typedef sequence<IModel> ModelCollection;
};
```

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

Remarks:

This type is platform specific. For details see the SMP2 Platform Mappings.

3.2.2.6 IService

```
module Smp
{
    interface IService : IComponent
    {
    };
};
```

All SMP2 services implement this interface.

Inheritance Diagram:

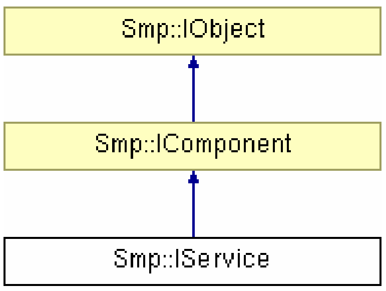

Figure 3-4: IService Interface

Remarks:

Currently, this interface does not add any functionality.

3.2.2.7 ServiceCollection

```
module Smp
{
   typedef sequence<IService> ServiceCollection;
};
```

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

Remarks:

This type is platform specific. For details see the SMP2 Platform Mappings.

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.

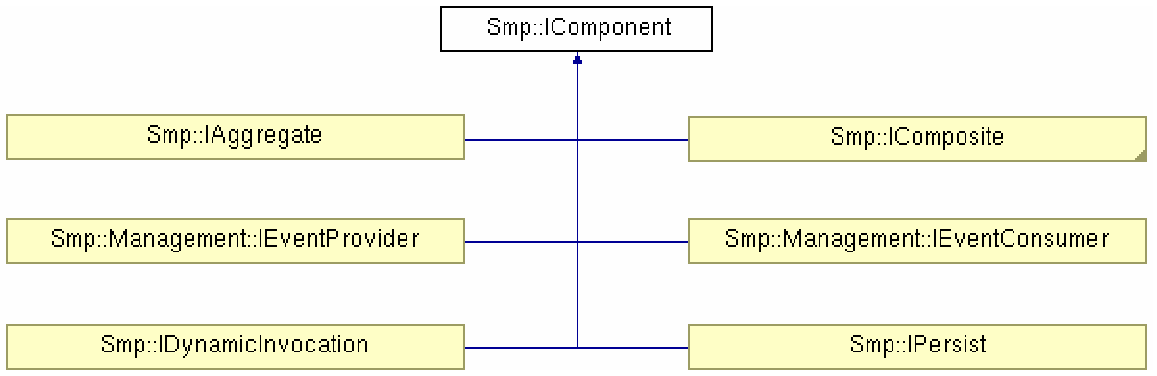

Figure 3-5: SMP2 Component Mechanisms

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

```
module Smp
{
   interface IAggregate : IComponent
   {
      ReferenceCollection GetReferences();
      IReference GetReference(in String8 name);
   };
};
```

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

Inheritance Diagram:

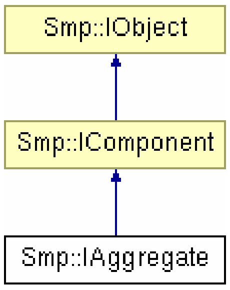

Figure 3-6: IAggregate Interface

Remarks:

This interface represents the Aggregation mechanism in the SMP2 Metamodel (via References). In UML 2.0, this maps to an aggregate required interface.

3.3.1.1.1 GetReference

```
IReference GetReference(in String8 name);
```

Query for a reference of this component by its name.

Parameters:

name	Reference name.

Returns:

Reference queried for by name, or null if no reference with this name exists.

Exceptions:

None.

Remarks:

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

3.3.1.1.2 GetReferences

```
ReferenceCollection GetReferences();
```

Query for the collection of all references of the component.

Parameters:

None.

Returns:

Collection of references.

Exceptions:

None.

Remarks:

The collection may be empty if no references exist.

3.3.1.2 IReference

```
module Smp
{
   interface IReference : IObject
   {
      ComponentCollection GetComponents();
      IComponent GetComponent(in String8 name);
   };
};
```

A reference allows querying for the referenced components.

Inheritance Diagram:

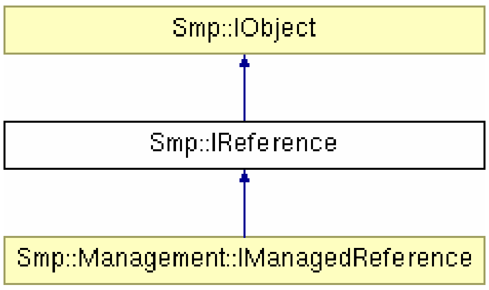

Figure 3-7: IReference Interface

Remarks:

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

3.3.1.2.1 GetComponent

```
IComponent GetComponent(in String8 name);
```

Query for a referenced component by its name.

Parameters:

name	Component name.

Returns:

Referenced component with the given name, or null if no referenced component with the given name could be found.

Exceptions:

None.

Remarks:

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

3.3.1.2.2 GetComponents

```
ComponentCollection GetComponents();
```

Query for the collection of all referenced components.

Parameters:

None.

Returns:

Collection of referenced components.

Exceptions:

None.

Remarks:

The collection may be empty if no components exist.

3.3.1.3 ReferenceCollection

```
module Smp
{
   typedef sequence<IReference> ReferenceCollection;
};
```

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

Remarks:

This type is platform specific. For details see the SMP2 Platform Mappings.

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

```
module Smp
{
   interface IComposite : IComponent
   {
      ContainerCollection GetContainers();
      IContainer GetContainer(in String8 name);
   };
};
```

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

Inheritance Diagram:

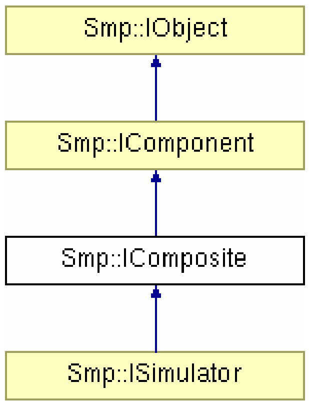

Figure 3-8: IComposite Interface

Remarks:

This interface represents the Composition mechanism in the SMP2 Metamodel (via Containers). In UML 2.0, this maps to a composite required interface.

3.3.2.1.1 GetContainer

```
IContainer GetContainer(in String8 name);
```

Query for a container of this component by its name.

Parameters:

name	Container name.

Returns:

Container queried for by name, or null if no container with this name exists.

Exceptions:

None.

Remarks:

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

3.3.2.1.2 GetContainers

```
ContainerCollection GetContainers();
```

Query for the collection of all containers of the component.

Parameters:

None.

Returns:

Collection of containers.

Exceptions:

None.

Remarks:

The collection may be empty if no containers exist.

3.3.2.2 IContainer

```
module Smp
{
   interface IContainer : IObject
   {
      ComponentCollection GetComponents();
      IComponent GetComponent(in String8 name);
   };
};
```

A container allows querying for its children.

Inheritance Diagram:

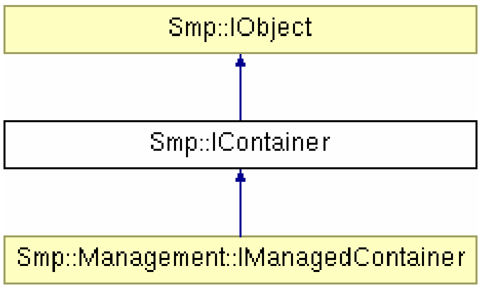

Figure 3-9: IContainer Interface

Remarks:

Containers are used together with the **`IComposite`** interface for composition.

3.3.2.2.1 GetComponent

```
IComponent GetComponent(in String8 name);
```

Query for a child component by its name.

Parameters:

name	Child name.

Returns:

Child component, or null if no child component with the given name exists.

Exceptions:

None.

Remarks:

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

3.3.2.2.2 GetComponents

```
ComponentCollection GetComponents();
```

Query for the collection of all children.

Parameters:

None.

Returns:

Collection of contained components.

Exceptions:

None.

Remarks:

The collection may be empty if no components exist.

3.3.2.3 ContainerCollection

```
module Smp
{
   typedef sequence<IContainer> ContainerCollection;
};
```

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

Remarks:

This type is platform specific. For details see the SMP2 Platform Mappings.

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

```
module Smp
{
   interface IEventSink : IObject
   {
      void Notify(in IObject sender, in AnySimple arg);
   };
};
```

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

Inheritance Diagram:

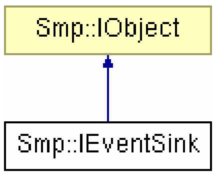

Figure 3-10: IEventSink Interface

Remarks:

None.

3.3.3.1.1 Notify

```
void Notify(in IObject sender, in AnySimple arg);
```

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

Parameters:

sender	Object sending the event.
arg	Event argument.

Returns:

Void.

Exceptions:

None.

Remarks:

None.

3.3.3.2 EventSinkCollection

```
module Smp
{
   typedef sequence<IEventSink> EventSinkCollection;
};
```

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

Remarks:

This type is platform specific. For details see the SMP2 Platform Mappings.

3.3.3.3 IEventSource

```
module Smp
{
   interface IEventSource : IObject
   {
      exception AlreadySubscribed
      {
         IEventSource eventSource;
         IEventSink   eventSink;
      };

      exception NotSubscribed
      {
         IEventSource eventSource;
         IEventSink   eventSink;
      };

      exception InvalidEventSink
      {
         IEventSource eventSource;
         IEventSink   eventSink;
      };

      void Subscribe  (in IEventSink eventSink)
         raises (AlreadySubscribed, InvalidEventSink);

      void Unsubscribe(in IEventSink eventSink)
         raises (NotSubscribed);
   };
};
```

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

Inheritance Diagram:

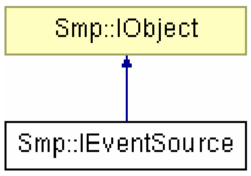

Figure 3-11: IEventSource Interface

Remarks:

None.

3.3.3.3.1 Subscribe

```
      void Subscribe  (in IEventSink eventSink)
         raises (AlreadySubscribed, InvalidEventSink);
```

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

Parameters:

eventSink	Event sink to subscribe to event source.

Returns:

Void.

Exceptions:

If the given event sink is already subscribed to the event source, the exception `AlreadySubscribed` is raised. In addition, an exception of type `InvalidEventSink` may be raised when the event argument of the event sink is not of the type the event source expects. This exception depends on additional metadata that is not defined in this component model. See the SMP2 Metamodel [AD-2] for details about event types.

Remarks:

An event sink can only be subscribed once to each event source.

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

Event sinks will be called synchronously in the order they have been subscribed to the event source.

3.3.3.3.2 Unsubscribe

```
void Unsubscribe(in IEventSink eventSink) raises (NotSubscribed);
```

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

Parameters:

eventSink	Event sink to unsubscribe from event source.

Returns:

Void.

Exceptions:

If the given event sink is not subscribed to the event source, the exception `NotSubscribed` is raised.

Remarks:

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

3.3.3.4 EventSourceCollection

```
module Smp
{
   typedef sequence<IEventSource> EventSourceCollection;
};
```

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

Remarks:

This type is platform specific. For details see the SMP2 Platform Mappings.

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`. The sequence diagram in Figure 3-12 shows all steps involved when using the `CreateRequest()` and `DeleteRequest()` methods. Especially when running distributed models, this implementation is slow. In these cases, the `Request` object should be created by the `Client`, and only passed to the `Model` on `Invoke()`.

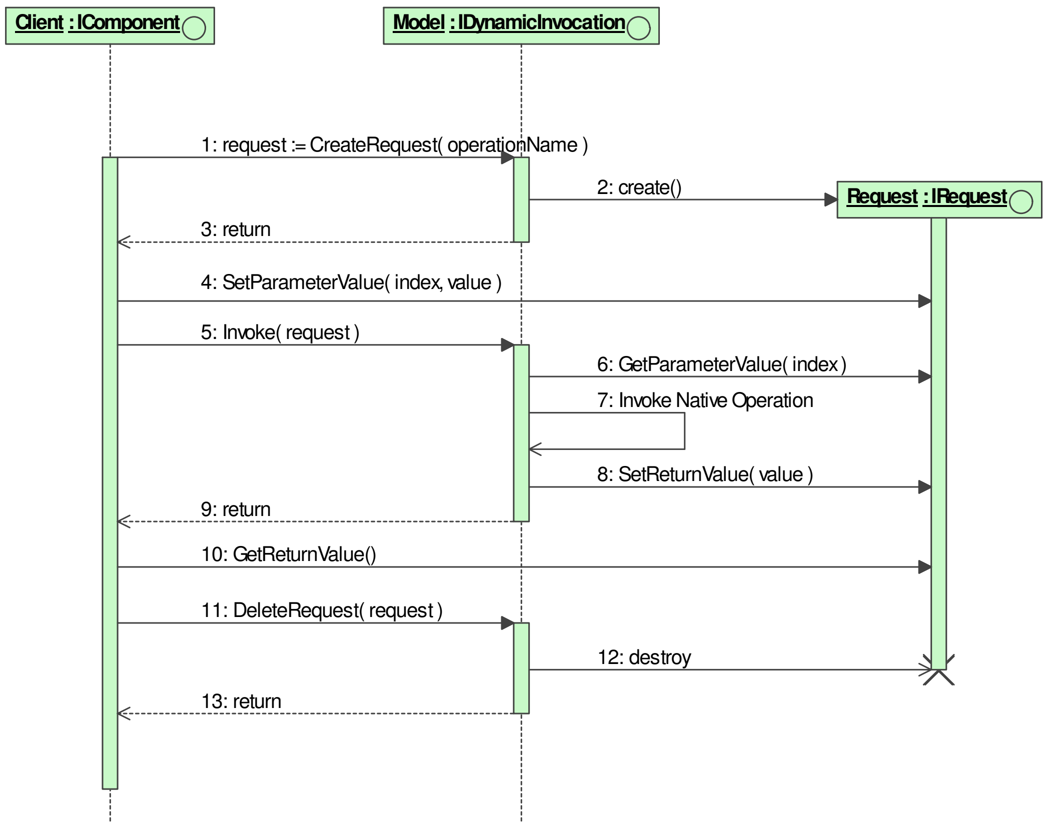
Client : IComponent
Model : IDynamicInvocation
Request : IRequest
1: request := CreateRequest(operationName)
2: create()
3: return
4: SetParameterValue(index, value)
5: Invoke(request)
6: GetParameterValue(index)
7: Invoke Native Operation
8: SetReturnValue(value)
9: return
10: GetReturnValue()
11: DeleteRequest(request)
12: destroy
13: return

Figure 3-12: Sequence of calls for dynamic invocation

The sequence diagram in Figure 3-12, using a Client component and a Model implementing `IDynamicInvocation`, contains the following steps:

1. The client calls the `CreateRequest()` operation of the component to create a request object for the operation, passing it the name of the operation.
2. The component creates a request object for the operation, using the default values of all parameters.
3. The component returns the Request object via its `IRequest` interface to the client.

4. The client calls the `SetParameterValue()` operation of the Request object to set parameters to non-default values.
5. The client calls the `Invoke()` operation of the component to invoke the corresponding operation.
6. The component calls the `GetParameterValue()` operation of the Request object to get parameters.
7. The component calls its internal operation that corresponds to the invoked operation.
8. The component calls the `SetReturnValue()` operation of the Request object to set the return value.
9. The component returns control to the client.
10. The client calls the `GetReturnValue()` operation of the Request object to get the return value.
11. The client calls the `DeleteRequest()` operation of the component to delete the Request object.
12. The component destroys the request object.
13. The component returns control to the client.

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

3.3.4.1 IDynamicInvocation

```
module Smp
{
   interface IDynamicInvocation : IComponent
   {
      exception InvalidOperationName
      {
         String8 operationName;
      };

      exception InvalidParameterCount
      {
         String8 operationName;
         Int32   operationParameters;
         Int32   requestParameters;
      };

      exception InvalidParameterType
      {
         String8 operationName;
         String8 parameterName;
         SimpleTypeKind invalidType;
         SimpleTypeKind expectedType;
      };

      IRequest CreateRequest(in String8 operationName);
      void Invoke(in IRequest request) raises
      (
         InvalidOperationName,
         InvalidParameterCount,
         InvalidParameterType
      );
      void DeleteRequest(in IRequest request);
   };
};
```

A component (typically a model) may implement this interface in order to allow dynamic invocation of its operations. Dynamic invocation is typically used for scripting.

Inheritance Diagram:

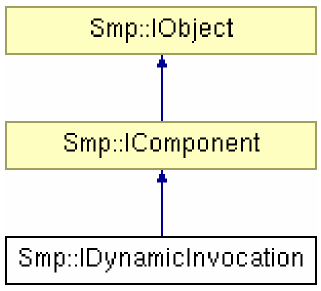

Figure 3-13: IDynamicInvocation Interface

3.3.4.1.1 CreateRequest

```
IRequest CreateRequest(in String8 operationName);
```

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

Parameters:

operationName	Name of operation.

Returns:

Request object for operation, or null if either no operation with the given name could be found, or the operation with the given name does not support dynamic invocation.

Exceptions:

None.

Remarks:

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

3.3.4.1.2 DeleteRequest

```
void DeleteRequest(in IRequest request);
```

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

Parameters:

request	Request object to destroy.

Returns:

Void.

Exceptions:

None.

Remarks:

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

3.3.4.1.3 Invoke

```
void Invoke(in IRequest request)
     raises (
               InvalidOperationName,
               InvalidParameterCount,
               InvalidParameterType
            );
```

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

Parameters:

request	Request object to invoke.

Returns:

Void.

Exceptions:

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

Remarks:

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

3.3.4.2 IRequest

```
module Smp
{
   interface IRequest
   {
      exception InvalidParameterIndex
      {
         String8 operationName;
         Int32 parameterIndex;
      };

      exception InvalidParameterValue
      {
         String8 parameterName;
         AnySimple value;
      };

      exception IllegalReturnValue
      {
         String8 operationName;
         AnySimple value;
      };

      exception VoidOperation
      {
         String8 operationName;
      };

      String8   GetOperationName ();
      Int32     GetParameterCount();
      Int32     GetParameterIndex(in String8 parameterName);

      void      SetParameterValue(in Int32 index, in AnySimple value)
                raises (InvalidParameterIndex,
                        InvalidParameterValue,
                        InvalidAnyType);

      AnySimple GetParameterValue(in Int32 index)
                raises (InvalidParameterIndex);

      void      SetReturnValue(in AnySimple value)
                raises (InvalidReturnValue,
                        VoidOperation,
                        InvalidAnyType);

      AnySimple GetReturnValue()
                raises (VoidOperation);
   };
};
```

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

Remarks:

None.

3.3.4.2.1 GetOperationName

```
String8 GetOperationName();
```

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

Parameters:

Void.

Returns:

Name of the operation.

Exceptions:

None.

Remarks:

A request is typically created using the `CreateRequest()` method to dynamically call a specific method of a component implementing the `IDynamicInvocation` interface. This method returns the name passed to it, to allow finding out which method is actually called on `Invoke()`.

3.3.4.2.2 GetParameterCount

```
Int32 GetParameterCount();
```

Returns the number of parameters stored in the request.

Parameters:

Void.

Returns:

Number of parameters in request object.

Exceptions:

None.

Remarks:

Parameters are typically accessed by their 0-based index. This index
- must not be negative,
- must be smaller than the parameter count.

Use the **`GetParameterIndex()`** method to access parameters by name.

3.3.4.2.3 GetParameterIndex

```
Int32 GetParameterIndex(in String8 parameterName);
```

Query for a parameter index by parameter name.

Parameters:

parameterName	Name of parameter.

Returns:

Index of parameter with the given name, or -1 if no parameter with the given name could be found.

Exceptions:

None.

Remarks:

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

3.3.4.2.4 SetParameterValue

```
void    SetParameterValue(in Int32 index, in AnySimple value)
        raises (InvalidParameterIndex, InvalidParameterValue, InvalidAnyType);
```

Assign a value to a parameter at a given position.

Parameters:

index	Index of parameter.
value	Value of parameter.

Returns:

Void.

Exceptions:

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

Remarks:

None.

3.3.4.2.5 GetParameterValue

```
AnySimple GetParameterValue(in Int32 index) raises (InvalidParameterIndex);
```

Query a value of a parameter at a given position.

Parameters:

index	Index of parameter.

Returns:

Value of parameter.

Exceptions:

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

Remarks:

None.

3.3.4.2.6 SetReturnValue

```
void SetReturnValue(in AnySimple value)
     raises (InvalidReturnValue, VoidOperation, InvalidAnyType);
```

Assign the return value of the operation.

Parameters:

value	Return value.

Returns:

Void.

Exceptions:

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

Remarks:

None.

3.3.4.2.7 GetReturnValue

```
AnySimple GetReturnValue() raises (VoidOperation);
```

Query the return value of the operation.

Parameters:

Void.

Returns:

Return value of the operation.

Exceptions:

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

Remarks:

None.

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.

   Note: 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.

   Note: 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.

3.3.5.1 IPersist

