ECSS-E-ST-40-07C
2 March 2020

Space engineering

Simulation modelling platform

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

Foreword

This Standard is one of the series of ECSS Standards intended to be applied together for the management, engineering and product assurance in space projects and applications. ECSS is a cooperative effort of the European Space Agency, national space agencies and European industry associations for the purpose of developing and maintaining common standards. Requirements in this Standard are defined in terms of what shall be accomplished, rather than in terms of how to organize and perform the necessary work. This allows existing organizational structures and methods to be applied where they are effective, and for the structures and methods to evolve as necessary without rewriting the standards.

This Standard has been prepared by the ECSS-E-ST-40-07C Working Group, reviewed by the ECSS Executive Secretariat and approved by the ECSS Technical Authority.

Disclaimer

Published by:	ESA Requirements and Standards Division ESTEC, P.O. Box 299, 2200 AG Noordwijk The Netherlands
Copyright:	2020© by the European Space Agency for the members of ECSS

Change log

ECSS-E-ST-40-07C 2 March 2020	First issue

Table of contents

Figures

Tables

Introduction

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

This standard integrates the ECSS-E-ST-40 with additional requirements which are specific to the development of simulation software. The formulation of this standard takes into account:

- The existing ISO 9000 family of documents, and
- The Simulation Model Portability specification version 1.2.

The intended readership of this standard is the simulator software customer and supplier.

1 Scope

ECSS-E-ST-40-07 is a standard based on ECSS-E-ST-40 for the engineering of simulation software.

ECSS-E-ST-40-07 complements ECSS-E-ST-40 in being more specific to simulation software. Simulation software include both Simulation environments and simulation models. The standard enables the effective reuse of simulation models within and between space projects and their stakeholders. In particular, the standard supports model reuse across different simulation environments and exchange between different organizations and missions.

This standard can be used as an additional standard to ECSS-E-ST-40 providing the additional requirements which are specific to simulation software.

This standard may be tailored for the specific characteristic and constrains of a space project in conformance with ECSS-S-ST-00.

Applicability

This standard lays down requirements for simulation software including both Simulation environments and simulation models. The requirements cover simulation models' interfaces and simulation environment interfaces for the purpose of model re-use and exchange to allow simulation models to be run in any conformant simulation environment.

A consequence of being compliant to this standard for a model is the *possibility* of being reused in several simulation facilities or even in several projects. However, adherence to this standard does not imply or guarantees model reusability, it is only a precondition. Other characteristics of the model, to be defined outside this standard, such as its functional interfaces and behaviour, its configuration data as well as quality, suitability and performance, etc. are also heavily affecting the potential for a model to be reused. In addition, agreements need to be reached on simulation environments compatibility, model validation status as well as legal issues and export control restrictions.

Therefore, this standard *enables* but does not mandate, impose nor guarantee successful model re-use and exchange.

Model reuse in this standard is meant both at source-code and binary level, with the latter restricted to a fixed platform.

2 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this ECSS Standard. For dated references, subsequent amendments to, or revision of any of these publications do not apply. However, parties to agreements based on this ECSS Standard are encouraged to investigate the possibility of applying the more recent editions of the normative documents indicated below. For undated references, the latest edition of the publication referred to applies.

ECSS-S-ST-00-01	ECSS system - Glossary of terms
ECSS-E-ST-40	Space engineering - Software general requirements
[SMP_FILES]	ECSS_SMP_Issue1(2March2020).zip – SMP C++ Header files, SMP XML schemas and SMP Catalogue. (Available from ECSS website)
https://www.w3.org/TR/xmlschema11-2/	XML schema specification
http://www.opengroup.org	The UUID specification from Open Group.
https://www.osgi.org/developer/specifications/	OSGi Specifications

3 Terms, definitions and abbreviated terms

3.1 Terms from other standards

a. For the purpose of this Standard, the terms and definitions from ECSS-S-ST-00-01 and ECSS-E-ST-40 apply.

b. For the purpose of this Standard, the terms and definitions from ECSS-E-ST-70 apply, in particular the following term:

   1. mission

3.2 Terms specific to the present standard

In the following list of terms, underlined words are further defined in the same list.

3.2.1 aggregate

relationship between two <u>components</u> implemented by storing their <u>references</u>

NOTE Each <u>component</u> in such a relationship keeps its own lifecycle and it does not dependent on that of other components.