```
module Smp
{
   interface IPersist : IComponent
   {
      exception CannotRestore
      {
         String8 message;
      };

      exception CannotStore
      {
         String8 message;
      };

      void Restore(in IStorageReader reader) raises (CannotRestore);
      void   Store(in IStorageWriter writer) raises (CannotStore);
   };
};
```

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

Inheritance Diagram:

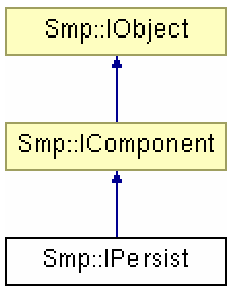

Figure 3-14: IPersist Interface

Remarks:

This is an optional interface. It needs to be implemented for components with self-persistence only.

3.3.5.1.1 Restore

```
void Restore(in IStorageReader reader) raises (CannotRestore);
```

Restore component state from storage.

Parameters:

reader	Interface that allows to read from storage.

Returns:

Void.

Exceptions:

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

Remarks:

None.

3.3.5.1.2 Store

```
void Store(in IStorageWriter writer) raises (CannotStore);
```

Store model state to storage.

Parameters:

writer	Interface that allows to write to storage.

Returns:

Void.

Exceptions:

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

Remarks:

None.

3.3.5.2 IStorageReader

```
module Smp
{
   interface IStorageReader
   {
   };
};
```

Provide functionality to read 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 model implementing `IPersist`.

Remarks:

This interface is platform specific. For details see the SMP2 Platform Mappings.

3.3.5.3 IStorageWriter

```
module Smp
{
   interface IStorageWriter
   {
   };
};
```

Provide functionality to write 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`.

Remarks:

This interface is platform specific. For details see the SMP2 Platform Mappings.

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

3.4.1.1 IEntryPoint

```
module Smp
{
    interface IEntryPoint : IObject
    {
        IComponent GetOwner();

        void Execute();
    };
};
```

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

Inheritance Diagram:

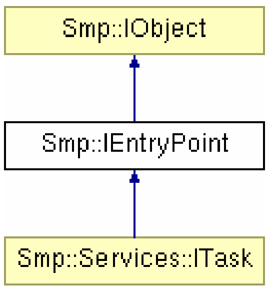

Figure 3-15: IEntryPoint Interface

Remarks:

None.

3.4.1.1.1 GetOwner

```
IComponent GetOwner();
```

This method returns the Component that owns the entry point.

Parameters:

None.

Returns:

Owner of entry point.

Exceptions:

None.

Remarks:

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

3.4.1.1.2 Execute

```
void Execute();
```

This method is called when an associated event is emitted.

Parameters:

None.

Returns:

Void.

Exceptions:

None.

Remarks:

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

3.4.1.2 EntryPointCollection

```
module Smp
{
   typedef sequence<IEntryPoint> EntryPointCollection;
};
```

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

Remarks:

This type is platform specific. For details see the SMP2 Platform Mappings.

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.

Remarks:

> Typically, a Loader or Model Manager Component calls these interfaces to push configuration information into the models, which has been read from an SMDL Assembly file.

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

```
module Smp
{
   module Management
   {
      interface IManagedObject : IObject
      {
         void SetName(in String8 name) raises (InvalidObjectName);
         void SetDescription(in String8 description);
      };
   };
};
```

A managed object additionally allows assigning name and description.

Inheritance Diagram:

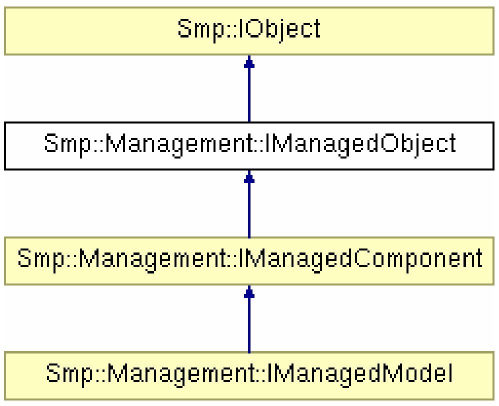
Smp::IObject
Smp::Management::IManagedObject
Smp::Management::IManagedComponent
Smp::Management::IManagedModel

Figure 3-16: IManagedObject Interface

3.5.1.1.1 SetName

```
void SetName(in String8 name) raises (InvalidObjectName);
```

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

Parameters:

name	Name of object.

Returns:

Void.

Exceptions:

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

Remarks:

Names
- must be unique within their context,
- must not be empty,
- must not exceed 32 characters in size,
- must start with a letter, and
- must only contain letters, digits, the underscore ("_") and brackets ("[" and "]").

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

3.5.1.1.2 SetDescription

```
void SetDescription(in String8 description);
```

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

Parameters:

description	Description of object.

Returns:

Void.

Exceptions:

None.

Remarks:

None.

3.5.1.2 IManagedComponent

```
module Smp
{
   module Management
   {
      interface IManagedComponent : IManagedObject, IComponent
      {
         void SetParent(in IComposite parent);
      };
   };
};
```

A managed component additionally allows assigning the parent.

Inheritance Diagram:

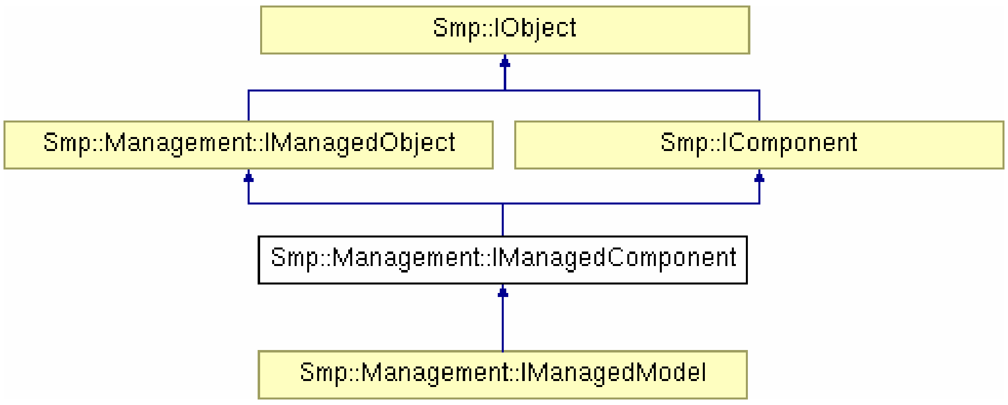
Smp::IObject
Smp::Management::IManagedObject
Smp::IComponent
Smp::Management::IManagedComponent
Smp::Management::IManagedModel

Figure 3-17: IManagedComponent Interface

3.5.1.2.1 SetParent

```
void SetParent(in IComposite parent);
```

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

Parameters:

parent	Parent of component.

Returns:

Void.

Exceptions:

None.

Remarks:

None.

3.5.2 Managed Component Mechanisms

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

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

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

3.5.2.1 IManagedReference

```
module Smp
{
   module Management
   {
      interface IManagedReference : IReference
      {
         exception ReferenceFull
         {
            String8 referenceName;
            Int64   referenceSize;
         };

         exception NotReferenced
         {
            String8 referenceName;
            IComponent _component;
         };

         void AddComponent(in IComponent _component)
              raises (ReferenceFull, InvalidObjectType);
         Int64 Count();
         Int64 Lower();
         Int64 Upper();
         void RemoveComponent(in IComponent _component)
              raises (NotReferenced);
      };
   };
};
```

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

Inheritance Diagram:

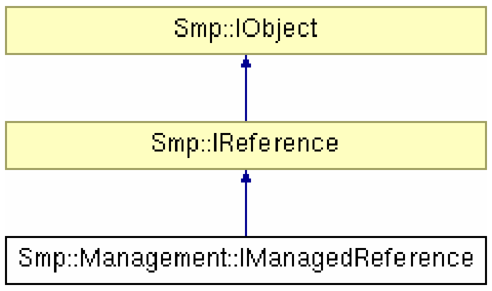

Figure 3-18: IManagedReference Interface

3.5.2.1.1 AddComponent

```
void AddComponent(in IComponent _component)
     raises (ReferenceFull, InvalidObjectType);
```

Add a referenced component.

Parameters:

component	New referenced component.

Returns:

Void.

Exceptions:

This method raises an exception of type `ReferenceFull` if called for a full reference, i.e. when the `Count()` has reached the `Upper()` limit. Based on additional metadata, this method may raise an exception of type `InvalidObjectType` when it expects the given component to implement another interface as well. See the SMP2 Metamodel AD-2] for details on references.

Remarks:

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

3.5.2.1.2 Count

```
Int64 Count();
```

Query for the number of referenced components.

Parameters:

None.

Returns:

Current number of referenced components.

Exceptions:

None.

Remarks:

None.

3.5.2.1.3 Lower

```
Int64 Lower();
```

Query the minimum number of components in the collection of references.

Parameters:

None.

Returns:

Minimum number of referenced components.

Exceptions:

None.

Remarks:

This information can be used to validate a model hierarchy. If a managed reference specified a `Lower()` value above its current `Count()`, then it is not properly configured. An external component may use this information to validate the configuration before executing it.

3.5.2.1.4 Upper

```
Int64 Upper();
```

Query the maximum number of components in the collection of references.

Parameters:

None.

Returns:

Maximum number of referenced components (-1 = unlimited).

Exceptions:

None.

Remarks:

A return value of -1 indicates that the reference has no upper limit. This is consistent with the use of upper bounds in UML, where a value of -1 represents no limit (typically shown as “*”).

This information can be used to validate that another component can be added to the managed reference before calling the `AddComponent()` method. If `Count()` has reached the `Upper()` limit, further calls to `AddComponent()` would result in an exception.

3.5.2.1.5 RemoveComponent

```
void RemoveComponent(in IComponent _component) raises (NotReferenced);
```

Remove a referenced component.

Parameters:

component	Referenced component.

Returns:

Void.

Exceptions:

This method raises an exception of type NotReferenced if called with a component that is not referenced.

Remarks:

This method throws an exception if the component is not referenced.

3.5.2.2 IManagedContainer

```
module Smp
{
   module Management
   {
      interface IManagedContainer : IContainer
      {
         exception ContainerFull
         {
            String8 containerName;
            Int64   containerSize;
         };

         void AddComponent(in IComponent _component)
              raises (ContainerFull, DuplicateName, InvalidObjectType);
         Int64 Count();
         Int64 Lower();
         Int64 Upper();
      };
   };
};
```

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

Inheritance Diagram:

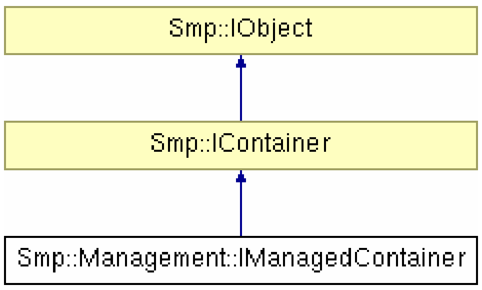

Figure 3-19: IManagedContainer Interface

3.5.2.2.1 AddComponent

```
void AddComponent(in IComponent _component)
     raises (ContainerFull, DuplicateName, InvalidObjectType);
```

Add a contained component to the container.

Parameters:

component	New contained component.

Returns:

Void.

Exceptions:

This method raises an exception of type `ContainerFull` if called for a full container, i.e. when the `Count()` has reached the `Upper()` limit. It raises an exception of type `DuplicateName` when trying to add a component with a name that is already contained in the container, as this would lead to duplicate names in the container. Based on additional metadata, this method may raise an exception of type `InvalidObjectType` when it expects the given component to implement another interface as well. See the SMP2 Metamodel AD-2] for details on containers.

Remarks:

This method throws an exception if the container is full, or if the name of the component would lead to duplicate names within the container.

3.5.2.2.2 Count

```
Int64 Count();
```

Query for the number of components in the container.

Parameters:

None.

Returns:

Current number of components.

Exceptions:

None.

Remarks:

None.

3.5.2.2.3 Lower

```
Int64 Lower();
```

Query the minimum number of components in the container.

Parameters:

None.

Returns:

Minimum number of components.

Exceptions:

None.

Remarks:

This information can be used to validate a model hierarchy. If a managed container specified a `Lower()` value above its current `Count()`, then it is not properly configured. An external component may use this information to validate the configuration before executing it.

.

3.5.2.2.4 Upper

```
Int64 Upper();
```

Query the maximum number of components in the container.

Parameters:

None.

Returns:

Maximum number of contained components (-1 = unlimited).

Exceptions:

None.

Remarks:

A return value of -1 indicates that the container has no upper limit. This is consistent with the use of upper bounds in UML, where a value of -1 represents no limit (typically shown as “*”).

This information can be used to validate that another component can be added to the managed container before calling the `AddComponent()` method. If `Count()` has reached the `Upper()` limit, further calls to `AddComponent()` would result in an exception.

3.5.2.3 IEventConsumer

```
module Smp
{
   module Management
   {
      interface IEventConsumer : IComponent
      {
         EventSinkCollection GetEventSinks();
         IEventSink GetEventSink(in String8 name);
      };
   };
};
```

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

Inheritance Diagram:

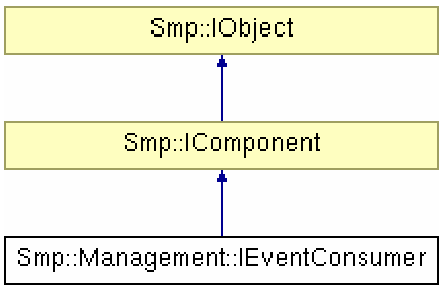

Figure 3-20: IEventConsumer Interface

Remarks:

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

3.5.2.3.1 GetEventSink

```
IEventSink GetEventSink(in String8 name);
```

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

Parameters:

name	Event sink name.

Returns:

Event sink with the given name, or null if no event sink with the given name could be found.

Exceptions:

None.

Remarks:

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

3.5.2.3.2 GetEventSinks

```
EventSinkCollection GetEventSinks();
```

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

Parameters:

None.

Returns:

Collection of event sinks.

Exceptions:

None.

Remarks:

The collection may be empty if no event sinks exist.

3.5.2.4 IEventProvider

```
module Smp
{
   module Management
   {
      interface IEventProvider : IComponent
      {
         EventSourceCollection GetEventSources();
         IEventSource GetEventSource(in String8 name);
      };
   };
};
```

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

Inheritance Diagram:

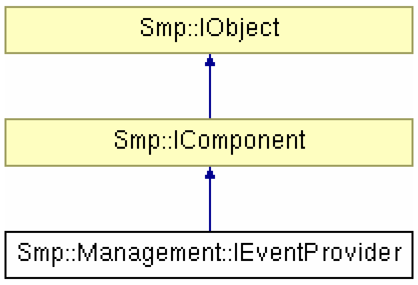

Figure 3-21: IEventProvider Interface

Remarks:

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

3.5.2.4.1 GetEventSource

```
IEventSource GetEventSource(in String8 name);
```

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

Parameters:

name	Event source name.

Returns:

Event source with the given name, or null if no event source with the given name could be found.

Exceptions:

None.

Remarks:

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

3.5.2.4.2 GetEventSources

```
EventSourceCollection GetEventSources();
```

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

Parameters:

None.

Returns:

Collection of event sources.

Exceptions:

None.

Remarks:

The collection may be empty if no event sources exist.

3.5.3 Managed Model Mechanisms

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

- Entry points can be registered with services by models, but an external component can not query for entry points by name.

- Fields are published against the simulation environment, but no read and write access to named fields is provided.

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

```
module Smp
{
   module Management
   {
      interface IManagedModel : IModel, IManagedComponent
      {
         exception InvalidFieldName
         {
            String8 fieldName;
         };

         exception IllegalFieldValue
         {
            String8 fieldName;
            AnySimple invalidValue;
         };

         exception InvalidArraySize
         {
            String8 fieldName;
            Int64 givenSize;
            Int64 arraySize;
         };

         exception InvalidArrayValue
         {
            String8 fieldName;
            AnySimpleArray invalidValues;
         };

         AnySimple GetFieldValue(in String8 fullName)
              raises (InvalidFieldName);

         void      SetFieldValue(in String8 fullName, in AnySimple value)
              raises (InvalidFieldName, IllegalFieldValue);

         void GetArrayValue(in String8 fullName,
                            inout AnySimpleArray values,
                            in Int64 length)
              raises (InvalidFieldName, InvalidArraySize);

         void SetArrayValue(in String8 fullName,
                            in AnySimpleArray values,
                            in Int64 length)
              raises (InvalidFieldName, InvalidArraySize, InvalidArrayValue);
      };
   };
};
```

A managed model additionally allows querying or modifying field values.

Inheritance Diagram:

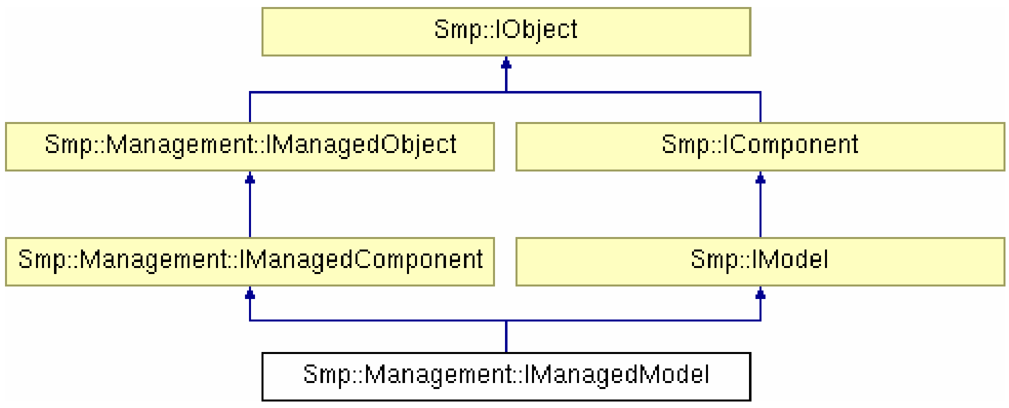
Smp::IObject
Smp::Management::IManagedObject
Smp::IComponent
Smp::Management::IManagedComponent
Smp::IModel
Smp::Management::IManagedModel

Figure 3-22: IManagedModel Interface

3.5.3.1.1 GetFieldValue

```
AnySimple GetFieldValue(in String8 fullName) raises (InvalidFieldName);
```

Get the value of a field that is typed by a simple type (see 2.2.1).

Parameters:

fullName	Fully qualified field name.

Returns:

Field value.

Exceptions:

This method raises an exception of type `InvalidFieldName` if called with an invalid field name.

Remarks:

This method can only be used to get values of simple fields. For getting values of structured or array fields, this method may be called multiply, for example by specifying a field name "`MyField.Position[2]`" in order to access an array item of a structure.

3.5.3.1.2 SetFieldValue

```
void SetFieldValue(in String8 fullName, in AnySimple value)
     raises (InvalidFieldName, InvalidFieldValue);
```

Set the value of a field that is typed by a simple type (see 2.2.1).

Parameters:

fullName	Fully qualified field name.
value	Field value.

Returns:

Void.

Exceptions:

This method raises an exception of type `InvalidFieldName` if called with an invalid field name, and an exception of type `InvalidFieldValue` if called with an invalid value.

Remarks:

This method can only be used to set values of simple fields. For setting values of structured or array fields, this method may be called multiply, for example by specifying a field name "`MyField.Position[2]`" in order to access an array item of a structure.

3.5.3.1.3 GetArrayValue

```
void GetArrayValue(in String8 fullName,
                   inout AnySimpleArray values,
                   in Int64 length)
     raises (InvalidFieldName, InvalidArraySize);
```

Get the value of a field that is an array of a simple type (see 2.2.1).

Parameters:

fullName	Fully qualified field name.
values	Pre-allocated array of values to store result to.
length	Size of given values array.

Returns:

Void.

Exceptions:

This method raises an exception of type `InvalidFieldName` if called with an invalid field name. It raises an exception of type `InvalidArraySize` if called with an array of wrong size.

Remarks:

This method can only be used to get values of array fields of simple type. The array with the values has to be pre-allocated by the caller, and has to be released by the caller as well. Therefore, an in/out parameter has been used, not the return value of the method.

3.5.3.1.4 SetArrayValue

```
void SetArrayValue(in String8 fullName,
                   in AnySimpleArray values,
                   in Int64 length)
     raises (InvalidFieldName, InvalidArraySize, InvalidArrayValue);
```

Set the value of a field that is typed by a simple type (see 2.2.1).

Parameters:

fullName	Fully qualified field name.
values	Array of values to store in array field.
length	Size of given values array.

Returns:

Void.

Exceptions:

This method raises an exception of type `InvalidFieldName` if called with an invalid field name. It raises an exception of type `InvalidArraySize` if called with an array of wrong size, and an exception of type `InvalidArrayValue` if called with an invalid values array.

Remarks:

This method can only be used to set values of array fields of simple type.

3.5.3.1.5 Field Names

In SMP2, models can have fields of various types, including not only the simple types introduced in section 2.2 (Data Types), but also complex types (strings, arrays, structures, classes). The available types are documented in the Simulation Model Definition Language (**SMDL**) in [AD-2]. As the methods `GetFieldValue()` and `SetFieldValue()` only support fields of simple type, complex types have to be accessed by accessing their individual members.

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

Rule 2: Members of strings and arrays are addressed by their 0-based index enclosed in brackets (`"["` and `"]")`.

Examples:

```
MyField.Position[2]
Structure!Field
Class/Array[1]/Structure/Field
```

3.5.3.2 IEntryPointPublisher

```
module Smp
{
   module Management
   {
      interface IEntryPointPublisher : IModel
      {
         EntryPointCollection GetEntryPoints();
         IEntryPoint GetEntryPoint(in String8 name);
      };
   };
};
```

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.

Inheritance Diagram:

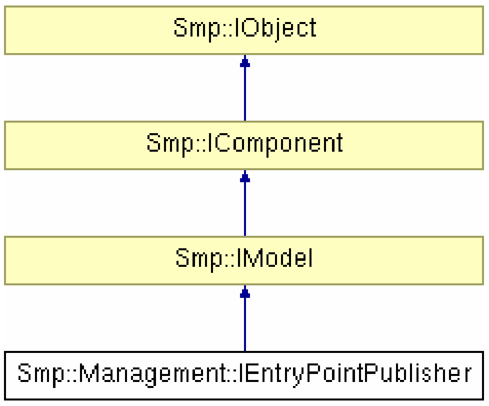

Figure 3-23: IEntryPointPublisher Interface

Remarks:

This is an optional interface. It needs to be implemented for managed models only.

3.5.3.2.1 GetEntryPoint

```
IEntryPoint GetEntryPoint(in String8 name);
```

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

Parameters:

name	Entry point name.

Returns:

Entry point with given name, or null if no entry point with given name could be found.

Exceptions:

None.

Remarks:

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

3.5.3.2.2 GetEntryPoints

```
EntryPointCollection GetEntryPoints();
```

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

Parameters:

None.

Returns:

Collection of entry points.

Exceptions:

None.

Remarks:

The collection may be empty if no entry points exist.

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.

```
module 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.

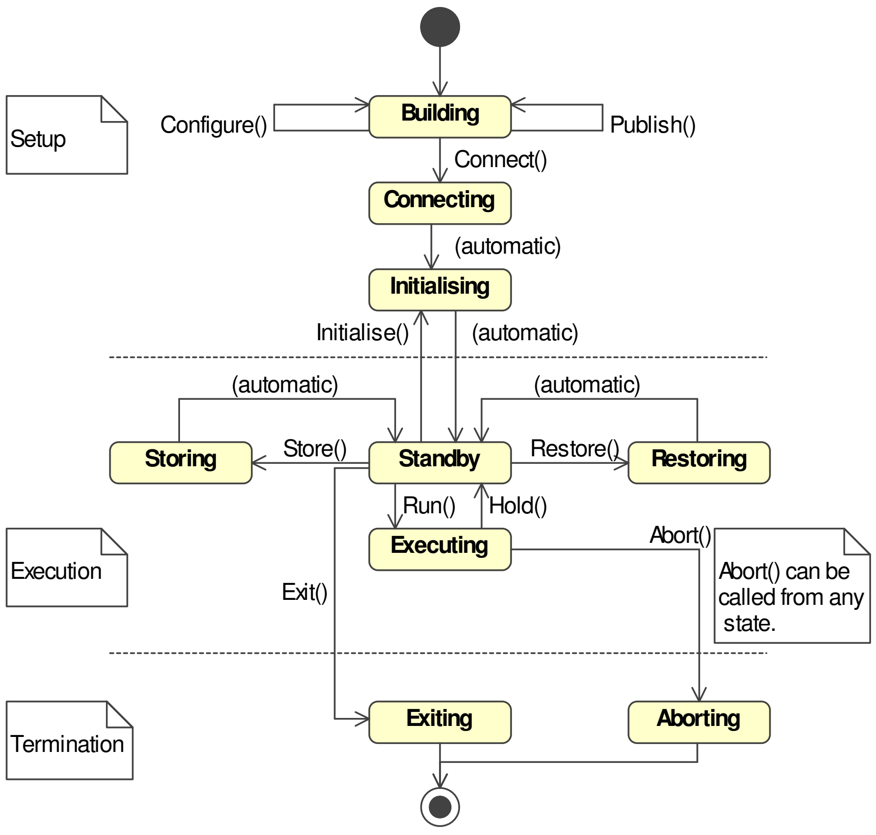

Figure 3-24: Simulation Environment State Diagram with State Transition Methods

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 3-24, and explained in the following subsections. However, when calling a state transition from another state, the simulation environment shall not raise an exception, but ignore the state transition. It may use the Logger service to log a warning message.

3.6.1.1 SimulatorStateKind

This is an enumeration of the available states of the simulator. The Setup phase is split into three different states, the Execution phase has four different states, and the Termination phase has two states.

```
module Smp
{
    enum SimulatorStateKind
    {
        SSK_Building,
        SSK_Connecting,
        SSK_Initialising,
        SSK_Standby,
        SSK_Executing,
        SSK_Storing,
        SSK_Restoring,
        SSK_Exiting,
        SSK_Aborting
    };
};
```

Table 3-2: Simulator States

State name	Description	Enter	Leave
`Building`	In `Building` state, the model hierarchy is created. This is done by an external component, not by the simulator.	This state is entered automatically after the simulation environment has performed its initialisation.	To leave it, call the `Connect()` state transition method.
`Connecting`	In `Connecting` state, the simulation environment traverses the model hierarchy and calls the `Connect()` method of each model.	This state is entered using the `Connect()` state transition	After connecting all models to the simulator, an automatic state transition to the `Initialising` state is performed.
`Initialising`	In `Initialising` state, the simulation environment executes all initialisation entry points in the order they have been added to the simulator using the `AddInitEntryPoint()` method.	This state is either entered automatically after the simulation environment has connected all models to the simulator, or manually from `Standby` state using the `Initialise()` state transition.	After calling all initialisation entry points, an automatic state transition to the `Standby` state is performed.
`Standby`	In `Standby` state, the simulation environment (namely the Time Keeper Service) does not progress simulation time. Only entry points registered relative to Zulu time are executed.	This state is entered automatically from the `Initialising`, `Storing`, and `Restoring` states, or manually from the `Executing` state using the `Hold()` state transition.	To leave this state, call one of the `Run()`, `Store()`, `Restore()`, `Initialise()`, or `Exit()` state transitions.

`Executing`	In `Executing` state, the simulation environment (namely the Time Keeper Service) does progress simulation time. Entry points registered with any of the available time kinds are executed.	This state is entered using the `Run()` state transition.	To leave this state, call the `Hold()` state transition.
`Storing`	In `Storing` state, the simulation environment first stores the values of all fields published with the `State` attribute to storage (typically a file). Afterwards, the `Store()` method of all components implementing the optional `IPersist` interface is called, to allow custom storing of additional information. While in this state, fields published with the `State` attribute must not be modified by the models, to ensure that a consistent set of field values is stored.	This state is entered using the `Store()` state transition.	After storing the simulator state, an automatic state transition to the `Standby` state is performed.
`Restoring`	In `Restoring` state, the simulation environment first restores the values of all fields published with the `State` attribute from storage. Afterwards, the `Restore()` method of all components implementing the optional `IPersist` interface is called, to allow custom restoring of additional information. While in this state, fields published with the `State` attribute must not be modified by the models, to ensure that a consistent set of field values is restored.	This state is entered using the `Restore()` state transition.	After restoring the simulator state, an automatic state transition to the `Standby` state is performed.
`Exiting`	In `Exiting` state, the simulation environment is properly terminating a running simulation.	This state is entered using the `Exit()` state transition.	After exiting, the simulator is in an undefined state.
`Aborting`	In this state, the simulation environment performs an abnormal simulation shut-down.	This state is entered using the `Abort()` state transition.	After aborting, the simulator is in an undefined state.

3.6.1.2 ISimulator

```
module Smp
{
    interface ISimulator : IComposite
    {
        ModelCollection GetModels();
        IModel GetModel(in String8 name);
        void AddModel(in IModel model) raises (DuplicateName);

        ServiceCollection GetServices();
        IService GetService(in String8 name);
        void AddService(in IService service) raises (DuplicateName);

        Services::ILogger GetLogger();
        Services::IScheduler GetScheduler();
        Services::ITimeKeeper GetTimeKeeper();
        Services::IEventManager GetEventManager();

        SimulatorStateKind GetState();

        void Publish();
        void Configure();
        void Connect();
        void Initialise();
        void Run();
        void Hold();
        void Store(in String8 filename);
        void Restore(in String8 filename);
        void Exit();
        void Abort();

        void AddInitEntryPoint(in IEntryPoint entryPoint);
    };
};
```

This interface gives access to the simulation environment state and state transitions. Further, it provides convenience methods to access the root models and simulation services.

Inheritance Diagram:

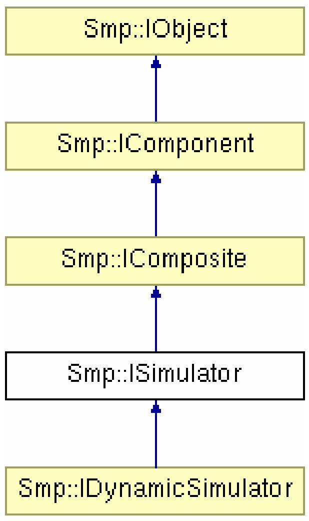

Figure 3-25: ISimulator Interface

Remarks:

This is a mandatory interface that every SMP2 compliant simulation environment has to implement.

3.6.1.2.1 GetModels

```
ModelCollection GetModels();
```

This method returns the collection of root models. Root models are immediate children of the “Models” container that the Simulator provides.

Parameters:

None.

Returns:

Collection of root models.

Exceptions:

None.

Remarks:

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

3.6.1.2.2 GetModel

```
IModel GetModel(in String8 name);
```

Query a root model by name.

Parameters:

name	Name of root model.

Returns:

Root model with the given name, or null if no root model with the given name exists.

Exceptions:

None.

Remarks:

This method does not resolve paths to sub-models, but only returns root models. Use the Resolver service to query sub-models.

3.6.1.2.3 AddModel

```
void AddModel(in IModel model) raises (DuplicateName);
```

Add a root model to the simulator.

Parameters:

model	New model to add to collection of root models, i.e. to the “Models” container.

Returns:

Void.

Exceptions:

This method raises an exception of type `DuplicateName` if the name of the new model conflicts with the name of an existing model.

Remarks:

None.

3.6.1.2.4 GetServices

```
ServiceCollection GetServices();
```

Query for the collection of all services.

Parameters:

None.

Returns:

Collection of service components.

Exceptions:

None.

Remarks:

The collection may be empty if no services exist.

3.6.1.2.5 GetService

```
IService GetService(in String8 name);
```

Query for a service component by its name.

Parameters:

name	Service name.

Returns:

Service with the given name, or null if no service with the given name could be found.

Exceptions:

None.

Remarks:

The returned component is null if no service with the given name could be found. Standard names are defined for the standardised services, while custom services use custom names.

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

The existence of optional services as well as of custom services is not guaranteed, so models should expect to get a null reference.

3.6.1.2.6 AddService

```
void AddService(in IService service) raises (DuplicateName);
```

This method adds a user-defined service to the services collection, i.e. to the “Services” container.

Parameters:

service	User-defines service to add to services collection.

Returns:

Void.

Exceptions:

This method raises an exception of type DuplicateName if the name of the new service conflicts with the name of an existing service.

Remarks:

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

3.6.1.2.7 GetLogger

```
Services::ILogger GetLogger();
```

Query for mandatory logger service.

Parameters:

None.

Returns:

Interface to mandatory logger service.

Exceptions:

None.

Remarks:

This is a type-safe convenience method, to avoid having to use the general GetService() method. For the mandatory services, it is recommended to use the convenience methods.

3.6.1.2.8 GetScheduler

```
Services::IScheduler GetScheduler();
```

Query for mandatory scheduler service.

Parameters:

None.

Returns:

Interface to mandatory scheduler service.

Exceptions:

None.

Remarks:

This is a type-safe convenience method, to avoid having to use the general `GetService()` method. For the mandatory services, it is recommended to use the convenience methods.

3.6.1.2.9 GetTimeKeeper

```
Services::ITimeKeeper GetTimeKeeper();
```

Query for mandatory time keeper service.

Parameters:

None.

Returns:

Interface to mandatory time keeper service.

Exceptions:

None.

Remarks:

This is a type-safe convenience method, to avoid having to use the general `GetService()` method. For the mandatory services, it is recommended to use the convenience methods.

3.6.1.2.10 GetEventManager

```
Services::IEventManager GetEventManager();
```

Query for mandatory event manager service.

Parameters:

None.

Returns:

Interface to mandatory event manager service.

Exceptions:

None.

Remarks:

This is a type-safe convenience method, to avoid having to use the general `GetService()` method. For the mandatory services, it is recommended to use the convenience methods.

3.6.1.2.11 GetState

```
SimulatorStateKind GetState();
```

Returns the current simulator state.

Parameters:

None.

Returns:

Current simulator state.

Exceptions:

None.

Remarks:

None.

3.6.1.2.12 Publish

```
void Publish();
```

This method asks the simulation environment to call the `Publish()` method of all model instances in the model hierarchy which are still in `Created` state.

Parameters:

None.

Returns:

Void.

Exceptions:

None.

Remarks:

This method can only be called when in `Building` state.

This method is typically called by an external component after creating new model instances, typically from information in an SMDL Assembly.

3.6.1.2.13 Configure

```
void Configure();
```

This method asks the simulation environment to call the `Configure()` method of all model instances which are still in `Publishing` state.

Parameters:

None.

Returns:

Void.

Exceptions:

None.

Remarks:

This method can only be called when in `Building` state.

This method is typically called by an external component after setting field values of new model instances, typically using the information in an SMDL Assembly.

3.6.1.2.14 Connect

```
void Connect();
```

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

Parameters:

None.

Returns:

Void.

Exceptions:

None.

Remarks:

This method can only be called when in `Building` state.

This method is typically called by an external component after configuring all model instances.

3.6.1.2.15 Initialise

```
void Initialise();
```

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

Parameters:

None.

Returns:

Void.

Exceptions:

None.

Remarks:

This method can only be called when in `Standby` state.

The entry points will be executed in the order they have been added to the simulator using the `AddInitEntryPoint()` method.

3.6.1.2.16 Run

```
void Run();
```

This method changes from `Standby` to `Executing` state.

Parameters:

None.

Returns:

Void.

Exceptions:

None.

Remarks:

This method can only be called when in `Standby` state.

3.6.1.2.17 Hold

```
void Hold();
```

This method changes from `Executing` to `Standby` state.

Parameters:

None.

Returns:

Void.

Exceptions:

None.

Remarks:

This method can only be called when in `Executing` state.

3.6.1.2.18 Store

```
void Store(in String8 filename);
```

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

Parameters:

filename	Name to use for simulation state vector file.

Returns:

Void.

Exceptions:

None.

Remarks:

This method can only be called when in `Standby` state.

3.6.1.2.19 Restore

```
void Restore(in String8 filename);
```

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

Parameters:

filename	Name of simulation state vector file.

Returns:

Void.

Exceptions:

None.

Remarks:

This method can only be called when in `Standby` state.

3.6.1.2.20 Exit

```
void Exit();
```

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

Parameters:

None.

Returns:

Void.

Exceptions:

None.

Remarks:

This method can only be called when in `Standby` state.

3.6.1.2.21 Abort

```
void Abort();
```

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

Parameters:

None.

Returns:

Void.

Exceptions:

None.

Remarks:

This method can be called from any other state.

3.6.1.2.22 AddInitEntryPoint

```
void AddInitEntryPoint(in IEntryPoint entryPoint);
```

This method can be used to add entry points that shall be executed in the `Initialising` state.

Parameters:

entryPoint	Entry point to execute in `Initialising` state.

Returns:

Void.

Exceptions:

None.

Remarks:

The entry points will be executed in the order they have been added to the simulator.

The `ITask` interface (which is derived from `IEntryPoint`) can be used to add several entry points in a well-defined order.

3.6.1.3 IDynamicSimulator

```
module Smp
{
   interface IDynamicSimulator : ISimulator
   {
        exception DuplicateUuid
        {
            /// Name of factory that tried to register under this Uuid.
            String8 newName;
            /// Name of factory already registered under this Uuid.
            String8 oldName;
        };

      void RegisterFactory(
         in IFactory componentFactory) raises (DuplicateUuid);

      IComponent CreateInstance(in Uuid implUuid);

      IFactory GetFactory(in Uuid implUuid);

      FactoryCollection GetFactories(in Uuid specUuid);
   };
};
```

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

Inheritance Diagram:

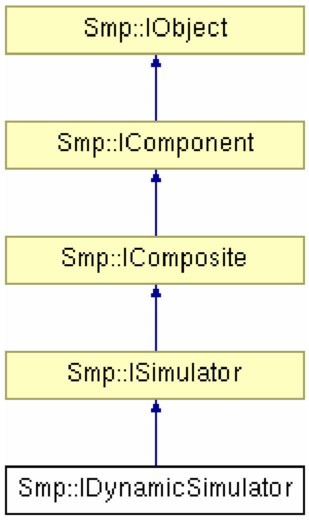

Figure 3-26: IDynamicSimulator Interface

Remarks:

This is an optional interface the simulation environment may implement.

3.6.1.3.1 RegisterFactory

```
void RegisterFactory(
         in IFactory componentFactory) raises (DuplicateUuid);
```

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

Parameters:

componentFactory	Factory to create instance of the component implementation.

Returns:

Void.

Exceptions:

This method raises an exception of type `DuplicateUuid` if another factory has been registered using the same implementation identifier already.

Remarks:

This method is typically called early in the `Building` state to register the available component factories before the hierarchy of model instances is created.

The implementation identifiers of the registered factories need to be unique. Otherwise, an exception of type `DuplicateUuid` is raised.

3.6.1.3.2 CreateInstance

```
IComponent CreateInstance(in Uuid implUuid);
```

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

Parameters:

implUuid	Implementation identifier of the component.

Returns:

New instance of the component with the given implementation identifier.

Exceptions:

None.

Remarks:

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

3.6.1.3.3 GetFactory

```
IFactory GetFactory(in Uuid implUuid);
```

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

Parameters:

implUuid	Implementation identifier of the component.

Returns:

Factory of the component with the given implementation identifier.

Exceptions:

None.

Remarks:

None.

3.6.1.3.4 GetFactories

```
FactoryCollection GetFactories(in Uuid specUuid);
```

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

Parameters:

specUuid	Specification identifier of the component.

Returns:

Collection of factories for the given specification identifier.

Exceptions:

None.

Remarks:

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

3.6.1.4 IFactory

```
module Smp
{
   interface IFactory : IObject
   {
      Uuid GetSpecification();
      Uuid GetImplementation();
      IComponent CreateInstance();
      void DeleteInstance(in IComponent instance);
   };
};
```

This interface is implemented by all component factories.

Inheritance Diagram:

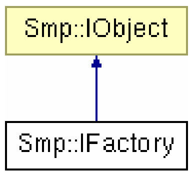

Figure 3-27: IFactory Interface

Remarks:

None.

3.6.1.4.1 GetSpecification

```
Uuid GetSpecification();
```

Get specification identifier of factory.

Parameters:

None.

Returns:

Universally unique identifier of component specification.

Exceptions:

None.

Remarks:

None.

3.6.1.4.2 GetImplementation

```
Uuid GetImplementation();
```

Get implementation identifier of factory.

Parameters:

None.

Returns:

Universally unique identifier of component implementation.

Exceptions:

None.

Remarks:

None.

3.6.1.4.3 CreateInstance

```
IComponent CreateInstance();
```

Create a new instance.

Parameters:

None.

Returns:

Void.

Exceptions:

None.

Remarks:

None.

3.6.1.4.4 DeleteInstance

```
void DeleteInstance(in IComponent instance);
```

Delete an existing instance.

Parameters:

None.

Returns:

Void.

Exceptions:

None.

Remarks:

None.

3.6.1.5 FactoryCollection

```
module Smp
{
   typedef sequence<IFactory> FactoryCollection;
};
```

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

Remarks:

This type is platform specific. For details see the SMP2 Platform Mapping.

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.

3.6.2.1 IPublication

```
module Smp
{
   interface IPublication
   {
   };
};
```

Provide functionality to allow publishing members, including fields and operations.

Remarks:

This interface is platform specific. For details see the SMP2 Platform Mappings.

3.6.3 Service Acquisition

Simulation services are closely related to models: Models need a mechanism to acquire simulation services, and most services interact with models. Further, simulation services are themselves components of the SMP2 Component Model. Therefore, this document puts simulation services into context with models and the simulation environment.

When the simulation environment connects a model, it calls the `Connect()` method of the `IModel` interface, passing it a global reference of `ISimulator`. A model can either immediately use this reference to query services, or store the reference to query services on demand.

Before a Model can call an operation of a Service, the following steps are needed:

1. The Simulator calls the `Connect()` operation of the Model, passing it a reference to itself.
2. The Model calls the `GetService()` operation of the simulator, passing it the name of the required service.
3. The Simulator returns the required service to the Model.
4. The Model calls the desired operation of the Service.
5. The Service returns the desired value to the Model.
6. The Model returns control to the Simulator.

If the model keeps a reference to the service, it can call the service again at any other time.

7. The Model calls the desired operation of the Service.
8. The Service returns the desired value to the Model.

These steps are shown in a sequence diagram in Figure 3-28.