3.2.2 association

relationship between two <u>instances of any data-type,</u> where each <u>instance</u> has its own lifecycle and there is no owner

3.2.3 breakpoint

unambiguous state of a simulation

3.2.4 component

building block of a simulation that can be instantiated and that has a well-defined <u>contract</u> to its environment

3.2.5 composite

<u>component</u> implementing <u>composition</u>

3.2.6 composition

hierarchical relationship where child <u>component</u> is destroyed if the parent <u>component</u> is destroyed

3.2.7 configuration

specification of values for fields of components

3.2.8 constructor

specific operation of a component, bearing the same name of the component, whose purpose is to allocate and build an instance of said component

3.2.9 consumer

component that can receive data in one of its input fields from an output field of another component

3.2.10 container

typed collection of child components

3.2.11 contract

set of interfaces, operations, fields, entry points, event sinks, event sources and all the associated constraints, used to interact with a component

3.2.12 data transfer

copy of value from an output field to an input field

3.2.13 entry point

operation without parameters that does not return a value, which can be added to the scheduler or event manager service

3.2.14 epoch time

absolute time of the simulation

3.2.15 event

see "simulation event"

3.2.16 event manager

component that implements the IEventManager interface

NOTE The IEventManager interface is specified in clause 5.3.4.

3.2.17 event sink

receiver of specific notifications, owned by a component and subscribed via a subscription mechanism

3.2.18 event source

emitter of specific notifications, owned by a component and offering a subscription mechanism

3.2.19 exception

non-recoverable error that can occur when calling into an operation or property

3.2.20 field

feature characterised by a value type and holding a value

3.2.21 input field

field explicitly marked for receiving values as a result of a data transfer

3.2.22 interface

named set of properties and operations

3.2.23 logger

component that implements the ILogger interface

NOTE The ILogger interface is specified in clause 5.3.1.

3.2.24 mission time

relative time measuring elapsed time from a mission specific point in time

3.2.25 model

component that implements the IModel interface

NOTE The IModel interface is specified in clause 5.2.3.2.

3.2.26 model implementation

executable code implementing a model

3.2.27 model instance

occurrence of a model implementation

3.2.28 output field

field explicitly marked for being the source of a value in a data transfer

3.2.29 operation

declaration of a behavioural feature of a component or an interface with the option to define parameters, return value and raised exceptions

3.2.30 package

collection of types, where each one is either a value type or a component

3.2.31 platform

set of subsystems/technologies that provide a coherent set of functionality through APIs and specified usage patterns

3.2.32 primitive type

type that can no longer be de-composed and that is pre-defined by the standard

NOTE The available primitive types are listed in Table 5-1: Primitive Types.

3.2.33 property

typed feature of a class, an interface or a component that can be accessed by two operations, the setter and the getter, not necessarily both present

3.2.34 provider

component that can send data of one of its output fields to an input field of another component

3.2.35 reference

pointer to a component

NOTE When dealing with the C++ mapping, the term reference has a meaning specific to that language, whereas in the rest of this standard it means point to a component (but it cannot for instance be a pointer to a class).

3.2.36 resolver

component that implements the IResolver interface

NOTE The IResolver interface is specified in clause 5.3.5.

3.2.37 schedule

planned time ordered execution of entry points

3.2.38 scheduler

component that implements the IScheduler interface

NOTE The IScheduler interface is specified in clause 5.3.3.

3.2.39 service

component that implements the IService interface

NOTE The IService interface is specified in clause 5.2.3.3.

3.2.40 simple field

field of a type that maps directly to a primitive type

3.2.41 simulation environment

platform implementing the standard E-40-07 services (event manager, link registry, logger, resolver, scheduler and time keeper) and the ISimulator interface

3.2.42 simulation event

call to an entry point by either scheduler or event manager

NOTE The term "event" is synonymous.

3.2.43 simulation time

relative time since start of simulation

3.2.44 simulator

collection of services and hierarchy of model instances together with a simulation environment

3.2.45 simulation

single execution of a simulator

3.2.46 simulation service

service instance resolvable by name in the global scope of the simulation environment

3.2.47 source

component that owns one or more references, one or more event links, or one or more output fields

NOTE The term "source component" is synonymous.