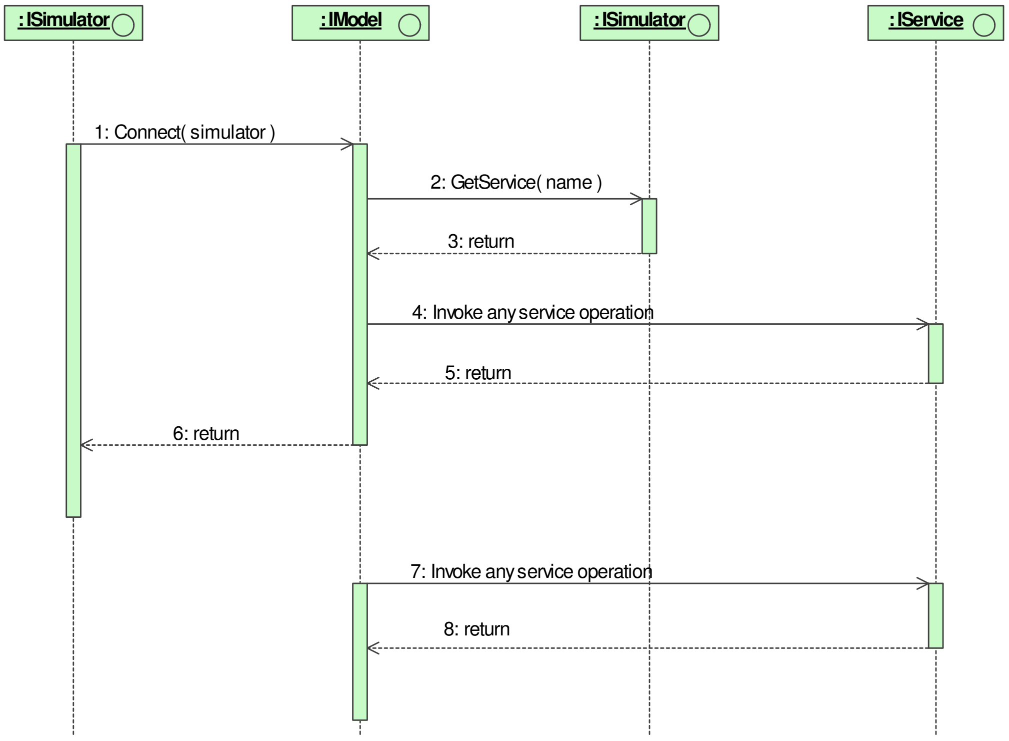

Figure 3-28: Sequence of calls for service acquisition

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.

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

```
module Smp
{
   module Services
   {
      const String8 SMP_Logger          = "Smp_Logger";

      typedef Int32 LogMessageKind;

      interface ILogger : IService
      {
         LogMessageKind GetLogMessageKind(in String8 messageKindName);

         void Log(
            in IObject sender,
            in String8 message,
            in LogMessageKind kind);
      };
   };
};
```

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

Inheritance Diagram:

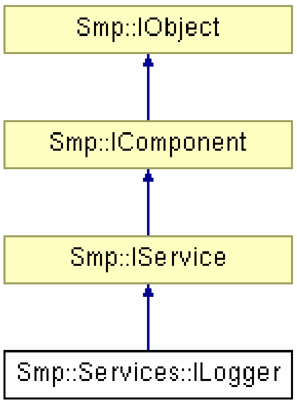

Figure 4-1: ILogger Interface

Remarks:

None.

4.1.1.1.1 GetLogMessageKind

```
LogMessageKind GetLogMessageKind(in String8 messageKindName);
```

Return identifier of log message kind by name. This method can be used for predefined log message kinds, but is especially useful for user-defined log message kinds.

Parameters:

messageKindName	Name of log message kind.

Returns:

Identifier of log message kind.

Exceptions:

None.

Remarks:

It is guaranteed that this method always returns the same id for the same *messageKindName* string.

4.1.1.1.2 Log

```
void Log(in IObject sender, in String8 message, in LogMessageKind kind);
```

This function logs a message to the simulation log file.

Parameters:

sender	Object that sends the message.
message	The message to log.
kind	Kind of message.

Returns:

Void.

Exceptions:

None.

Remarks:

None.

4.1.1.2 Predefined Log Message Kinds

```
module Smp
{
    module Services
    {
        // Identifiers of 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, for example to allow filtering messages by message kind. However, the standard pre-defines a few message kinds that are assumed to be used in most simulations.

Table 4-1: Predefined Log Message Kinds

Message Kind	Id	Description
`Information`	0	The message contains general information.
`Event`	1	The message has been sent from an event, typically from a state transition.
`Warning`	2	The message contains a warning.
`Error`	3	The message has been raised because of an error.
`Debug`	4	The message contains debug information.

4.1.1.3 User-defined Log Message Kinds

With the `GetLogMessageKind()` method, it is possible to add a user-defined lgo message kind to the logger service. The first call of this method with a user-defined message kind name returns a new, unique identifier that can be used as third parameter for the `Log()` method. Further calls of the method `GetLogMessageKind()` with the same user-defined message kind name are guaranteed to return the same identifier again.

This mechanism allows using a user-defined log message kind within several different models, without the need to store the log message kind identifier into a global variable. Further, it assigns a user-readable name to each log message kind, so that e.g. a log message viewer can show log message kinds by name rather than by identifier.

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

4.1.2.1 ITimeKeeper

```
module Smp
{
    module Services
    {
        const String8 SMP_TimeKeeper     = "Smp_TimeKeeper";

        interface ITimeKeeper : IService
        {
            Duration GetSimulationTime();
            DateTime GetEpochTime();
            Duration GetMissionTime();
            DateTime GetZuluTime();

            void SetEpochTime(in DateTime epochTime);
            void SetMissionStart(in DateTime missionStart);
            void SetMissionTime(in Duration missionTime);
        };
    };
};
```

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

Inheritance Diagram:

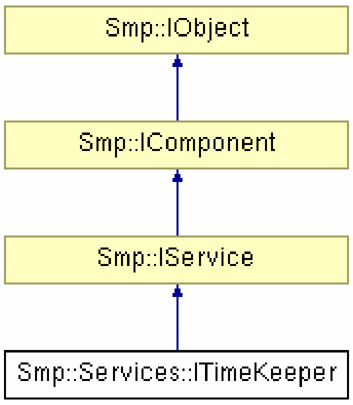

Figure 4-2: ITimeKeeper Interface

Remarks:

None.

4.1.2.1.1 GetSimulationTime

```
Duration GetSimulationTime();
```

Simulation time is a relative time that starts at 0.

Parameters:

None.

Returns:

Current simulation time.

Exceptions:

None.

Remarks:

None.

4.1.2.1.2 GetEpochTime

```
DateTime GetEpochTime();
```

Epoch time is an absolute time with a fixed offset to simulation time.

Parameters:

None.

Returns:

Current epoch time.

Exceptions:

None.

Remarks:

Epoch time typically progresses with simulation time, but can be changed with `SetEpochTime`.

4.1.2.1.3 GetMissionTime

```
Duration GetMissionTime();
```

Mission time is a relative time with a fixed offset to simulation time.

Parameters:

None.

Returns:

Current mission time.

Exceptions:

None.

Remarks:

Mission time typically progresses with simulation time, but can be changed with the two methods `SetMissionTime` and `SetMissionStart`. Further, mission time is updated when changing epoch time with `SetEpochTime`.

4.1.2.1.4 GetZuluTime

```
DateTime GetZuluTime();
```

Zulu time is a system dependent time and not related to simulation time.

Parameters:

None.

Returns:

Current Zulu time.

Exceptions:

None.

Remarks:

Zulu time is typically related to the system clock of the computer.

4.1.2.1.5 SetEpochTime

```
void SetEpochTime(in DateTime epochTime);
```

Changes the offset between simulation time and epoch time.

Parameters:

epochTime	New epoch time.

Returns:

Void.

Exceptions:

None.

Remarks:

Calling this method shall raise a global `EpochTimeChanged` event in the Event Manager.

4.1.2.1.6 SetMissionStart

```
void SetMissionStart(in DateTime missionStart);
```

Changes the offset between simulation time and mission time.

Parameters:

missionStart	Mission start date and time.

Returns:

Void.

Exceptions:

None.

Remarks:

The mission time itself will be calculated as the offset between the current epoch time and the given mission start:

```
MissionTime = EpochTime - MissionStart
```

Calling this method shall raise a global `MissionTimeChanged` event in the Event Manager.

4.1.2.1.7 SetMissionTime

```
void SetMissionTime(in Duration missionTime);
```

Changes the offset between simulation time and epoch time.

Parameters:

missionTime	New mission time.

Returns:

Void.

Exceptions:

None.

Remarks:

Calling this method shall raise a global `MissionTimeChanged` event in the Event Manager.

4.1.2.2 DateTime

```
module Smdl
{
        typedef Int64      DateTime;    ///< Absolute time in Nanoseconds
};
```

This type is used for absolute time values. It specifies a time in nanoseconds, relative to a reference time. The following holds for `DateTime`:

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

2. Time is stored as a signed 64-bit integer value, relative to the reference time (01.01.2000, 12:00, Modified Julian Date (**MJD**) 2000+0.5).

3. Positive values correspond to times after the reference time, and negative values correspond to time values before the reference time.

This allows specifying time values roughly between 1710 and 2290.

With this definition, the `DateTime` type is compatible with the `Duration` type.

4.1.2.3 Duration

```
module Smdl
{
      typedef Int64     Duration;    ///< Duration in Nanoseconds
};
```

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

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

2. Duration is stored as a signed 64-bit integer value.

3. Positive values correspond to positive durations, and negative values correspond to negative durations.

This allows specifying duration values roughly between -290 years and 290 years.

With this definition, the `Duration` type is compatible with the `DateTime` type.

4.1.2.4 TimeKind

```
module Smp
{
    module Services
    {
        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.

4.1.2.4.1 Simulation Time

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.1.2.4.2 Epoch Time

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

1. Epoch time is a value measured in nanoseconds, which is the lowest level of granularity supported for time in SMP2.
2. Epoch time is returned as a signed 64-bit integer value, relative to the epoch reference time (01.01.2000, 12:00, Modified Julian Date 2000+0.5). This allows specifying time values roughly between 1710 and 2290.
3. Epoch time can be queried using the `GetEpochTime()` method of the time keeper (via the `ITimeKeeper` interface).
4. Epoch time is initialised to 0 (i.e. 01.01.2000, 12:00) at the beginning of the initialisation phase, but can be changed already before entering the execution phase.
5. Epoch time is progressed linearly with simulation time (i.e. with a fixed offset to simulation time). Using the `SetEpochTime()` method of the time keeper (via the `ITimeKeeper` interface), the offset between simulation time and epoch time can be changed.
6. When storing a state vector, epoch time (i.e. its offset to simulation time) is stored as well.
7. When restoring a state vector, epoch time (i.e. its offset to simulation time) is restored as well.

4.1.2.4.3 Mission Time

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

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

4.1.2.4.4 Zulu Time

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

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

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

1. Zulu time is measured in nanoseconds.
2. Zulu time is returned as a signed 64-bit integer value, relative to the epoch reference time (see 4.1.2.4.2).
3. Zulu time represents the current time at Greenwich, England, called as well UTC or GMT.

As Zulu time is not managed by the time keeper service, but provided based on an external clock (typically the computer clock), it is not related to simulation time, and progresses independently of the state of the simulation environment. When a simulator interfaces to an external system, for example a ground station or some Hardware-In-The-Loop (**HITL**), Zulu time is often used as a time stamp.

4.1.3 Scheduler

The scheduler service calls entry points of models based on events triggered by one of the four time kinds.

4.1.3.1 IScheduler

```
module Smp
{
    module Services
    {
        const String8 SMP_Scheduler      = "Smp_Scheduler";

        interface IScheduler : IService
        {
            exception InvalidEventId { EventId event; };

            void AddImmediateEvent(
                in IEntryPoint entryPoint);
            EventId AddSimulationTimeEvent(
                in IEntryPoint entryPoint,
                in Duration simulationTime,
                in Duration cycleTime,
                in Int64 count);
            EventId AddMissionTimeEvent(
                in IEntryPoint entryPoint,
                in Duration missionTime,
                in Duration cycleTime,
                in Int64 count);
            EventId AddEpochTimeEvent(
                in IEntryPoint entryPoint,
                in Duration epochTime,
                in Duration cycleTime,
                in Int64 count);
            EventId AddZuluTimeEvent(
                in IEntryPoint entryPoint,
                in Duration zuluTime,
                in Duration cycleTime,
                in Int64 count);

            void SetEventSimulationTime(
                in EventId event,
                in Duration simulationTime)    raises (InvalidEventId);
            void SetEventMissionTime(
                in EventId event,
                in Duration missionTime)       raises (InvalidEventId);
            void SetEventEpochTime(
                in EventId event,
                in DateTime epochTime)         raises (InvalidEventId);
            void SetEventZuluTime(
                in EventId event,
                in DateTime zuluTime)          raises (InvalidEventId);
            void SetEventCycleTime(
                in EventId event,
                in Duration cycleTime)         raises (InvalidEventId);
            void SetEventCount(
                in EventId event,
                in Int64 count)                raises (InvalidEventId);

            void RemoveEvent(in EventId event) raises (InvalidEventId);
        };
    };
};
```

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

Inheritance Diagram:

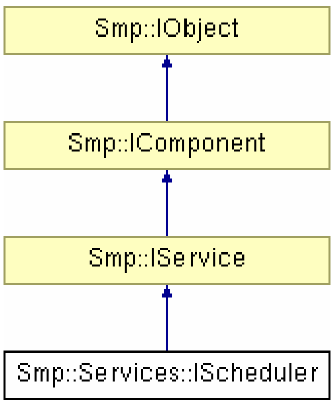

Figure 4-3: IScheduler Interface

Remarks:

None.

4.1.3.1.1 AddImmediateEvent

```
void AddImmediateEvent(in IEntryPoint entryPoint);
```

Add an immediate event to the scheduler.

Parameters:

entryPoint	Entry point to call from event.

Returns:

Void.

Exceptions:

None.

Remarks:

None.

4.1.3.1.2 AddSimulationTimeEvent

```
EventId AddSimulationTimeEvent(
                in IEntryPoint entryPoint,
                in Duration simulationTime,
                in Duration cycleTime,
                in Int64 count);
```

Add event to scheduler that is called based on simulation time.

Parameters:

entryPoint	Entry point to call from event.
simulationTime	Duration from now when to trigger the event for the first time.
cycleTime	Duration between two triggers of the event.
count	Number of times the event shall be repeated, or 0 for a single event, or -1 for no limit.

Returns:

Event identifier that can be used to change or remove event.

Exceptions:

None.

Remarks:

An event with `count=0` is not cyclic. It will be removed automatically after is has been triggered.

An event with `count>0` is cyclic, and will be repeated `count` times. Therefore, it will be called `count+1` times, and then it will be removed automatically.

An event with `count=-1` is cyclic as well, but it will be triggered forever, unless it is removed from the scheduler using the `RemoveEvent()` method.

For a cyclic event, the *cycleTime* needs to be positive. For non-cyclic events, it is ignored.

The *simulationTime* must not be negative. Otherwise, the event will never be executed, but immediately removed.

4.1.3.1.3 AddMissionTimeEvent

```
EventId AddMissionTimeEvent(
                in IEntryPoint entryPoint,
                in Duration missionTime,
                in Duration cycleTime,
                in Int64 count);
```

Add event to scheduler that is called based on mission time.

Parameters:

entryPoint	Entry point to call from event.
missonTime	Absolute mission time when to trigger the event for the first time.
cycleTime	Duration between two triggers of the event.
count	Number of times the event shall be repeated, or 0 for a single event, or -1 for no limit.

Returns:

Event identifier that can be used to change or remove event.

Exceptions:

None.

Remarks:

An event with `count=0` is not cyclic. It will be removed automatically after is has been triggered.

An event with `count>0` is cyclic, and will be repeated `count` times. Therefore, it will be called `count+1` times, and then it will be removed automatically.

An event with `count=-1` is cyclic as well, but it will be triggered forever, unless it is removed from the scheduler using the `RemoveEvent()` method.

For a cyclic event, the *cycleTime* needs to be positive. For non-cyclic events, it is ignored.

The *missionTime* must not be before the current mission tiem of the `ITimeKeeper` service. Otherwise, the event will never be executed, but immediately removed.

4.1.3.1.4 AddEpochTimeEvent

```
EventId AddEpochTimeEvent(
                in IEntryPoint entryPoint,
                in DateTime epochTime,
                in Duration cycleTime,
                in Int64 count);
```

Add event to scheduler that is called based on epoch time.

Parameters:

entryPoint	Entry point to call from event.
epochTime	Epoch time when to trigger the event for the first time.
cycleTime	Duration between two triggers of the event.
count	Number of times the event shall be repeated, or 0 for a single event, or -1 for no limit.

Returns:

Event identifier that can be used to change or remove event.

Exceptions:

None.

Remarks:

An event with `count=0` is not cyclic. It will be removed automatically after is has been triggered.

An event with `count>0` is cyclic, and will be repeated `count` times. Therefore, it will be called `count+1` times, and then it will be removed automatically.

An event with `count=-1` is cyclic as well, but it will be triggered forever, unless it is removed from the scheduler using the `RemoveEvent()` method.

For a cyclic event, the *cycleTime* needs to be positive. For non-cyclic events, it is ignored.

The *epochTime* must not be before the current epoch time of the `ITimeKeeper` service. Otherwise, the event will never be executed, but immediately removed.

4.1.3.1.5 AddZuluTimeEvent

```
EventId AddZuluTimeEvent(
                in IEntryPoint entryPoint,
                in DateTime zuluTime,
                in Duration cycleTime,
                in Int64 count);
```

Add event to scheduler that is called based on Zulu time.

Parameters:

entryPoint	Entry point to call from event.
zuluTime	Absolute (Zulu) time when to trigger the event for the first time.
cycleTime	Duration between two triggers of the event.
count	Number of times the event shall be repeated, or 0 for a single event, or -1 for no limit.

Returns:

Event identifier that can be used to change or remove event.

Exceptions:

None.

Remarks:

An event with `count=0` is not cyclic. It will be removed automatically after is has been triggered.

An event with `count>0` is cyclic, and will be repeated `count` times. Therefore, it will be called `count+1` times, and then it will be removed automatically.

An event with `count=-1` is cyclic as well, but it will be triggered forever, unless it is removed from the scheduler using the `RemoveEvent()` method.

For a cyclic event, the *cycleTime* needs to be positive. For non-cyclic events, it is ignored.

The *zuluTime* must not be before the current Zulu time of the `ITimeKeeper` service. Otherwise, the event will never be executed, but immediately removed.

4.1.3.1.6 SetEventSimulationTime

```
void SetEventSimulationTime(
                in EventId event,
                in Duration simulationTime)
     raises (InvalidEventId);
```

Update when an existing event on the scheduler shall be triggered.

Parameters:

event	Event identifier of the event to remove.
simulationTime	Duration from now when to trigger the event.

Returns:

Void.

Exceptions:

When the given *event* is not a valid identifier of a scheduler event, the method throws an exception of type `InvalidEventId`.

Remarks:

The *simulationTime* must not be negative. Otherwise, the event will never be executed, but immediately removed.

4.1.3.1.7 SetEventMissionTime

```
void SetEventMissionTime(
                in EventId event,
                in Duration missionTime)
     raises (InvalidEventId);
```

Update when an existing event on the scheduler shall be triggered.

Parameters:

event	Event identifier of the event to remove.
missionTime	Absolute mission time when to trigger the event.

Returns:

Void.

Exceptions:

When the given *event* is not a valid identifier of a scheduler event, the method throws an exception of type `InvalidEventId`.

Remarks:

The *missionTime* must not be before the current mission tiem of the `ITimeKeeper` service. Otherwise, the event will never be executed, but immediately removed.

4.1.3.1.8 SetEventEpochTime

```
void SetEventEpochTime(
                in EventId event,
                in DateTime epochTime)
     raises (InvalidEventId);
```

Update when an existing event on the scheduler shall be triggered.

Parameters:

event	Event identifier of the event to remove.
epochTime	Epoch time when to trigger the event.

Returns:

Void.

Exceptions:

When the given *event* is not a valid identifier of a scheduler event, the method throws an exception of type `InvalidEventId`.

Remarks:

The *epochTime* must not be before the current epoch time of the `ITimeKeeper` service. Otherwise, the event will never be executed, but immediately removed.

4.1.3.1.9 SetEventZuluTime

```
void SetEventZuluTime(
                in EventId event,
                in DateTime zuluTime)
     raises (InvalidEventId);
```

Update when an existing event on the scheduler shall be triggered.

Parameters:

event	Event identifier of the event to remove.
zuluTime	Absolute (Zulu) time when to trigger the event.

Returns:

Void.

Exceptions:

When the given *event* is not a valid identifier of a scheduler event, the method throws an exception of type `InvalidEventId`.

Remarks:

The *zuluTime* must not be before the current Zulu time of the `ITimeKeeper` service. Otherwise, the event will never be executed, but immediately removed.

4.1.3.1.10 SetEventCycleTime

```
void SetEventCycleTime(
                in EventId event,
                in Duration cycleTime)
     raises (InvalidEventId);
```

Update cycle time of an existing event on the scheduler.

Parameters:

event	Event identifier of the event to remove.
cycleTime	Duration between two triggers of the event.

Returns:

Void.

Exceptions:

When the given *event* is not a valid identifier of a scheduler event, the method throws an exception of type `InvalidEventId`.

Remarks:

For a cyclic event, the *cycleTime* needs to be positive. For non-cyclic events, it is ignored.

4.1.3.1.11 SetEventCount

```
void SetEventCount(
                in EventId event,
                in Int64 count)
     raises (InvalidEventId);
```

Update the count of an existing event on the scheduler.

Parameters:

event	Event identifier of the event to remove.
count	Number of times the event shall be repeated, or 0 for a single event, or -1 for no limit.

Returns:

Void.

Exceptions:

When the given *event* is not a valid identifier of a scheduler event, the method throws an exception of type `InvalidEventId`.

Remarks:

An event with `count=0` is not cyclic. It will be removed automatically after is has been triggered.

An event with `count>0` is cyclic, and will be repeated `count` times. Therefore, it will be called `count+1` times, and then it will be removed automatically.

An event with `count=-1` is cyclic as well, but it will be triggered forever, unless it is removed from the scheduler using the `RemoveEvent()` method.

For a cyclic event, the *cycleTime* needs to be positive. For non-cyclic events, it is ignored.

4.1.3.1.12 RemoveEvent

```
void RemoveEvent(in EventId event);
```

Remove an event from the scheduler.

Parameters:

event	Event identifier of the event to remove.

Returns:

Void.

Exceptions:

When the given *event* is not a valid identifier of a scheduler event, the method throws an exception of type `InvalidEventId`..

Remarks:

An event with `count=0` is removed automatically after it has been triggered.

4.1.3.2 ITask

```
module Smp
{
    module Services
    {
        interface ITask : IEntryPoint
        {
            EntryPointCollection GetEntryPoints();
            void AddEntryPoint(in IEntryPoint entryPoint);
        };
    };
};
```

This interface extends `IEntryPoint` to allow scheduling tasks.

Inheritance Diagram:

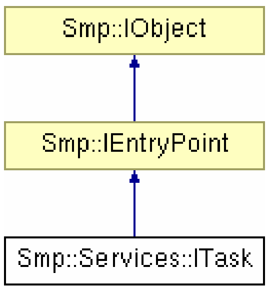

Figure 4-4: ITask Interface

Remarks:

A Task is an ordered collection of entry points.

4.1.3.2.1 GetEntryPoints

```
EntryPointCollection GetEntryPoints();
```

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

Parameters:

None.

Returns:

Collection of entry points.

Exceptions:

None.

Remarks:

The collection may be empty if no entry points exist.

4.1.3.2.2 AddEntryPoint

```
void AddEntryPoint(in IEntryPoint entryPoint);
```

Add an entry point to the task.

Parameters:

entryPoint	Entry point to add to task.

Returns:

Void.

Exceptions:

None.

Remarks:

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

```
module Smp
{
    module Services
    {
        const String8 SMP_EventManager   = "Smp_EventManager";

        typedef Int64 EventId;

        exception InvalidEventId
        {
            /// Identifier of invalid event.
            EventId event;
        };

        interface IEventManager : IService
        {
            exception AlreadySubscribed
            {
                String8 eventName;
                IEntryPoint entryPoint;
            };

            exception NotSubscribed
            {
                String8 eventName;
                IEntryPoint entryPoint;
            };

            EventId GetEventId(in String8 eventName);

            void Subscribe(in EventId event, in IEntryPoint entryPoint)
                raises (InvalidEventId, AlreadySubscribed);

            void Unsubscribe(in EventId event, in IEntryPoint entryPoint)
                raises (InvalidEventId, NotSubscribed);

            void Emit(in EventId event) raises (InvalidEventId);
        };
    };
};
```

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

Inheritance Diagram:

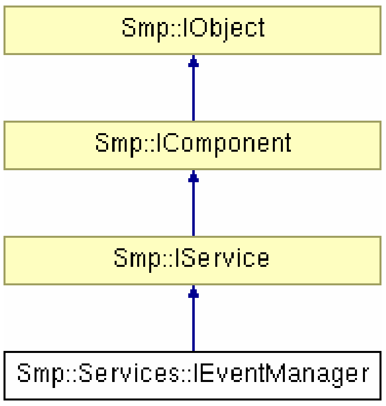

Figure 4-5: IComposite Interface

Remarks:

None.

4.1.4.1.1 GetEventId

```
EventId GetEventId(in String8 eventName);
```

Get unique event identifier for an event name.

Parameters:

eventName	Name of the global event.

Returns:

Event identifier for global event with given name.

Exceptions:

None.

Remarks:

It is guaranteed that this method will always return the same value when called with the same event name. This holds for predefined event names as well as for user-defined events.

4.1.4.1.2 Subscribe

```
void Subscribe(in EventId event, in IEntryPoint entryPoint)
       raises (InvalidEventId, AlreadySubscribed);
```

Subscribe entry point to a global event.

Parameters:

event	Event identifier of global event to subscribe to.
entryPoint	Entry point to subscribe to global event.

Returns:

Void.

Exceptions:

This method raises an exception of type `InvalidEventId` when called with an invalid event identifier. When the entry point is already subscribed to the same event, an exception of type `AlreadySubscribed` is raised.

Remarks:

An entry point can only be subscribed once to an event.

4.1.4.1.3 Unsubscribe

```
void Unsubscribe(in EventId event, in IEntryPoint entryPoint)
         raises (InvalidEventId, NotSubscribed);
```

Unsubscribe entry point from a global event.

Parameters:

event	Event identifier of global event to unsubscribe from.
entryPoint	Entry point to unsubscribe from global event.

Returns:

Void.

Exceptions:

This method raises an exception of type `InvalidEventId` when called with an invalid event identifier. When the entry point is not subscribed to the event, an exception of type `NotSubscribed` is raised.

Remarks:

An entry point can only be unsubscribed from an event when it has been subscribed earlier `Subscribe`.

4.1.4.1.4 Emit

```
void Emit(in EventId event) raises (InvalidEventId);
```

Emit a global event.

Parameters:

event	Event identifier of global event to emit.

Returns:

Void.

Exceptions:

None.

Remarks:

This will call all entry points that are subscribed to the global event with the given identifier. Entry points will be called synchronously in the order they have been subscribed to the global event.

4.1.4.2 Predefined Event Kinds

```
module Smp
{
  module 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.
    const EventId SMP_LeaveInitialisingId  =  3; ///< Leave Initialising.
    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 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.

The names and ids of the predefined event kinds are as listen in Table 4-2. As most of these events relate to state changes, the state diagram of the simulation environment is shown in Figure 3-24: Simulation Environment State Diagram with State Transition Methods.

Table 4-2: Pre-defined global Event Names and Ids of the Event Manager

Event Name	Id	Description
`LeaveConnecting`	1	This event is raised when leaving the `Connecting` state with an automatic state transition to `Initializing` state.
`EnterInitialising`	2	This event is raised when entering the `Initialising` state with an automatic state transition from `Connecting` state, or with the `Initialise()` state transition.
`LeaveInitialising`	3	This event is raised when leaving the `Initialising` state with an automatic state transition to `Standby` state.
`EnterStandby`	4	This event is raised when entering the `Standby` state with an automatic state transition from `Initialising, Storing` or `Restoring` state, or with the `Hold()` state transition command from `Executing` state.
`LeaveStandby`	5	This event is raised when leaving the Standby state with the `Run()` state transition command to `Executing` state, with the `Store()` state transition command to `Storing` state, with the `Restore()` state transition command to `Restoring` state, or with the `Initialise()` state transition command to `Initialising` state.
`EnterExecuting`	6	This event is raised when entering the `Executing` state with the `Run()` state transition command from `Standby` state.
`LeaveExecuting`	7	This event is raised when leaving the `Executing` state with the `Hold()` state transition command to `Standby` state.
`EnterStoring`	8	This event is raised when entering the `Storing` state with the `Store()` state transition command from `Standby` state.
`LeaveStoring`	9	This event is raised when leaving the `Storing` state with an automatic state transition to `Standby` state.
`EnterRestoring`	10	This event is raised when entering the `Restoring` state with the `Restore()` state transition command from `Standby` state.
`LeaveRestoring`	11	This event is raised when leaving the `Restoring` state with an automatic state transition to `Standby` state.
`EnterExiting`	12	This event is raised when entering the `Exiting` state with the `Exit()` state transition command from `Standby` state.
`EnterAborting`	13	This event is raised when entering the `Aborting` state with the `Abort()` state transition command from any other state.
`EpochTimeChanged`	14	This event is raised when changing the epoch time with the `SetEpochTime()` method of the time keeper service.
`MissionTimeChanged`	15	This event is raised when changing the mission time with one of the `SetMissionTime()` and `SetMissionStart()` methods of the time keeper service.

User-defined event ids can be generated using the `GetEventId()` method, which will return the same identifier every time it is called with the identical event name.

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

```
module Smp
{
    module Services
    {
        const String8 SMP_Resolver       = "Smp_Resolver";

        interface IResolver : IService
        {
            IComponent ResolveAbsolute(in String8 absolutePath);

            IComponent ResolveRelative(in String8 relativePath,
                                       in IComponent sender);
        };
    };
};
```

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

Inheritance Diagram:

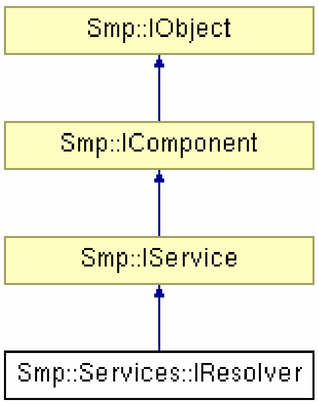

Figure 4-6: IResolver Interface

Remarks:

None.

4.2.1.1.1 ResolveAbsolute

```
IComponent ResolveAbsolute(in String8 absolutePath);
```

Resolve reference to component via absolute path.

Parameters:

absolutePath	Absolute path to component in simulation.

Returns:

Component identified by path, or null if no component with the given path could be found.

Exceptions:

None.

Remarks:

An absolute path contains the name of either the “`Models`” or the “`Services`” container, but not the name of the simulator, although the simulator itself is the top-level component. This allows keeping names as short as possible, and avoids a dependency on the name of the simulator itself.

4.2.1.1.2 ResolveRelative

```
IComponent ResolveRelative(in String8 relativePath, in IComponent sender);
```

Resolve reference to component via relative path.

Parameters:

relativePath	Relative path to component in simulation.
sender	Component that asks for resolving the reference.

Returns:

Component identified by path, or null if no component with the given path could be found.

Exceptions:

None.

Remarks:

None.

4.2.1.2 Component Paths

The Resolver can be used to resolve component references by name, either by a full path or by a relative path with respect to a given component. Therefore, it needs to be specified how path names have to be built. As all components in the tree of a simulation have a name, all that needs to be specified is how these names compose a path name, and how the parent component is identified.

Rule 1: Component names are assembled with one of the following characters: `"\", "/", "!"`

Rule 2: The parent component is specified by the following string: `".."`

Examples:

```
Models/Satellite/Receivers/Receiver1
Services!Logger
..\..\Transmitters\Transmitter4
```

This Page is Intentionally left Blank

A COMPONENT MODEL IDL CODE

This appendix contains the complete CORBA IDL Code that has been used for the code snippets in this document. In addition to the code shown already, it contains documentation and some forward references.

The CORBA IDL file has been compiled with idlj.exe compiler from the Java Development Kit (**JDK**) version 1.5.0 on Microsoft Windows, and with the omniidl compiler from omniORB on Linux.

```
//
// Title:
//      ComponentModel.idl
//
// Project:
//      Simulation Model Portability (SMP) Standard 2.0
//      Component Model (CM)
//      Version 1.2 from October 28th, 2005
//
// Origin:
//      Simulation Model Portability (SMP) Configuration Control Board (CCB)
//
// Purpose:
//      This file defines the SMP2 Component Model in CORBA IDL.
//      This includes simple types, AnySimple, Uuid, and String8
//      as well as all Interfaces and Exceptions.
//
// Author:
//      Peter Ellsiepen, Peter Fritzen
//

// ----------------------------------------------------------------------------
// ---------------------------- Primitive types -------------------------------
// ----------------------------------------------------------------------------

/// SMP standard types and interfaces.
/// This namespace contains all types used within the SMP Component Model.
/// This includes simple types, derived types, interfaces and exceptions.
/// Some specialized interfaces and exceptions are defined in nested namespaces.
module Smp
{
    // Character type that is used as well by strings.
    typedef char                Char8;    ///< 8 bit character type.

    // Boolean type that is either <code>true</code> or <code>false</code>.
    typedef boolean             Bool;     ///<  Boolean with true and false.

    // Integer types
    typedef octet               Int8;     ///<  8 bit   signed integer type.
    typedef octet               UInt8;    ///<  8 bit unsigned integer type.
    typedef short               Int16;    ///< 16 bit   signed integer type.
    typedef unsigned short      UInt16;   ///< 16 bit unsigned integer type.
    typedef long                Int32;    ///< 32 bit   signed integer type.
    typedef unsigned long       UInt32;   ///< 32 bit unsigned integer type.
    typedef long long           Int64;    ///< 64 bit   signed integer type.
    typedef unsigned long long UInt64;    ///< 64 bit unsigned integer type.

    // Floating point types
    typedef float               Float32;  ///< Single-precision float type.
    typedef double              Float64;  ///< Double-precision float type.

    // Date and time types
    typedef Int64               Duration; ///< Duration in Nanoseconds.
    typedef Int64               DateTime; ///< Relative to MJD2000+0.5.
};

// ----------------------------------------------------------------------------
// --------------------------- AnySimple type ---------------------------------
// ----------------------------------------------------------------------------

module 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 Boolean 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.
    };

    /// Union of simple type values
    union AnySimple switch (SimpleTypeKind)
    {
        case ST_Char8:    Char8    char8Value;    ///< Value for ST_Char8.
        case ST_Bool:     Bool     boolValue;     ///< Value for ST_Bool.

        case ST_Int8:     Int8     int8Value;     ///< Value for ST_Int8.
        case ST_UInt8:    UInt8    uint8Value;    ///< Value for ST_UInt8.
        case ST_Int16:    Int16    int16Value;    ///< Value for ST_Int16.
        case ST_UInt16:   UInt16   uint16Value;   ///< Value for ST_UInt16.
        case ST_Int32:    Int32    int32Value;    ///< Value for ST_Int32.
        case ST_UInt32:   UInt32   uint32Value;   ///< Value for ST_UInt32.
        case ST_Int64:    Int64    int64Value;    ///< Value for ST_Int64.
        case ST_UInt64:   UInt64   uint64Value;   ///< Value for ST_UInt64.

        case ST_Float32:  Float32  float32Value;  ///< Value for ST_Float32.
        case ST_Float64:  Float64  float64Value;  ///< Value for ST_Float64.

        case ST_Duration: Duration durationValue; ///< Value for ST_Duration.
        case ST_DateTime: DateTime datetimeValue; ///< Value for ST_DateTime.
    };
};

// ---------------------------------------------------------------------------
// --------------------------- AnySimpleArray type ---------------------------
// ---------------------------------------------------------------------------

module Smp
{
    /// Array of AnySimple values.
    typedef sequence<AnySimple> AnySimpleArray;
};

// ---------------------------------------------------------------------------
// --------------------------- Uuid type -------------------------------------
// ---------------------------------------------------------------------------

module 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
    };
};

// ---------------------------------------------------------------------------
// --------------------------- String8 type ----------------------------------
// ---------------------------------------------------------------------------
```

```
module Smp
{
   typedef string String8;  ///< String of 8 bit characters.
};

// ---------------------------------------------------------------------------
// --------------------------- Exceptions ------------------------------------
// ---------------------------------------------------------------------------

module Smp
{
    // Forward declaration because of circular references
    interface IObject;

    /// Invalid object name.
    /// This exception is raised when trying to set an object's name to an
    /// invalid name. Names
    ///          - must not be empty,
    ///          - must not exceed 32 characters in size,
    ///          - must start with a letter, and
    ///          - must only contain letters, digits, the underscore ("_")
    ///            and brackets ("[" and "]").
    exception InvalidObjectName
    {
        /// Name that is not valid.
        String8 objectName;
    };

    /// Duplicate name.
    /// This exception is raised when trying to add an object to a collection
    /// of objects which have to have unique names, but another object with
    /// the same name does exist. This would lead to duplicate names.
    exception DuplicateName
    {
        /// Name that already exists in the collection.
        String8 name;
    };

    /// Invalid type of AnySimple.
    /// This exception is raised when trying to use an AnySimple of wrong type.
    exception InvalidAnyType
    {
        /// Type that is not valid.
        SimpleTypeKind invalidType;
        /// Type that was expected.
        SimpleTypeKind expectedType;
    };

    /// Invalid type of an Object.
    /// This exception is raised when trying to pass an object of wrong type.
    /// @remarks This can happen when adding a component to a container or
    ///          reference which is semantically typed by a specific type
    ///          derived from IComponent.
    exception InvalidObjectType
    {
        /// Object that is not of valid type.
        IObject invalidObject;
    };
};

// ---------------------------------------------------------------------------
// Objects: IObject, IEntryPoint, IEventSink, IEventSource
// ---------------------------------------------------------------------------

module Smp
{
    // Forward declaration because of circular references
    interface IComponent;

    // -----------------------------------------------------------------------
    // IObject
    // -----------------------------------------------------------------------

    /// Base interface for all objects.
    /// This is the base interface for all other SMP interfaces.
    /// While most interfaces derive from IComponent, which itself is derived
    /// from IObject, some objects (including IEntryPoint, IEventSink, and
    /// IEventSource) are directly derived from IObject.
    /// @remarks The two methods of this interface ensure that all SMP objects
    ///          can be shown with a name, and with an optional description.
    interface IObject
    {
        /// Returns the name of the object ("property getter").
        /// Applications may display the name as user readable object
        /// identification.
        /// @return  Name of object.
        /// @remarks Names
        ///          - must be unique within their context,
        ///          - must not be empty,
        ///          - must not exceed 32 characters in size,
        ///          - must start with a letter, and
        ///          - must only contain letters, digits, the underscore ("_")
        ///            and brackets ("[" and "]").
        String8 GetName();

        /// Returns the description of the object ("property getter").
        /// Applications may display the description as additional
        /// information on the object.
        /// @return  Description of object.
        /// @remarks Descriptions are optional and may be empty.
        String8 GetDescription();
    };

    // -------------------------------------------------------------------------
    // IEntryPoint
    // -------------------------------------------------------------------------

    /// Entry point for IScheduler or IEventManager.
    /// Provide a notification method (event handler) that can be called
    /// by the Scheduler or Event Manager when an event is emitted.
    interface IEntryPoint : IObject
    {
        /// Entry point owner.
        /// This method returns the Component that owns the entry point.
        /// @return  Owner of entry point.
        /// @remarks This is required to be able to store and later restore
        ///          entry points.
        IComponent GetOwner();

        /// Entry point execution.
        /// This method is called when an associated event is emitted.
        /// @remarks Models providing entry points must ensure that these
        ///          entry points do not throw exceptions.
        void Execute();
    };

    /// Collection of entry points.
    /// An entry point collection is an ordered collection of entry points,
    /// which allows iterating all members.
    /// @remarks This type is platform specific.
    ///          For details see the SMP Platform Mappings.
    typedef sequence<IEntryPoint> EntryPointCollection;

    // -------------------------------------------------------------------------
    // IEventSink
    // -------------------------------------------------------------------------