3.2.48 source component

See source

3.2.49 target

component that implements one or more interfaces, provides one or more event sinks, or one or more input fields

NOTE The term "target component" is synonymous.

3.2.50 target component

see "target"

3.2.51 time keeper

component that implements the ITimeKeeper interface

NOTE The ITimeKeeper interface is specified in clause 5.3.2.

3.2.52 value

state of a value type

3.2.53 value type

set of values which a variable can possess

3.2.54 Zulu time

the computer clock time, also called wall clock time

3.3 Abbreviated terms

For the purpose of this Standard, the abbreviated terms and symbols from ECSS-S-ST-00-01 and the following apply:

Abbreviation	**Meaning**
DES	Discrete-Event Simulation
SMDL	Simulation Model Definition Language
SMP	Simulation Modelling Platform
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UUID	Universally Unique IDentifier

3.4 Nomenclature

The following nomenclature applies throughout this document:

a. The word “shall” is used in this Standard to express requirements. All the requirements are expressed with the word “shall”.

b. The word “should” is used in this Standard to express recommendations. All the recommendations are expressed with the word “should”.

NOTE It is expected that, during tailoring, recommendations in this document are either converted into requirements or tailored out.

c. The words “may” and “need not” are used in this Standard to express positive and negative permissions, respectively. All the positive permissions are expressed with the word “may”. All the negative permissions are expressed with the words “need not”.

d. The word “can” is used in this Standard to express capabilities or possibilities, and therefore, if not accompanied by one of the previous words, it implies descriptive text.

NOTE In ECSS “may” and “can” have completely different meanings: “may” is normative (permission), and “can” is descriptive.

e. The present and past tenses are used in this Standard to express statements of fact, and therefore they imply descriptive text.

4 Principles

4.1 Objectives

The main objective of this standard is to enable the effective reuse of simulation models and applications within and between space projects and their stakeholders. In particular, the standard supports model reuse across different simulation environments and exchange between different organizations and missions.

The portability of models between different simulation environments is supported by defining a standard interface between the simulation environment and the models. Models can therefore be plugged into a different simulation environment without requiring any modification to the model source code.

The portability of models between different operating systems and hardware takes into consideration dependencies such as avoiding calls to operating specific APIs or use of hardware specific features. The guidelines to the model developer, on how to avoid developing models with such dependencies, is outside the scope of this standard.

4.2 Common Concepts and common types

The main purpose of SMP is to promote platform independence, interoperability and reuse of simulation models. This is done by defining;

- Common Concepts: All SMP models fulfil common high-level concepts addressing fundamental modelling issues. This enables the development of models on an abstract level, which is essential for independence from simulation environments and reuse of models;
- Common Type System: All SMP models are built upon a common type system. This enables different models to have a common understanding of the syntax and semantics of basic types, which is essential for interoperability between different models.

In other words, models are using common concepts and a common type system to become interoperable. Thus, models ‘live’ in between these two common layers as shown in Figure 4-1.

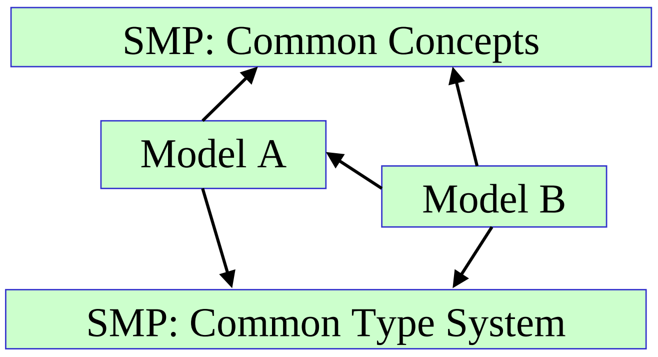

Figure 4-1: Common Concepts and Type System

4.3 Architecture

The SMP architecture covers two types of components;

- Simulation Models provide application specific behaviour;
- Simulation Environments provide Simulation Services.

This architecture is depicted in Figure 4-2.