    /// Event sink that can be subscribed to event source (IEventSource).
    /// Provide notification method (event handler) that
    /// can be called by event providers when an event is emitted.
    interface IEventSink : IObject
    {
        /// Event notification.
        /// This event handler method is called when an event is emitted.
        /// @param sender Object sending the event.
        /// @param arg Event argument.
        void Notify(in IObject sender, in AnySimple arg);
    };
```

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

// --------------------------------------------------------------------------
// IEventSource
// --------------------------------------------------------------------------

/// Event source that event sinks (IEventSink) can subscribe to.
/// Allow event consumers to subscribe or unsubscribe to/from an event.
interface IEventSource : IObject
{
    /// Event sink is already subscribed.
    /// This exception is raised when trying to subscribe an event sink
    /// to an event source that is already subscribed.
    exception AlreadySubscribed
    {
        /// Event source the event sink is subscribed to.
        IEventSource eventSource;
        /// Event sink that is already subscribed.
        IEventSink eventSink;
    };

    /// Event sink is not subscribed.
    /// This exception is raised when trying to unsubscribe an event sink
    /// from an event source that is not subscribed to it.
    exception NotSubscribed
    {
        /// Event source the event sink is not subscribed to.
        IEventSource eventSource;
        /// Event sink that is not subscribed.
        IEventSink eventSink;
    };

    /// Event sink is not compatible with event source.
    /// This exception is raised when trying to subscribe an event sink
    /// to an event source that has a different event type.
    exception InvalidEventSink
    {
        /// Event source the event sink is subscribed to.
        IEventSource eventSource;
        /// Event sink that is not of valid type.
        IEventSink   eventSink;
    };

    /// Event subscription.
    /// Subscribe to the event source, i.e. request notifications.
    /// @param eventSink Event sink to subscribe to event source.
    /// @remarks An event sink can only be subscribed once to each
    ///          event source.
    ///          @par
    ///          Implementations may perform type checking on the
    ///          optional event argument of event source and event sink.
    ///          @par
    ///          Event sinks will be called synchronously in the order
    ///          they have been subscribed to the event source.
    void Subscribe(in IEventSink eventSink) raises
    (
        AlreadySubscribed,
        InvalidEventSink
    );

    /// Event unsubscription.
    /// Unsubscribe from the event source, i.e. cancel notifications.
    /// @param eventSink Event sink to unsubscribe from event source.
    /// @remarks An event sink can only be unsubscribed if it has been
    ///          subscribed before.
    void Unsubscribe(in IEventSink eventSink) raises
    (
        NotSubscribed
    );
};
```

```
    /// Collection of event sources.
    /// An event source collection is an ordered collection of event sources,
    /// which allows iterating all members.
    /// @remarks This type is platform specific.
    ///          For details see the SMP Platform Mappings.
    typedef sequence<IEventSource> EventSourceCollection;
};

// ---------------------------------------------------------------------------
// Components: IComponent, IModel, IService, ISimulator
// ---------------------------------------------------------------------------

module Smp
{
    // Forward declaration because of circular references.
    interface IComposite;
    interface ISimulator;
    module Services
    {
        interface ILogger;
    };

    // -----------------------------------------------------------------------
    // IComponent
    // -----------------------------------------------------------------------

    /// Base interface for all components.
    /// All SMP components implement this interface.
    /// @remarks Services (IService) and models (IModel) are typical components
    ///          in SMP, but the simulator (ISimulator) is a component as well.
    interface IComponent : IObject
    {
        /// Returns the parent component of the component ("property getter").
        /// Components link to their parent to allow traversing the
        /// tree of components upwards to the root component, which is
        /// typically the simulation environment (ISimulator).
        /// @return  Parent composite of component, or null if component
        ///          has no parent.
        /// @remarks Components that are part of a composition
        ///          point to their parent via this property.
        ///          The parent is null for root components.
        ///          Typically, only the simulator itself is a root component,
        ///          so all other components should have a parent component.
        IComposite GetParent();
    };

    /// Collection of components.
    /// A component collection is an ordered collection of components,
    /// which allows iterating all members.
    /// @remarks This type is platform specific.
    ///          For details see the SMP Platform Mappings.
    typedef sequence<IComponent> ComponentCollection;

    // -----------------------------------------------------------------------
    // IModel
    // -----------------------------------------------------------------------

    /// Publication receiver.
    /// Provide functionality to allow publishing fields and operations.
    /// @remarks This interface is platform specific.
    ///          For details see the SMP Platform Mappings.
    interface IPublication
    {
    };

    /// Enumeration of available model states.
    /// Each model is always in one of these four model states.
    /// @par
    /// Before going into <em>Initialising</em> state, the simulator has to
    /// ensure that every model is in <em>Connected</em> state.
    enum ModelStateKind
    {
        /// The <em>Created</em> state is the initial state of a model.
        /// Model creation is done by an external mechanism, e.g. by Factories.
        /// @par
        /// This state is entered automatically after the model
        /// has been created.
        /// @par
        /// To leave it, call the Publish() state transition method.
        MSK_Created,

        /// In <em>Publishing</em> state, the model is allowed to publish features.
        /// This includes publication of fields, operations and properties.
        /// In addition, the model is allowed to create other models.
        /// @par
        /// This state is entered using the Publish() state transition.
        /// @par
        /// To leave it, call the Configure() state transition method.
        MSK_Publishing,

        /// In <em>Configured</em> state, the model has been fully configured.
        /// This configuration may be done by external components, e.g. based
        /// on information stored in an SMDL Assembly, or internally by the
        /// model itself, e.g. by reading data from an external source.
        /// @par
        /// This state is entered using the Configure() state transition.
        /// @par
        /// To leave it, call the Connect() state transition method.
        MSK_Configured,

        /// In <em>Connected</em> state, the model is connected to the simulator.
        /// In this state, neither publication nor creation of other models
        /// is allowed anymore.
        /// @par
        /// This state is entered using the Connect() state transition.
        MSK_Connected
    };

    /// Model base interface.
    /// All SMP models implement this interface.
    /// @remarks This is the only mandatory interface models have to implement.
    ///          All other component or model functionality is optional.
    interface IModel : IComponent
    {
        /// Invalid model state.
        /// This exception is raised by a model when one of the
        /// state transition commands is called in an invalid state.
        exception InvalidModelState
        {
            /// State that is not valid.
            ModelStateKind invalidState;
            /// State that was expected.
            ModelStateKind expectedState;
        };

        /// Return model state.
        /// Return the state the model is currently in.
        /// @return  Current model state.
        /// @remarks The model state can be changed using the Publish(),
        ///          Configure() and Connect() state transition methods.
        ModelStateKind GetState();

        /// Request for publication.
        /// Request the model to publish its fields and operations
        /// against the provided IPublication interface.
        /// @par
        /// This method can only be called once for each model, and only
        /// when the model is in the <em>Created</em> state.
        /// When this operation is called, the model immediately enters
        /// the <em>Publishing</em> state, before it publishes any of its
        /// features.
        /// @param   receiver Publication receiver.
        /// @remarks The simulation environment typically calls this
        ///          method in the <em>Building</em> state.
        void Publish(in IPublication receiver) raises (InvalidModelState);

        /// Request for configuration.
        /// Request the model to perform any custom configuration.
        /// The model can create and configure other models using
        /// the field values of its published fields.
        /// @par
        /// This method can only be called once for each model, and only
        /// when the model is in <em>Publishing</em> state. The model can still
        /// publish further features in this call, and can even create other
        /// models, but at the end of this call, it needs to enter the
        /// <em>Configured</em> state.
        /// @param   logger Logger service for logging or error messages.
        /// @remarks The simulation environment typically calls this
        ///          method in the <em>Building</em> state.
        void Configure(in Services::ILogger logger) raises (InvalidModelState);

        /// Connect model to simulator.
        /// Allow the model to connect to the simulator (ISimulator).
        /// @par
        /// This method can only be called once for each model, and only
        /// when the model is in the <em>Configured</em> state.
        /// When this operation is called, the model immediately enters
        /// the <em>Connected</em> state, before it uses any of the simulator
        /// methods and services.
        /// @par
        /// In this method, the model may query for and use any of the
        /// available simulation services, as they are all guaranteed to be
        /// fully functional at that time. It may as well
        /// connect to other models' functionality (e.g. to event sources),
        /// as it is guaranteed that all models have been created and
        /// configured before the Connect() method of any model is called.
        /// @param   simulator Simulation Environment that hosts the model.
        /// @remarks The simulation environment typically calls this
        ///          method in the <em>Connecting</em> state.
        void Connect(in ISimulator simulator) raises (InvalidModelState);
    };

    /// Collection of models.
    /// A model collection is an ordered collection of models,
    /// which allows iterating all members.
    /// @remarks This type is platform specific.
    ///          For details see the SMP Platform Mappings.
    typedef sequence<IModel> ModelCollection;

    // -----------------------------------------------------------------------
    // IService
    // -----------------------------------------------------------------------

    /// Base interface for simulation services.
    /// All SMP services implement this interface.
    /// @remarks Currently, this interface does not add any functionality.
    interface IService : IComponent
    {
    };

    /// Collection of services.
    /// A service collection is an ordered collection of services,
    /// which allows iterating all members.
    /// @remarks This type is platform specific.
    ///          For details see the SMP Platform Mappings.
    typedef sequence<IService> ServiceCollection;
};

// ---------------------------------------------------------------------------
// Component Mechanisms: Composition (Containers)
// ---------------------------------------------------------------------------

module Smp
{
    // -----------------------------------------------------------------------
    // IContainer
    // -----------------------------------------------------------------------

    /// Container of components.
    /// A container allows to query for its contained components.
    /// @remarks Together with the IComposite interface, this interface
    ///          represents the composition mechanism (Container) of the SMP
    ///          Metamodel (SMDL).
    interface IContainer : IObject
    {
        /// Get all contained components.
        /// Query for the collection of all contained components.
        /// @return Collection of contained component.
        /// @remarks The collection may be empty if no components exist.
        ComponentCollection GetComponents();

        /// Get a contained component by name.
        /// Query for a contained component by its name.
        /// @param   name Child name.
        /// @return  Child component, or null if no component
        ///          with the given name is contained.
        /// @remarks The returned component may be null if no
        ///          component with the given name could be found.
        IComponent GetComponent(in String8 name);
    };

    /// Collection of containers.
    /// A container collection is an ordered collection of containers,
    /// which allows iterating all members.
    /// @remarks This type is platform specific.
    ///          For details see the SMP Platform Mappings.
    typedef sequence<IContainer> ContainerCollection;

    // ------------------------------------------------------------------------
    // IComposite
    // ------------------------------------------------------------------------

    /// Composite component.
    /// A component with contained components implements this interface.
    /// Child components are held in named containers.
    /// @remarks Together with the IContainer interface, this interface
    ///          represents the composition mechanism (Container) of the SMP
    ///          Metamodel (SMDL).
    ///          In UML 2.0, this maps to a composite required interface.
    interface IComposite : IComponent
    {
        /// Get all containers.
        /// Query for the collection of all containers of the component.
        /// @return Collection of containers.
        /// @remarks The collection may be empty if no containers exist.
        ContainerCollection GetContainers();

        /// Get a container by name.
        /// Query for a container of this component by its name.
        /// @param   name Container name.
        /// @return  Container queried for by name, or null if
        ///          no container with this name exists.
        /// @remarks The returned container may be null if no
        ///          container with the given name could be found.
        IContainer GetContainer(in String8 name);
    };
};

// ----------------------------------------------------------------------------
// Component Mechanisms: Aggregation (References)
// ----------------------------------------------------------------------------

module Smp
{
    // ------------------------------------------------------------------------
    // IReference
    // ------------------------------------------------------------------------

    /// Reference to components.
    /// A reference allows to query for the referenced components.
    /// @remarks Together with the IAggregate interface, this interface
    ///          represents the aggregation mechanism (Reference) of the SMP
    ///          Metamodel (SMDL).
    interface IReference : IObject
    {
        /// Get all referenced components.
        /// Query for the collection of all referenced components.
        /// @return Collection of referenced components.
        /// @remarks The collection may be empty if no components exist.
        ComponentCollection GetComponents();

        /// Get a referenced component by name.
        /// Query for a referenced component by its name.
        /// @param   name Component name.
        /// @return  Referenced component with the given name, or null if no
        ///          referenced component with the given name could be found.
        /// @remarks The returned component may be null if no
        ///          component with the given name could be found.
        ///          If more than one component with the given name exists,
        ///          it is undefined which one is returned with this method.
        IComponent GetComponent(in String8 name);
    };

    /// Collection of references.
    /// A reference collection is an ordered collection of references,
    /// which allows iterating all members.
    /// @remarks This type is platform specific.
    ///          For details see the SMP Platform Mappings.
    typedef sequence<IReference> ReferenceCollection;

    // ------------------------------------------------------------------------
    // IAggregate
    // ------------------------------------------------------------------------

    /// Aggregate component.
    /// A component with references to other components implements this interface.
    /// Referenced components are held in named references.
    /// @remarks Together with the IReference interface, this interface
    ///          represents the aggregation mechanism (Reference) of the SMP
    ///          Metamodel (SMDL).
    ///          In UML 2.0, this maps to an aggregate required interface.
    interface IAggregate : IComponent
    {
        /// Get all references.
        /// Query for the collection of all references of the component.
        /// @return  Collection of references.
        /// @remarks The collection may be empty if no references exist.
        ReferenceCollection GetReferences();

        /// Get a reference by name.
        /// Query for a reference of this component by its name.
        /// @param   name Reference name.
        /// @return  Reference with the given name, or null if no
        ///          reference with the given name could be found.
        /// @remarks The returned reference may be null if no
        ///          reference with the given name could be found.
        IReference GetReference(in String8 name);
    };
};

// ----------------------------------------------------------------------------
// Component Mechanisms: Self-Persistence
// ----------------------------------------------------------------------------

module Smp
{
    /// Storage reader.
    /// Provide functionality to read data from a storage.
    /// This interface is provided by a client (typically the simulation
    /// environment) to allow components implementing the IPersist interface
    /// to restore their state. It is passed to the Restore() method of every
    /// component implementing IPersist.
    /// @remarks This interface is platform specific.
    ///          For details see the SMP Platform Mappings.
    interface IStorageReader
    {
    };

    /// Storage writer.
    /// Provide functionality to write data to a storage.
    /// This interface is provided by a client (typically the simulation
    /// environment) to allow components implementing the IPersist interface
    /// to store their state. It is passed to the Store() method of every
    /// component implementing IPersist.
    /// @remarks This interface is platform specific.
    ///          For details see the SMP Platform Mappings.
    interface IStorageWriter
    {
    };

    /// Self persistence for components.
    /// A component may implement this interface if it wants to have control
    /// over loading and saving of its state.
    /// @remarks This is an optional interface. It needs to be implemented
    ///          for components with self-persistence only.
    interface IPersist : IComponent
    {
        /// Cannot restore from storage reader (IStorageReader).
        /// This exception is raised when the content of the storage reader
        /// passed to the Restore() method contains invalid data.
        /// @remarks This typically happens when a Store() has been created
        /// from a different configuration of components.
        exception CannotRestore
        {
            /// Error message indicating details of the problem.
            String8 message;
        };

        /// Cannot store to storage writer (IStorageWriter).
        /// This exception is raised when the component cannot store its data
        /// to the storage writer given to the Store() method.
        /// @remarks This may e.g. be if there is no disk space left.
        exception CannotStore
        {
            /// Error message indicating details of the problem.
            String8 message;
        };

        /// Restore component state from storage.
        /// @param    reader Interface that allows reading from storage.
        void Restore(in IStorageReader reader) raises
        (
            CannotRestore
        );

        /// Store component state to storage.
        /// @param    writer Interface that allows writing to storage.
        void Store(in IStorageWriter writer) raises
        (
            CannotStore
        );
    };
};

// ---------------------------------------------------------------------------
// Component Mechanisms: Dynamic Invocation
// ---------------------------------------------------------------------------

module Smp
{
    /// Request for dynamic invocation.
    /// The request holds information which is passed between a client
    /// invoking an operation via the IDynamicInvocation interface
    /// and a component (model or service) being invoked.
    interface IRequest
    {
        /// Invalid parameter index.
        /// This exception is raised when using an invalid parameter index to
        /// set (SetParameterValue()) or get (GetParameterValue()) a parameter
        /// value of an operation in a request.
        exception InvalidParameterIndex
        {
            /// Name of operation.
            String8 operationName;
            /// Invalid parameter index used.
            Int32 parameterIndex;
        };

        /// Invalid value for parameter.
        /// This exception is raised when trying to assign an illegal value to
        /// a parameter of an operation in a request using SetParameterValue().
        exception InvalidParameterValue
        {
            /// Name of parameter value was assigned to.
            String8 parameterName;
            /// Value that was passed as parameter.
            AnySimple value;
        };

        /// Invalid value for return value.
        /// This exception is raised when trying to assign an illegal return
        /// value of an operation in a request using SetReturnValue().
        exception InvalidReturnValue
        {
            /// Name of operation the return value was assigned to.
            String8 operationName;
            /// Value that was passed as return value.
            AnySimple value;
        };

        /// Operation is a void operation.
        /// This exception is raised when trying to read (GetReturnValue())
        /// or write (SetReturnValue()) the return value of a void operation.
        exception VoidOperation
        {
            /// Name of operation.
            String8 operationName;
        };

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

        /// Get parameter count.
        /// Returns the number of parameters stored in the request.
        /// @remarks Parameters are typically accessed by their 0-based index.
        ///          This index
        ///          - must not be negative,
        ///          - must be smaller than the parameter count.
        ///
        ///          Use the GetParameterIndex() method to access parameters by
        ///          name.
        Int32 GetParameterCount();

        /// Get index of a parameter.
        /// Query for a parameter index by parameter name.
        /// @param   parameterName Name of parameter.
        /// @return  Index of parameter with the given name (0-based), or
        ///          -1 if no parameter with the given name could be found.
        /// @remarks An index of -1 indicates that no parameter with the
        ///          given name exists.
        Int32 GetParameterIndex(in String8 parameterName);

        /// Set a parameter value.
        /// Assign a value to a parameter at a given position.
        /// @param   index Index of parameter (0-based).
        /// @param   value Value of parameter.
        void SetParameterValue(in Int32 index, in AnySimple value) raises
        (
            InvalidParameterIndex,
            InvalidParameterValue,
            InvalidAnyType
        );

        /// Get a parameter value.
        /// Query a value of a parameter at a given position.
        /// @param   index Index of parameter (0-based).
        /// @return  Value of parameter with given index.
        AnySimple GetParameterValue(in Int32 index) raises
```

```
        (
            InvalidParameterIndex
        );

        /// Set the return value.
        /// Assign the return value of the operation.
        /// @param   value Return value.
        void SetReturnValue(in AnySimple value) raises
        (
            InvalidReturnValue,
            InvalidAnyType,
            VoidOperation
        );

        /// Get the return value.
        /// Query the return value of the operation.
        /// @return  Return value of the operation.
        AnySimple GetReturnValue() raises
        (
            VoidOperation
        );
    };

    /// Dynamic invocation.
    /// A component may implement this interface in order to allow
    /// dynamic invocation of its operations.
    /// @remarks This is an optional interface. It needs to be implemented
    ///          only for components supporting dynamic invocation.
    ///          Dynamic invocation is typically used for scripting.
    ///          In its current implementation, dynamic invocation is limited
    ///          to operations using only simple types.
    interface IDynamicInvocation : IComponent
    {
        /// Invalid operation name.
        /// This exception is raised by the Invoke() method when trying to
        /// invoke a method that does not exist, or that does not support
        /// dynamic invocation.
        /// @remarks The name of the operation
        ///          can be extracted from the request using the method
        ///          GetOperationName().
        exception InvalidOperationName
        {
            /// Operation name of request passed to the Invoke() method.
            String8 operationName;
        };

        /// Invalid parameter count.
        /// This exception is raised by the Invoke() method when trying to
        /// invoke a method with a wrong number of parameters.
        /// @remarks The wrong number of parameters
        ///          can be extracted from the request using the method
        ///          GetParameterCount().
        exception InvalidParameterCount
        {
            /// Operation name of request passed to the Invoke() method.
            String8 operationName;
            /// Correct number of parameters of operation.
            Int32 operationParameters;
            /// Wrong number of parameters of operation.
            Int32 requestParameters;
        };

        /// Invalid parameter type.
        /// This exception is raised by the Invoke() method when trying to
        /// invoke a method passing a parameter of wrong type.
        /// @remarks The index of parameter of wrong type
        ///          can be extracted from the request using the method
        ///          GetParameterIndex().
        exception InvalidParameterType
        {
            /// Operation name of request passed to the Invoke() method.
            String8 operationName;
            /// Name of parameter of wrong type.
            String8 parameterName;
            /// Type that is not valid.
            SimpleTypeKind invalidType;
            /// Type that was expected.
            SimpleTypeKind expectedType;
        };

        /// Create request.
        /// Returns a request for the given operation
        /// that describes the parameters and the return value.
        /// @param   operationName Name of operation.
        /// @return  Request object for operation with given name,
        ///               or null if no operation with given name exists.
        /// @remarks The returned request may be null if no
        ///          operation with given name could be found, or if the
        ///          method of this name does not support dynamic invocation.
        ///          In its current implementation, dynamic invocation is
        ///          limited to operations using only simple types.
        IRequest CreateRequest(in String8 operationName);

        /// Dynamic invocation of operation.
        /// Dynamically invokes an operation using a request
        /// that has been created by CreateRequest() and filled
        /// with parameter values by the caller.
        /// @param   request Request object.
        /// @remarks On successful invocation, the return value of the
        ///          operation can be retrieved via GetReturnValue().
        void Invoke(in IRequest request) raises
        (
            InvalidOperationName,
            InvalidParameterCount,
            InvalidParameterType
        );

        /// Delete request.
        /// Destroys a request that has been created
        /// with the CreateRequest() method before.
        /// @param   request Request object to delete.
        /// @remarks The request must not be used anymore after
        ///          DeleteRequest() has been called for it.
        void DeleteRequest(in IRequest request);
    };
};

// -----------------------------------------------------------------------------
// Management Interfaces
// -----------------------------------------------------------------------------

module Smp
{
    /// SMP managed interfaces.
    /// 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 respective references.
    /// 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
    /// to read and write field values.
    /// @remarks Typically, these interfaces are called by a Loader or Model
    ///          Manager component to push configuration information into the
    ///          models, which has been read from an SMDL Assembly file.
    module Management
    {
        // ---------------------------------------------------------------------
        // Managed Components
        // ---------------------------------------------------------------------

        /// Managed object.
        /// A managed object additionally allows assigning name and description.
        interface IManagedObject : IObject
        {
            /// Defines the name of the managed object ("property setter").
            /// Management components may use this to assign names to objects.
            /// @param   name Name of object.
            /// @remarks Names
            ///          - must be unique within their context,
            ///          - must not be empty,
            ///          - must not exceed 32 characters in size,
            ///          - must start with a letter, and
            ///          - must only contain letters, digits,
            ///            the underscore ("_") and brackets ("[" and "]").
            void SetName(in String8 name) raises
            (
                InvalidObjectName
            );

            /// Defines the description of the managed object ("property setter").
            /// Management components may use this to set object descriptions.
            /// @param   description Description of object.
            void SetDescription(in String8 description);
        };

        /// Managed component.
        /// A managed component additionally allows assigning the parent.
        interface IManagedComponent : IManagedObject, IComponent
        {
            /// Defines the parent component ("property setter").
            /// Components link to their parent to allow traversing the
            /// tree of components in any direction.
            /// @param   parent Parent composite of component.
            /// @remarks Components that are part of a composition
            ///          point to their parent via this property.
            ///          The parent is null for root components.
            void SetParent(in IComposite parent);
        };

        /// Managed model.
        /// A managed model additionally allows getting and setting
        /// field values.
        interface IManagedModel : IModel, IManagedComponent
        {
            /// Invalid field name.
            /// This exception is raised when an invalid field name
            /// is specified.
            exception InvalidFieldName
            {
                /// Fully qualified field name that is invalid.
                String8 fieldName;
            };

            /// Invalid value for field.
            /// This exception is raised when trying to assign an
            /// illegal value to a field.
            exception InvalidFieldValue
            {
                /// Fully qualified field name the value was assigned to.
                String8 fieldName;
                /// Value that was passed as new field value.
                AnySimple invalidValue;
            };

            /// Invalid array size.
            /// This exception is raised when an invalid array size
            /// is specified.
            exception InvalidArraySize
            {
                /// Name of field that has been accessed.
                String8 fieldName;
                /// Invalid array size.
                Int64 givenSize;
                /// Real array size.
                Int64 arraySize;
            };

            /// Invalid value for array field.
            /// This exception is raised when trying to assign an
            /// illegal array value to an array field.
            exception InvalidArrayValue
            {
                /// Fully qualified field name the value was assigned to.
                String8 fieldName;
                /// Value array that was passed as new field value.
                AnySimpleArray invalidValues;
            };

            /// Get the value of a field which is typed by a system type.
            /// @param   fieldName Fully qualified field name.
            /// @return  Field value.
            /// @remarks This method can only be used to get values of simple
            ///          fields. For getting values of structured fields,
            ///          this method may be called multiply, e.g. specifying
            ///          "Position.x" to get the x component of a position.
            ///          For arrays of simple type, use GetArrayValue.
            AnySimple GetFieldValue(in String8 fieldName) raises
            (
                InvalidFieldName
            );

            /// Set the value of a field which is typed by a system type.
            /// @param   fieldName Fully qualified field name.
            /// @param   value Field value.
            /// @remarks This method can only be used to set values of simple
            ///          fields. For setting values of structured fields,
            ///          this method may be called multiply, e.g. specifying
            ///          "Position.x" to set the x component of a position.
            ///          For arrays of simple type, use SetArrayValue.
            void SetFieldValue(in String8 fieldName, in AnySimple value) raises
            (
                InvalidFieldName,
                InvalidFieldValue
            );

            /// Get the value of an array field which is typed by a system type.
            /// @param   fullName Fully qualified array field name.
            /// @param   values Array of field values.
            /// @param   length Length of array for safety checks.
            /// @remarks This method can only be used to get values of arrays of
            ///          simple type. These may be nested in a structure or array.
            void GetArrayValue(
                in String8 fullName,
                inout AnySimpleArray values,
                in Int64 length) raises
            (
                InvalidFieldName,
                InvalidArraySize
            );

            /// Set the value of an array field which is typed by a system type.
            /// @param   fullName Fully qualified array field name.
            /// @param   values Array of field values.
            /// @param   length Length of array for safety checks.
            /// @remarks This method can only be used to set values of arrays of
            ///          simple type. These may be nested in a structure or array.
            void SetArrayValue(
                in String8 fullName,
                in AnySimpleArray values,
                in Int64 length) raises
            (
                InvalidFieldName,
                InvalidArraySize,
                InvalidArrayValue
            );
        };

        // ---------------------------------------------------------------------
        // Managed Component Mechanisms
        // ---------------------------------------------------------------------

        /// Managed container.
        /// A managed container additionally allows
        /// querying the size limits and adding contained components.
        interface IManagedContainer : IContainer
        {
            /// Container is full.
            /// This exception is raised when trying to add a component to a
            /// container that is full, i.e. where the Count() has reached the
            /// Upper() limit.
            exception ContainerFull
            {
                /// Name of full container.
                String8 containerName;
                /// Number of components in the container, which is its Upper()
                /// limit when the container is full.
                Int64  containerSize;
            };

            /// Add component.
            /// Add a contained component to the container.
            /// @param   _component New contained component.
            /// @remarks This method raises a ContainerFull exception if the
            ///          container is full.
            void AddComponent(in IComponent _component) raises
            (
                ContainerFull,
                DuplicateName,
                InvalidObjectType
            );

            /// Get the number of contained components.
            /// Query for the number of components in the container.
            /// @return Current number of components.
            Int64 Count();

            /// Get lower bound for number of components.
            /// Query the minimum number of components in the container.
            /// @return Minimum number of components.
            Int64 Lower();

            /// Get upper bound for number of components.
            /// Query the maximum number of components in the container.
            /// @return  Maximum number of contained components (-1=unlimited).
            /// @remarks A return value of -1 indicates that the container has
            ///          no upper limit. This is consistent with the use of
            ///          upper bounds in UML, where a value of -1 represents
            ///          no limit (typically shown as "*").
            Int64 Upper();
        };

        /// Managed reference.
        /// A managed reference additionally allows
        /// querying the size limits and adding referenced components.
        interface IManagedReference : IReference
        {
            /// Reference is full.
            /// This exception is raised when trying to add a component to a
            /// reference that is full, i.e. where the Count() has reached the
            /// Upper() limit.
            exception ReferenceFull
            {
                /// Name of full reference.
                String8 referenceName;
                /// Number of components in the reference, which is its Upper()
                /// limit when the reference is full.
                Int64  referenceSize;
            };

            /// Component not Referenced.
            /// This exception is raised when trying to remove a component from
            /// a reference that does not reference this component.
            exception NotReferenced
            {
                /// Name of reference.
                String8 referenceName;
                /// Component that is not referenced.
                IComponent _component;
            };

            /// Add component.
            /// Add a referenced component.
            /// @param   _component New referenced component.
            /// @remarks This method raises a ReferenceFull exception if the
            ///          reference is full.
            void AddComponent(in IComponent _component) raises
            (
                ReferenceFull,
                InvalidObjectType
            );

            /// Get the number of referenced components.
            /// Query for the number of referenced components.
            /// @return Current number of referenced components.
            Int64 Count();

            /// Get lower bound for number of components.
            /// Query the minimum number of components in the collection of references.
            /// @return Minimum number of referenced components.
            Int64 Lower();

            /// Get upper bound for number of components.
            /// Query the maximum number of components in the collection
            /// of references.
            /// @return  Maximum number of referenced components (-1=unlimited).
            /// @remarks A return value of -1 indicates that the reference has
            ///          no upper limit. This is consistent with the use of
            ///          upper bounds in UML, where a value of -1 represents
            ///          no limit (typically shown as "*").
            Int64 Upper();

            /// Remove component.
            /// Remove a referenced component.
            /// @param   _component Referenced component to remove.
            /// @remarks This method raises a NotReferenced exception if the
            ///          given component is not referenced.
            void RemoveComponent(in IComponent _component) raises
            (
                NotReferenced
            );
        };

        // ---------------------------------------------------------------------
        // Managed Component Mechanisms: Events
        // ---------------------------------------------------------------------

        /// Event provider.
        /// Component that holds event sources, which allow
        /// other components to subscribe their event sinks.
        /// @remarks This is an optional interface.
        ///          It needs to be implemented for managed components only,
        ///          which want to allow access to event sources by name.
        interface IEventProvider : IComponent
        {
            /// Get all event sources.
            /// Query for the collection of all event sources of the component.
            /// @return  Collection of event sources.
            /// @remarks The collection may be empty if no event sources exist.
            EventSourceCollection GetEventSources();

            /// Get an event source by name.
            /// Query for an event source of this component by its name.
            /// @param   name Event source name.
            /// @return  Event source with the given name, or null if no
            ///          event source with the given name could be found.
            /// @remarks The returned event source may be null if no
            ///          event source with the given name could be found.
            IEventSource GetEventSource(in String8 name);
        };

        /// Event consumer.
        /// Component that holds event sinks, which may
        /// be subscribed to other component's event sources.
        /// @remarks This is an optional interface.
        ///          It needs to be implemented for managed components only,
        ///          which want to allow access to event sinks by name.
        interface IEventConsumer : IComponent
        {
            /// Get all event sinks.
            /// Query for the collection of all event sinks of the component.
```

```
            /// @return Collection of event sinks.
            /// @remarks The collection may be empty if no event sinks exist.
            EventSinkCollection GetEventSinks();

            /// Get an event sink by name.
            /// Query for an event sink of this component by its name.
            /// @param   name Event sink name.
            /// @return  Event sink with the given name, or null if no
            ///          event sink with the given name could be found.
            /// @remarks The returned event sink may be null if no
            ///          event sink with the given name could be found.
            IEventSink GetEventSink(in String8 name);
        };

        // ---------------------------------------------------------------------
        // Model Mechanisms: Entry Points
        // ---------------------------------------------------------------------

        /// Entry point publisher.
        /// An entry point publisher is a model that holds entry points,
        /// which may be registered,
        /// for example, with the Scheduler or the Event Manager services.
        /// @remarks This is an optional interface.
        ///          It needs to be implemented for managed models only,
        ///          which want to allow access to entry points by name.
        interface IEntryPointPublisher : IModel
        {
            /// Get all entry points.
            /// Query for the collection of all entry points of the model.
            /// @return Collection of entry points.
            /// @remarks The collection may be empty if no entry points exist.
            EntryPointCollection GetEntryPoints();

            /// Get an entry point by name.
            /// Query for an entry point of this model by its name.
            /// @param  name Entry point name.
            /// @return  Entry point with given name, or null if no
            ///          entry point with given name could be found.
            /// @remarks The returned entry point may be null if no
            ///          entry point with the given name could be found.
            IEntryPoint GetEntryPoint(in String8 name);
        };
    };
};

// -----------------------------------------------------------------------------
// Simulation Services
// -----------------------------------------------------------------------------

module Smp
{
    /// Simulation Services interfaces.
    /// This namespace defines the available simulation services of SMP.
    /// For each pre-defined service, a constant name is defined as well.
    module Services
    {
        // ---------------------------------------------------------------------
        // Predefined Service names
        // ---------------------------------------------------------------------

        const String8 SMP_Logger       = "Smp_Logger";       ///< Logger service.
        const String8 SMP_Scheduler    = "Smp_Scheduler";    ///< Scheduler service.
        const String8 SMP_TimeKeeper   = "Smp_TimeKeeper";   ///< Time Keeper service.
        const String8 SMP_EventManager = "Smp_EventManager"; ///< Event Manager service.
        const String8 SMP_Resolver     = "Smp_Resolver";     ///< Resolver service.

        // ---------------------------------------------------------------------
        // Logger Service
        // ---------------------------------------------------------------------

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

        // Identifiers of 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";

        /// This interface gives access to the Logger.
        /// All objects in a simulation can log messages using this service.
        /// @remarks This is a mandatory service the simulation environment
        ///          has to provide via its GetService() method.
        interface ILogger : IService
        {
            /// Return identifier of log message kind by name.
            /// This method can be used for pre-defined log message kinds,
            /// but is especially useful for user-defined log message kinds.
            /// @param   messageKindName Name of log message kind.
            /// @return  Identifier of log message kind.
            /// @remarks It is guaranteed that this method always returns
            ///          the same id for the same messageKindName string.
            LogMessageKind GetLogMessageKind(in String8 messageKindName);

            /// Mechanism to log a messages.
            /// This function logs a message to the simulation log file.
            /// @param   sender Object that sends the message.
            /// @param   message The message to log.
            /// @param   kind Kind of message.
            void Log(
                in IObject sender,
                in String8 message,
                in LogMessageKind kind);
        };

        // ---------------------------------------------------------------------
        // Time Keeper Service
        // ---------------------------------------------------------------------

        /// This interface gives access to the Time Keeper.
        /// Components can query for the time, and set epoch and mission time.
        /// @remarks This is a mandatory service the simulation environment
        ///          has to provide via its GetService() method.
        interface ITimeKeeper : IService
        {
            /// Return simulation time.
            /// Simulation time is a relative time that starts at 0.
            /// @return  Current simulation time.
            Duration GetSimulationTime();

            /// Return Epoch time.
            /// Epoch time is an absolute time with a fixed offset to
            /// simulation time.
            /// @return  Current epoch time.
            /// @remarks Epoch time typically progresses with simulation
            ///          time, but can be changed with SetEpochTime().
            DateTime GetEpochTime();

            /// Return Mission time.
            /// Mission time is a relative time with a fixed offset to epoch
            /// time.
            /// @return  Current mission time.
            /// @remarks Mission time typically progresses with epoch
            ///          time, but can be changed with SetMissionTime().
            Duration GetMissionTime();

            /// Return Zulu time.
            /// Zulu time is a system dependent time and not related to
            /// simulation time.
            /// @return  Current Zulu time.
            /// @remarks Zulu time is typically related to the system clock
            ///          of the computer.
```

```
        DateTime GetZuluTime();

        /// Set Epoch time.
        /// Changes the offset between simulation time and epoch time.
        /// @param   epochTime New epoch time.
        /// @remarks This shall raise an SMP_EpochTimeChanged event.
        ///          As mission time is relative to epoch time, this will
        ///          change the mission time as well.
        void SetEpochTime(in DateTime epochTime);

        /// Set Mission start.
        /// Changes the offset between mission time and epoch time.
        /// The mission time itself will be calculated as the offset
        /// between the current epoch time and the given mission start.
        /// @param   missionStart New mission start date and time.
        /// @remarks This shall raise an SMP_MissionTimeChanged event.
        void SetMissionStart(in DateTime missionStart);

        /// Set Mission time.
        /// Changes the offset between mission time and epoch time.
        /// @param   missionTime New mission time.
        /// @remarks This shall raise an SMP_MissionTimeChanged event.
        void SetMissionTime(in Duration missionTime);
    };

    // -------------------------------------------------------------------
    // Scheduler Service
    // -------------------------------------------------------------------

    /// Identifier of global event of scheduler or event manager service.
    typedef Int64 EventId;

    /// Invalid event identifier.
    /// This exception is raised when calling SetEventSimulationTime(),
    /// SetEventMissionTime(), SetEventEpochTime(), SetEventZuluTime(),
    /// SetEventCycleTime(), SetEventCount() or RemoveEvent() of the
    /// Scheduler service using an invalid scheduler event identifier.
    /// @par
    /// This exception is raised when calling Subscribe(), Unsubscribe()
    /// or Emit() of the Event Manager using an invalid global event.
    exception InvalidEventId
    {
        /// Identifier of invalid event.
        EventId event;
    };

    /// Enumeration of supported time kinds.
    enum TimeKind {
        TK_SimulationTime,  ///< Simulation time.
        TK_MissionTime,     ///< Mission time.
        TK_EpochTime,       ///< Epoch time.
        TK_ZuluTime         ///< Zulu time.
    };

    /// This interface gives access to the Scheduler.
    /// Components can register and unregister entry points for scheduling.
    /// @remarks This is a mandatory service the simulation environment
    ///          has to provide via its GetService() method.
    interface IScheduler : IService
    {
        /// Add immediate event.
        /// Add an immediate event to the scheduler.
        /// @param   entryPoint Entry point to call from event.
        void AddImmediateEvent(
            in IEntryPoint entryPoint);

        /// Add event relative to simulation time.
        /// Add event to scheduler that is called based on simulation time.
        /// @param   entryPoint Entry point to call from event.
        /// @param   simulationTime Duration from now when to trigger event.
        /// @param   cycleTime Duration between two triggers of the event.
        /// @param   count Number of times the event shall be repeated,
        ///                or 0 for a single event, or -1 for no limit.
        /// @return  Identifier that can be used to change or remove event.
        /// @remarks An event with <code>count=0</code> is not cyclic.
```

```
///          It will be removed automatically after it has been
///          triggered.
///          @par
///          An event with <code>count>0</code> is cyclic, and will
///          be repeated <code>count</code> times. Therefore, it
///          will be called <code>count+1</code> times, and then it
///          will be removed automatically.
///          @par
///          An event with <code>count=-1</code> is cyclic as well,
///          but it will be triggered forever, unless it is removed
///          from the scheduler using the RemoveEvent() method.
///          @par
///          For a cyclic event, the cycleTime needs to be positive.
///          For non-cyclic events, it is ignored.
///          @par
///          The simulationTime must not be negative. Otherwise, the
///          event will never be executed, but immediately removed.
EventId AddSimulationTimeEvent(
    in IEntryPoint entryPoint,
    in Duration simulationTime,
    in Duration cycleTime,
    in Int64 count);

/// Add event relative to mission time.
/// Add event to scheduler that is called based on mission time.
/// @param   entryPoint Entry point to call from event.
/// @param   missionTime Absolute mission time when to trigger event.
/// @param   cycleTime Duration between two triggers of the event.
/// @param   count Number of times the event shall be repeated,
///                or 0 for a single event, or -1 for no limit.
/// @return  Identifier that can be used to change or remove event.
/// @remarks An event with <code>count=0</code> is not cyclic.
///          It will be removed automatically after it has been
///          triggered.
///          @par
///          An event with <code>count>0</code> is cyclic, and will
///          be repeated <code>count</code> times. Therefore, it
///          will be called <code>count+1</code> times, and then it
///          will be removed automatically.
///          @par
///          An event with <code>count=-1</code> is cyclic as well,
///          but it will be triggered forever, unless it is removed
///          from the scheduler using the RemoveEvent() method.
///          @par
///          For a cyclic event, the cycleTime needs to be positive.
///          For non-cyclic events, it is ignored.
///          @par
///          The missionTime must not be before the current mission
///          time of the ITimeKeeper service. Otherwise, the
///          event will never be executed, but immediately removed.
EventId AddMissionTimeEvent(
    in IEntryPoint entryPoint,
    in Duration missionTime,
    in Duration cycleTime,
    in Int64 count);

/// Add event relative to epoch time.
/// Add event to scheduler that is called based on epoch time.
/// @param   entryPoint Entry point to call from event.
/// @param   epochTime Epoch time when to trigger event.
/// @param   cycleTime Duration between two triggers of the event.
/// @param   count Number of times the event shall be repeated,
///                or 0 for a single event, or -1 for no limit.
/// @return  Identifier that can be used to change or remove event.
/// @remarks An event with <code>count=0</code> is not cyclic.
///          It will be removed automatically after it has been
///          triggered.
///          @par
///          An event with <code>count>0</code> is cyclic, and will
///          be repeated <code>count</code> times. Therefore, it
///          will be called <code>count+1</code> times, and then it
///          will be removed automatically.
///          @par
///          An event with <code>count=-1</code> is cyclic as well,
///          but it will be triggered forever, unless it is removed
///          from the scheduler using the RemoveEvent() method.
///          @par
///          For a cyclic event, the cycleTime needs to be positive.
///          For non-cyclic events, it is ignored.
///          @par
///          The epochTime must not be before the current epoch
///          time of the ITimeKeeper service. Otherwise, the
///          event will never be executed, but immediately removed.
EventId AddEpochTimeEvent(
    in IEntryPoint entryPoint,
    in DateTime epochTime,
    in Duration cycleTime,
    in Int64 count);

/// Add event relative to Zulu time.
/// Add event to scheduler that is called based on Zulu time.
/// @param   entryPoint Entry point to call from event.
/// @param   zuluTime Absolute (Zulu) time when to trigger event.
/// @param   cycleTime Duration between two triggers of the event.
/// @param   count Number of times the event shall be repeated,