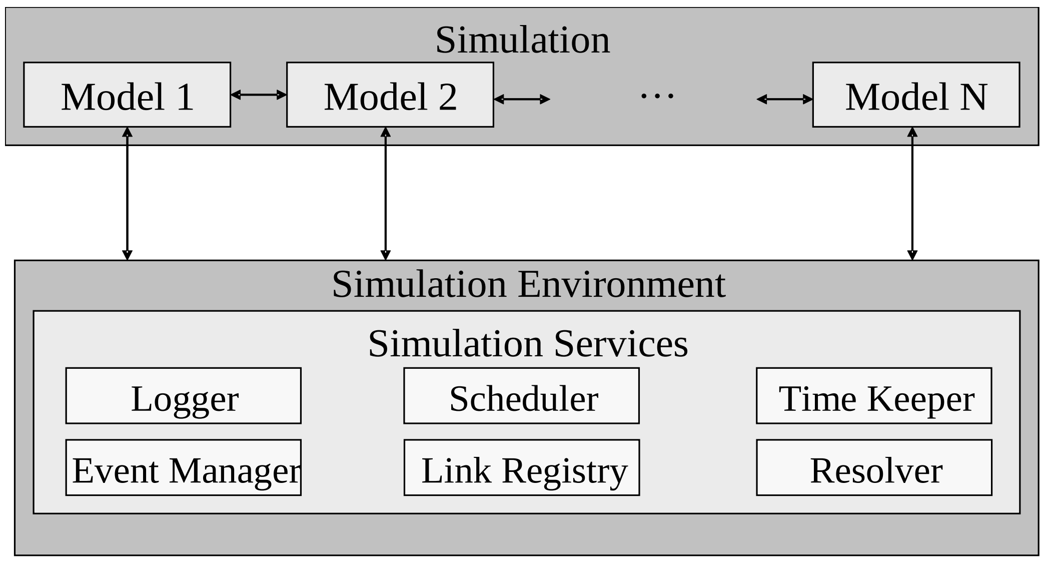

Figure 4-2: SMP Architecture

An SMP compliant simulation environment provides the following six simulation services:

- Logger: Allows logging messages (see clause 5.3.1);
- Time Keeper: Provides the four different SMP time kinds (see clause 4.4 and 5.3.2);
- Scheduler: Allows calls of entry points based on timed or cyclic events (see clause 5.3.3);
- Event Manager: Provides mechanisms for global asynchronous events (see clause 5.3.4);

- Resolver: Provides the ability to get a reference to any model within a simulation (see clause 5.3.5);
- Link Registry: Maintains a list of the links between model instances (see clause 5.3.6).

In addition, it supports other concepts laid out in this standard via some dedicated interfaces:

- Simulation state machine controlling interface (see clause 4.5 and 5.3.7);
- Interfaces allowing Self-persistence as described in clause 4.8 (see also IStorageReader in clause 5.3.8.1 and IStorageWriter in clause 5.3.8.2);
- Publication: A set of interfaces allowing models to publish their state to the simulation environment (see clause 5.3.9);
- Type registry: A registry allowing components to register types that later can be used for publication (see clause 5.3.10);
- Component Factory: Ability to create components via a factory (see clause 5.3.11).

The arrows in Figure 4-2 indicate interaction between components. In SMP, communication is performed via interfaces. Two different types of interfaces can be identified in this architecture:

- Interfaces between components and the Simulation Environment, and
- Inter-component communication interfaces.

4.4 Time handling principle

SMP defines four different time scales, referred to as time kinds (see clause 5.1.2 for exact specification):

- Simulation Time: Relative time since start of simulation, starting at 0 when the simulation is setup.
- Zulu Time: Zulu Time is the computer clock time, also called wall clock time.
- Epoch Time: The absolute time of the simulation.
- Mission Time: Mission time is a relative time, i.e. it measures elapsed time from a mission specific point in time.

SMP defines both Epoch and Mission Time as a fixed offset from Simulation Time. The Offset is set via calls to the SMP time keeper service and the two time kinds progress linearly with Simulation time. SMP does not define how Simulation time progress with respect to Zulu time. Typical examples of such a correlation is:

- **Real-Time:** The simulation time progresses with real-time, where real-time is typically defined by the computer clock.
- **Accelerated**: The simulation time progresses relative to real-time using a constant acceleration factor. This factor can 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.

- **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).

SMP does not mandate which of these modes a simulation environment supports.

4.5 Simulation lifecycle

Any SMP simulation goes via a lifecycle as defined in Figure 4-3. The simulation environment is responsible to ensure that this state diagram is followed. It is controlled via the ISimulator interface (see clause 5.3.7).