///                 or 0 for a single event, or -1 for no limit.
/// @return  Identifier that can be used to change or remove event.
/// @remarks An event with <code>count=0</code> is not cyclic.
///          It will be removed automatically after it has been
///          triggered.
///          @par
///          An event with <code>count>0</code> is cyclic, and will
///          be repeated <code>count</code> times. Therefore, it
///          will be called <code>count+1</code> times, and then it
///          will be removed automatically.
///          @par
///          An event with <code>count=-1</code> is cyclic as well,
///          but it will be triggered forever, unless it is removed
///          from the scheduler using the RemoveEvent() method.
///          @par
///          For a cyclic event, the cycleTime needs to be positive.
///          For non-cyclic events, it is ignored.
///          @par
///          The zuluTime must not be before the current Zulu
///          time of the ITimeKeeper service. Otherwise, the
///          event will never be executed, but immediately removed.
EventId AddZuluTimeEvent(
    in IEntryPoint entryPoint,
    in DateTime zuluTime,
    in Duration cycleTime,
    in Int64 count);

/// Set event execution time using simulation time.
/// Update when an existing event on the scheduler shall be called.
/// @param   event Identifier of event to modify.
/// @param   simulationTime Duration from now when to trigger event.
/// @remarks The event identifier must be valid, i.e. an identifier
///          of a scheduler event that has not yet been removed.
///          @par
///          The simulationTime must not be negative. Otherwise, the
///          event will never be executed, but immediately removed.
void SetEventSimulationTime(
    in EventId event,
    in Duration simulationTime) raises
(
    InvalidEventId
);

/// Set event execution time using mission time.
/// Update when an existing event on the scheduler shall be called.
/// @param   event Identifier of event to modify.
/// @param   missionTime Absolute mission time when to trigger event.
/// @remarks The event identifier must be valid, i.e. an identifier
///          of a scheduler event that has not yet been removed.
///          @par
///          The missionTime must not be before the current mission
///          time of the ITimeKeeper service. Otherwise, the
///          event will never be executed, but immediately removed.
void SetEventMissionTime(
    in EventId event,
    in Duration missionTime) raises
(
    InvalidEventId
);

/// Set event execution time using epoch time.
/// Update when an existing event on the scheduler shall be called.
/// @param   event Identifier of event to modify.
/// @param   epochTime Epoch time when to trigger event.
/// @remarks The event identifier must be valid, i.e. an identifier
///          of a scheduler event that has not yet been removed.
///          @par
///          The epochTime must not be before the current epoch
///          time of the ITimeKeeper service. Otherwise, the
///          event will never be executed, but immediately removed.
void SetEventEpochTime(
    in EventId event,
    in DateTime epochTime) raises
(
    InvalidEventId
);

/// Set event execution time using Zulu time.
/// Update when an existing event on the scheduler shall be called.
/// @param   event Identifier of event to modify.
/// @param   zuluTime Absolute (Zulu) time when to trigger event.
/// @remarks The event identifier must be valid, i.e. an identifier
///          of a scheduler event that has not yet been removed.
///          @par
///          The zuluTime must not be before the current Zulu
///          time of the ITimeKeeper service. Otherwise, the
///          event will never be executed, but immediately removed.
void SetEventZuluTime(
    in EventId event,
    in DateTime zuluTime) raises
(
    InvalidEventId
);

/// Set event cycle time.
/// Update cycle time of an existing event on the scheduler.
/// @param   event Identifier of event to modify.
/// @param   cycleTime Duration between two triggers of the event.
/// @remarks The event identifier must be valid, i.e. an identifier
///          of a scheduler event that has not yet been removed.
///          @par
///          For a cyclic event, the cycleTime needs to be positive.
///          For non-cyclic events, it is ignored.
void SetEventCycleTime(
    in EventId event,
    in Duration cycleTime) raises
(
    InvalidEventId
);

/// Set event count.
/// Update the count of an existing event on the scheduler.
/// @param   event Identifier of event to modify.
/// @param   count Number of times the event shall be repeated,
///                or 0 for a single event, or -1 for no limit.
/// @remarks The event identifier must be valid, i.e. an identifier
///          of a scheduler event that has not yet been removed.
///          @par
///          An event with <code>count=0</code> is not cyclic.
///          It will be removed automatically after it has been
///          triggered.
///          @par
///          An event with <code>count>0</code> is cyclic, and will
///          be repeated <code>count</code> times. Therefore, it
///          will be called <code>count+1</code> times, and then it
///          will be removed automatically.
///          @par
///          An event with <code>count=-1</code> is cyclic as well,
///          but it will be triggered forever, unless it is removed
///          from the scheduler using the RemoveEvent() method.
```

```
        ///          @par
        ///          For a cyclic event, the cycleTime needs to be positive.
        ///          For non-cyclic events, it is ignored.
        void SetEventCount(
            in EventId event,
            in Int64 count) raises
        (
            InvalidEventId
        );

        /// Remove an event from the scheduler.
        /// @param   event Event identifier of the event to remove.
        /// @remarks The event identifier must be valid, i.e. an identifier
        ///          of a scheduler event that has not yet been removed.
        ///          An event with <code>count>=0</code> is removed
        ///          automatically after it has been triggered
        ///          <code>count+1</code> times, and is no longer accessable.
        void RemoveEvent(in EventId event) raises
        (
            InvalidEventId
        );
    };

    /// Interface for Scheduler task.
    /// This interface extends IEntryPoint to allow scheduling tasks.
    /// A Task is an ordered collection of entry points.
    interface ITask : IEntryPoint
    {
        /// Get all entry points.
        /// Query for the collection of all entry points of the task.
        /// @return  Collection of entry points in the order they have been
        ///          added using the AddEntryPoint() method.
        /// @remarks The collection may be empty if no entry points exist.
        EntryPointCollection GetEntryPoints();

        /// Add entry point.
        /// Add an entry point to the task.
        /// @param   entryPoint Entry point to add to tasl.
        void AddEntryPoint(in IEntryPoint entryPoint);
    };

    // ---------------------------------------------------------------------
    // Event Manager Service
    // ---------------------------------------------------------------------

    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.

/// This interface gives access to the Event Manager.
/// Components can register entry points with events, can emit events,
/// and can define their own event typess.
/// @remarks This is a mandatory service the simulation environment
///          has to provide via its GetService() method.
interface IEventManager : IService
{
    /// Entry point is already subscribed.
    /// This exception is raised when trying to subscribe
    /// an entry point to an event that is already subscribed.
    exception AlreadySubscribed
    {
        /// Name of event the entry point is already subscribed to.
        String8 eventName;
        /// Entry point that is already subscribed.
        IEntryPoint entryPoint;
    };

    /// Entry point is not subscribed.
    /// This exception is raised when trying to unsubscribe
    /// an entry point from an event that is not subscribed to it.
    exception NotSubscribed
    {
        /// Name of event the entry point is not subscribed to.
        String8 eventName;
        /// Entry point that is not subscribed.
        IEntryPoint entryPoint;
    };

    /// Get event identifier.
    /// Get unique event identifier for an event name.
    /// @param   eventName Name of the global event.
    /// @return  Event identifier for global event with given name.
    /// @remarks It is guaranteed that this method will always return
    ///          the same value when called with the same event name.
    EventId GetEventId(in String8 eventName);

    /// Subscribe entry point.
    /// Subscribe an entry point to a global event.
    /// @param   event Event identifier of global event to subscribe to.
    /// @param   entryPoint Entry point to subscribe to global event.
    /// @remarks An entry point can only be subscribed once to an event.
    ///          If trying to subscribe it again, an exception of type
    ///          AlreadySubscribed is raised.
    void Subscribe(in EventId event, in IEntryPoint entryPoint) raises
    (
        InvalidEventId,
        AlreadySubscribed
    );

    /// Unsubscribe entry point.
    /// Unsubscribe an entry point from a global event.
    /// @param   event Event identifier of global event to unsubscribe from.
    /// @param   entryPoint Entry point to unsubscribe from global event.
    /// @remarks An entry point can only be unsubscribed from an event
    ///          after it has been subscribed before using Subscribe().
    ///          If trying to unsubscribe an entry point without subscribing
    ///          it before, a NotSubscribed exception is raised.
    void Unsubscribe(in EventId event, in IEntryPoint entryPoint) raises
    (
        InvalidEventId,
        NotSubscribed
    );

    /// Emit a global event.
    /// @param   event Event identifier of global event to emit.
    /// @remarks This will call all entry points that are subscribed to the
    ///          global event with the given identifier.
    ///          @par
    ///          Entry points will be called synchronously in the order
    ///          they have been subscribed to the global event.
```

```
            void Emit(in EventId event) raises
            (
                InvalidEventId
            );
        };

        // ---------------------------------------------------------------------
        // Resolver Service
        // ---------------------------------------------------------------------

        /// This interface gives access to the Resolver.
        /// Components can use the Resolver to resolve references to other
        /// components.
        /// @remarks This is an optional service the simulation environment
        ///          may provide via its GetService() method.
        interface IResolver : IService
        {
            /// Resolve reference to component via absolute path.
            /// @param   absolutePath Absolute path to component in simulation.
            /// @return  Component identified by path, or null for invalid path.
            IComponent ResolveAbsolute(in String8 absolutePath);

            /// Resolve reference to component via relative path.
            /// @param   relativePath Relative path to component in simulation.
            /// @param   sender Component that asks for resolving the reference.
            /// @return  Component identified by path, or null for invalid path.
            IComponent ResolveRelative(in String8 relativePath, in IComponent sender);
        };
    };
};

// -----------------------------------------------------------------------------
// Simulation Environment
// -----------------------------------------------------------------------------

module Smp
{
    // -------------------------------------------------------------------------
    // Simulator State Kind Enumeration
    // -------------------------------------------------------------------------

    /// Enumeration of available simulator states.
    /// The Simulator is always in one of these simulator states.
    enum SimulatorStateKind
    {
        /// In Building state, the model hierarchy is created. This
        /// is done by an external component, not by the simulator.
        /// @par
        /// This state is entered automatically after the simulation
        /// environment has performed its initialisation.
        /// @par
        /// To leave it, call the Connect() state transition method.
        SSK_Building,

        /// In Connecting state, the simulation environment traverses the
        /// model hierarchy and calls the Connect() method of each model.
        /// @par
        /// This state is entered using the Connect() state transition.
        /// @par
        /// After connecting all models to the simulator, an automatic
        /// state transition to the Initialising state is performed.
        SSK_Connecting,

        /// In Initialising state, the simulation environment executes all
        /// initialisation entry points in the order they have been added
        /// to the simulator using the AddInitEntryPoint() method.
        /// @par
        /// This state is either entered automatically after the simulation
        /// environment has connected all models to the simulator, or
        /// manually from Standby state using the Initialise() state transition.
        /// @par
        /// After initialising all models, an automatic state transition to
        /// the Standby state is performed.
        SSK_Initialising,
        /// In Standby state, the simulation environment (namely
        /// the Time Keeper Service) does not progress simulation time.
        /// Only entry points registered relative to Zulu time
        /// are executed.
        /// @par
        /// This state is entered automatically from the Initialising,
        /// Storing, and Restoring states, or manually from the Executing state
        /// using the Hold() state transition.
        /// @par
        /// To leave this state, call one of the Run(), Store(), Restore(), or
        /// Exit() state transitions.
        SSK_Standby,

        /// In Executing state, the simulation environment (namely
        /// the Time Keeper Service) does progress simulation time.
        /// Entry points registered with any of the available time kinds
        /// are executed.
        /// @par
        /// This state is entered using the Run() state transition.
        /// @par
        /// To leave this state, call the Hold() state transition.
        SSK_Executing,

        /// In Storing state, the simulation environment stores the values
        /// of all fields published with the State attribute to disk.
        /// Further, the Store() method of all components implementing the
        /// optional IPersist interface is called, to allow custom
        /// serialisation of additional information.
        /// While in this state, fields published with the State attribute
        /// must not be modified by the models, to ensure a consistent
        /// vector of field values is stored.
        /// @par
        /// This state is entered using the Store() state transition.
        /// @par
        /// After storing the simulator state, an automatic state transition
        /// to the Standby state is performed.
        SSK_Storing,

        /// In Restoring state, the simulation environment restores the values
        /// of all fields published with the State attribute from disk.
        /// Further, the Restore() method of all components implementing the
        /// optional IPersist interface is called, to allow custom
        /// deserialisation of additional information.
        /// While in this state, fields published with the State attribute
        /// must not be modified by the models, to ensure a consistent
        /// vector of field values is stored.
        /// @par
        /// This state is entered using the Restore() state transition.
        /// @par
        /// After restoring the simulator state, an automatic state transition
        /// to the Standby state is performed.
        SSK_Restoring,

        /// In Exiting state, the simulation environment is properly
        /// terminating a running simulation.
        /// @par
        /// This state is entered using the Exit() state transition.
        /// @par
        /// After exiting, the simulator is in an undefined state.
        SSK_Exiting,

        /// In this state, the simulation environment performs an
        /// abnormal simulation shut-down.
        /// @par
        /// This state is entered using the Abort() state transition.
        /// @par
        /// After aborting, the simulator is in an undefined state.
        SSK_Aborting
    };

    // ------------------------------------------------------------------------
    // ISimulator
    // ------------------------------------------------------------------------

    /// Interface to the Simulator.
```

```
/// The Simulator is a composite component that manages models and services.
/// As a service provider, it allows to add services, and to query for them.
/// As a model container, it allows to add root models, and to query for them.
/// In addition, the Simulator gives access to the simulator state.
/// @remarks Typically, this interface is implemented by
///          the simulation environment itself.
interface ISimulator : IComposite
{
    /// Get all models.
    /// Query for the collection of all root models.
    /// @return Collection of root model components.
    /// @remarks The collection may be empty if no models exist.
    ModelCollection GetModels();

    /// Get a root model by name.
    /// Query for a root model by its name.
    /// @param   name Model name.
    /// @return  Root model with the given name, or null if no
    ///          root model with the given name could be found.
    /// @remarks Users should always check the return value, as null
    ///          is returned when no root model with the given name
    ///          could be found.
    IModel GetModel(in String8 name);

    /// Add a root model.
    /// Add a new root model to the simulator.
    /// @param   model The root model.
    /// @remarks As root models are identified by their name, the new model
    ///          needs to have a unique name within the root models.
    ///          Otherwise, it will not be added, and an exception of type
    ///          DuplicateName will be raised.
    void AddModel(in IModel model) raises
    (
        DuplicateName
    );

    /// Get all services.
    /// Query for the collection of all services.
    /// @return  Collection of services.
    /// @remarks The collection may be empty if no services exist.
    ServiceCollection GetServices();

    /// Get a service by name.
    /// Query for a service by its name.
    /// @param   name Service name.
    /// @return  Service with the given name, or null if no
    ///          service with the given name could be found.
    /// @remarks Users should always check the return value, as null
    ///          is returned when no service with the given name
    ///          could be found.
    IService GetService(in String8 name);

    /// Add a user-defined service.
    /// Add a new service to the simulator.
    /// @param   service The user-defined service.
    /// @remarks As services are identified by their name, the new service
    ///          needs to have a unique name within the services.
    ///          Otherwise, it will not be added, and an exception of type
    ///          DuplicateName will be raised.
    void AddService(in IService service) raises
    (
        DuplicateName
    );

    /// Get logger service.
    /// Query for mandatory logger service.
    /// @return  Mandatory logger service.
    /// @remarks This is a type-safe convenience method to query for the
    ///          logger service, to avoid having to use the general
    ///          GetService() method. For the mandatory services, it is
    ///          recommended to use the convenience methods.
    Services::ILogger GetLogger();

    /// Get scheduler service.
    /// Query for mandatory scheduler service.
```

```
        /// @return  Mandatory scheduler service.
        /// @remarks This is a type-safe convenience method to query for the
        ///          scheduler service, to avoid having to use the general
        ///          GetService() method. For the mandatory services, it is
        ///          recommended to use the convenience methods.
        Services::IScheduler GetScheduler();

        /// Get time keeper service.
        /// Query for mandatory time keeper service.
        /// @return  Mandatory time keeper service.
        /// @remarks This is a type-safe convenience method to query for the
        ///          time keeper service, to avoid having to use the general
        ///          GetService() method. For the mandatory services, it is
        ///          recommended to use the convenience methods.
        Services::ITimeKeeper GetTimeKeeper();

        /// Get event manager service.
        /// Query for mandatory event manager service.
        /// @return  Mandatory event manager service.
        /// @remarks This is a type-safe convenience method to query for the
        ///          event manager service, to avoid having to use the general
        ///          GetService() method. For the mandatory services, it is
        ///          recommended to use the convenience methods.
        Services::IEventManager GetEventManager();

        /// Get simulator state.
        /// Returns the current simulator state.
        /// @return  Current simulator state.
        SimulatorStateKind GetState();

        /// Call Publish() method of models.
        /// This method asks the simulation environment to call the Publish()
        /// method of all model instance in the hierarchy which are still in
        /// <em>Created</em> state.
        /// @remarks This method is typically called by an external component
        ///          after creating new model instances, typically using the
        ///          information in an SMDL Assembly.
        ///          This method can only be called in Building state.
        void Publish();

        /// Call Configure() method of models.
        /// This method asks the simulation environment to call the Configure()
        /// method of all model instance in the hierarchy which are still in
        /// <em>Publishing</em> state.
        /// @remarks This method is typically called by an external component
        ///          after setting field values of new model instances,
        ///          typically using the information in an SMDL Assembly.
        ///          This method can only be called in Building state.
        void Configure();

        /// Enter Connecting state.
        /// This method informs the simulation environment that the hierarchy
        /// of model instances has been built, and can now be connected
        /// to the simulator. After connecting all models, an automatic state
        /// transition to the Initialising state is performed.
        /// @remarks This method is typically called by an external component
        ///          after creating and configuring all model instances.
        ///          This method can only be called in Building state.
        void Connect();

        /// Enter initialising state.
        /// This method changes from Standby state to Initialising state,
        /// where each initialisation entry point will be executed again.
        /// @remarks The entry points will be called in the order they
        ///          have been added using the AddInitEntryPoint() method.
        ///          This method can only be called in Standby state.
        void Initialise();

        /// Enter standby state.
        /// This method changes from executing to standby state.
        /// @remarks This method can only be called in Executing state.
        void Hold();

        /// Enter executing state.
        /// This method changes from standby to executing state.
        /// @remarks This method can only be called in Standby state.
        void Run();

        /// Enter storing state.
        /// This method is used to store a state vector to file.
        /// @param   filename Name to use for simulation state vector file.
        /// @remarks This method can only be called in Standby state.
        void Store(in String8 filename);

        /// Enter restoring state.
        /// This method is used to restore a state vector from file.
        /// @param   filename Name of simulation state vector file.
        /// @remarks This method can only be called in Standby state.
        void Restore(in String8 filename);

        /// Enter exiting state.
        /// This method is used for a normal termination of a simulation.
        /// @remarks This method can only be called in Standby state.
        void Exit();

        /// Enter aborting state.
        /// This method is used for an abnormal termination of a simulation.
        /// @remarks This method can be called from any other state.
        void Abort();

        /// Add initialisation entry point.
        /// This method can be used to add entry points that shall be executed
        /// in the Initialising state.
        /// These entry points will be called in the order they have been
        /// added to the simulator using this method.
        /// @par
        /// The ITask interface (which is derived from IEntryPoint) can be
        /// used to add several entry points in a well-defined order.
        void AddInitEntryPoint(in IEntryPoint entryPoint);
    };

    // -----------------------------------------------------------------------
    // Predefined Simulator Containers
    // -----------------------------------------------------------------------

    const String8 SMP_SimulatorModels   = "Models";
    const String8 SMP_SimulatorServices = "Services";

    // -----------------------------------------------------------------------
    // IFactory
    // -----------------------------------------------------------------------

    /// This interface is implemented by all component factories.
    interface IFactory : IObject
    {
        /// Get specification identifier of factory.
        /// @return  Identifier of component specification.
        Uuid GetSpecification();

        /// Get implementation identifier of factory.
        /// @return  Identifier of component implementation.
        Uuid GetImplementation();

        /// Create a new instance.
        /// This method creates a new component instance.
        IComponent CreateInstance();

        /// Delete an existing instance.
        /// This method deletes an existing component instance that has been
        /// created using CreateInstance() earlier.
        /// @param instance Instance to delete.
        void DeleteInstance(in IComponent instance);
    };

    /// Collection of factories.
    /// A factory collection is an ordered collection of factories,
    /// which allows iterating all members.
    /// @remarks This type is platform specific.
    ///          For details see the SMP Platform Mappings.
    typedef sequence<IFactory> FactoryCollection;
    // ------------------------------------------------------------------------
    // IDynamicSimulator
    // ------------------------------------------------------------------------

    /// This interface gives access to a dynamic simulator.
    /// External applications can register component factories with a dynamic
    /// simulator, and can use its CreateInstance() method to create component
    /// instances.
    interface IDynamicSimulator : ISimulator
    {
        /// Duplicate Uuid.
        /// This exception is raised when trying to register a factory with
        /// an implementation Uuid of an existing factory.
        /// This would lead to duplicate implementation Uuids.
        exception DuplicateUuid
        {
            /// Name of factory that tried to register under this Uuid.
            String8 newName;
            /// Name of factory already registered under this Uuid.
            String8 oldName;
        };

        /// Register a global component factory with the simulator.
        /// @param   componentFactory Component factory that creates and
        ///          deletes the component instances.
        void RegisterFactory(in IFactory componentFactory) raises
        (
            DuplicateUuid
        );

        /// Create an instance of the given component implementation.
        /// @param   implUuid Identifier of component Implementation.
        /// @return  New component instance of given component implementation,
        ///          or null if no factory is available.
        IComponent CreateInstance(in Uuid implUuid);

        /// Get the factory of the given component implementation.
        /// @param   implUuid Identifier of component Implementation.
        /// @return  Component factory of given component implementation,
        ///          or null if no factory is available.
        IFactory GetFactory(in Uuid implUuid);

        /// Get all factories of the given component specification.
        /// @param   specUuid Identifier of component Specification.
        /// @return  Collection of factories of given component specification.
        /// @remarks The collection may be empty if no factories have been
        ///          registered for the given specification identifier.
        FactoryCollection GetFactories(in Uuid specUuid);
    };
};
```