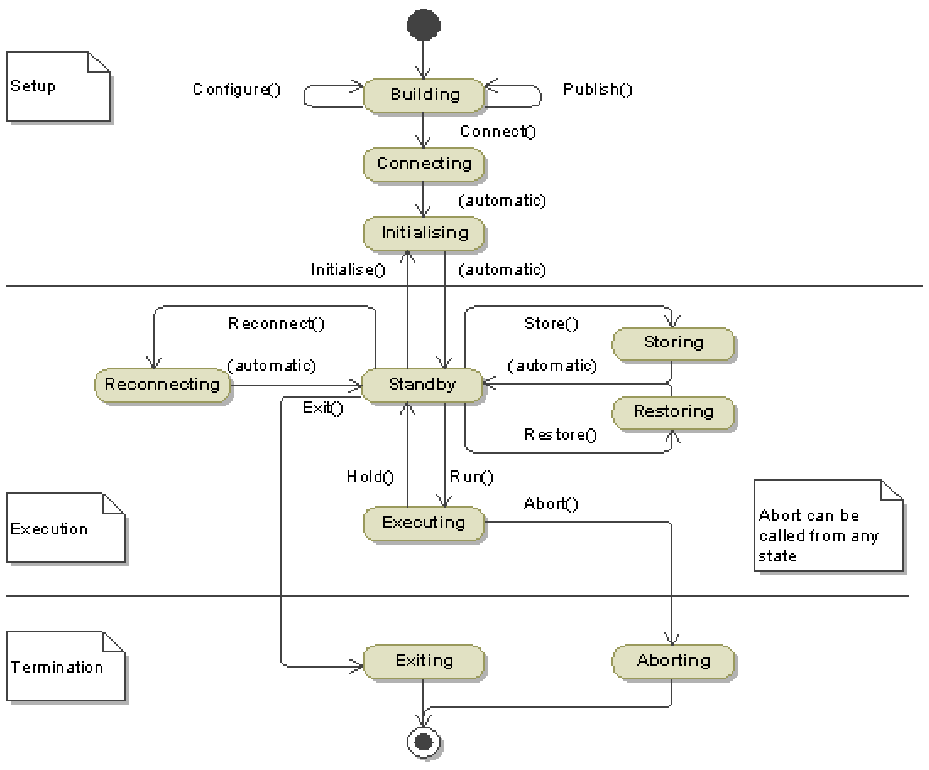

Figure 4-3: SMP State machine

Each state in Figure 4-3 has its own purpose and behaviour as explained in Table 4-1. Notice that some state transitions are automatically performed by the Simulation Environment as indicated in Figure 4-3, while others need explicit calls to the ISimulator interface.

Table 4-1: Overview of simulation states

Name	Description
Building	In Building state, the model hierarchy is created. In this state, Publish() and Configure() can be called any time to call the corresponding Publish() and Configure() operations of each component.
Connecting	In Connecting state, the simulation environment traverses the model hierarchy and calls the Connect() method of each component.
Initialising	In Initialising state, the simulation environment executes all initialization entry points in the order they have been added to the simulator using the ISimulator AddInitEntryPoint() method (see 5.3.7).
Standby	The simulation environment does not progress simulation time. Only entry points registered relative to Zulu time are executed.
Executing	The simulation environment does progress simulation time. Entry points registered with any of the available time kinds are executed.
Storing	In Storing state, the simulation environment first stores the values of all fields published with the State attribute to storage (typically a file). Afterwards, the Store() method of all components (Models and Services) implementing the optional IPersist interface is called, to allow custom storing of additional information.
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.
Reconnecting	In Reconnecting state, the simulation environment makes sure that models that have been added to the simulator after leaving the Building state are properly published, configured and connected.
Exiting	In Exiting state, the simulation environment is properly terminating a running simulation.
Aborting	In this state, the simulation environment attempts a simulation shut-down, whereby the simulation can stop executing as the users expect, without guaranties for actual release of resources.

4.6 Simulation method

4.6.1 Discrete-event simulation (DES)

SMP is built on discrete-event simulation (DES) theory where the behaviour of a system is modelled as a discrete sequence of events in time. Each event that marks a change in the state of the systems occurs at a particular instant in time. The simulation can jump in time from one event to the next since no change in the system occurs between consecutive events.

The main elements in SMP that support this approach are:

- The simulation components schedule EntryPoints (see clause 5.2.7.1) on the SMP scheduler (see clause 5.3.3) for execution of events.
- The Simulation state is captured in the persisted data.

4.6.2 Parallelization and distribution

SMP assumes a single scheduler executing its events in sequence. All components are loaded inside the same address space allowing direct communication between them. The standard however does not prevent parallelization or distribution to be built into layers on top of the standard.

4.6.3 Inter component communication

4.6.3.1 Overview

SMP supports the following main method of communication between components:

- Direct interface based communication
- Data flow based communication
- Event based communication

4.6.3.2 Interface based communication

An interface-based design adds interfaces as the standard mechanisms for inter-model communication. This isolates the definition of an interface (the “contract”) from its implementation. In an interface-based design, a model provides any number of interfaces. An Interface defines a contract between models. Every model implementing the interface provides all the functionality of the interface, so that every model, which consumes this interface can rely on the interface implementation. As interfaces are a mechanism to de-couple models, they do not give access to fields, but only to operations. With special operations (i.e. use of Properties. See definition 3.2.33 "property") that read or write a single value, access to fields can be added.

4.6.3.3 Data flow based communication

4.6.3.3.1 Overview

In a data flow based communication, two components exchange the value of a field. The Provider component transfers an Output field value into an Input field of the Consumer component. The Output field is said to be connected to the Input field through a Dataflow connection.

The dataflow communication can be automatic, i.e. whenever the Output field is updated by its owning component, the value is immediately propagated to the connected Input field. The output fields which take part in an automatic dataflow based communication implement IDataflowField (see 5.2.11.7).

The dataflow communication can be scheduled, i.e. when the Output field is updated by its owning component, the value is not automatically propagated to the connected Input field. The transfer is only performed on request, e.g. cyclically by the Simulation Environment.

It is neither mandated when an Output field pushes its value into connected Input fields, nor whether an Input field performs any specific action after it has been updated. An Input field can be implemented in a way that it notifies its containing model about a change, which can be used to trigger certain behaviour in the Consumer component.

4.6.3.3.2 Data types consideration

When two fields are involved in a dataflow connection, compatibility of the field types ensures correct transfer of data between the provider model and the consumer model.

Two levels of type compatibility are specified for the fields:

a. Strict compatibility: both fields are typed by the same type as identified by their type UUID published in the Type Registry. In this case, it is obvious that the fields can be connected via a dataflow connection.

b. Equivalence: the types are equivalent as per their semantics or physical representation. For example, when type 1 is a user-defined type Voltage mapped on a Smp::Float64 and type 2 is an user-defined type Tension mapped on Smp::Float64. As both type 1 and type 2 are Smp::Float64, they are said “equivalent”. Dataflow connection is allowed as there is no issue to transfer information between type 1 and type 2.

4.6.3.4 Event Based communication

For an event-based design the components are modelled using Event Sinks (see 5.2.6.1) and Event Sources (see 5.2.6.2). Events issued by the source are received by all sinks subscribed to receive the corresponding event.

4.7 Models, Services and Components

4.7.1 Objects

An object is the base class for all SMP elements. It provides the basic features of a name, description and parent to all SMP elements. This implies that all SMP elements are organized in a hierarchical structure and always able to traverse upwards towards its root. The elements inheriting directly from Object are:

- Entry Points (IEntryPoint) as void operations that can be called by the scheduler and event manager services;
- Entry Point Publisher (IEntryPointPublisher) for publishing of entry points;
- Event Sinks (IEventSink) and Event Sources (IEventSource) for event based communication between objects;
- Composites (IComposite) and Containers (IContainer) to build the object hierarchy;
- Collection (ICollect) allowing to collect SMP elements in a collection;
- References (IReference) allowing objects reference other components;
- Components (IComponent) implementing the simulation behaviour;
- Types (IType) to be used for the definition of fields;

- Factory (IFactory) that creates components;
- Persistent Objects (IPersist) that can store and restore their state. IPersist is the basis for the following important elements:
  - Fields (IField) to hold the simulation state and data;
  - Failures (IFailure) to allow objects to represent a failure in a system.

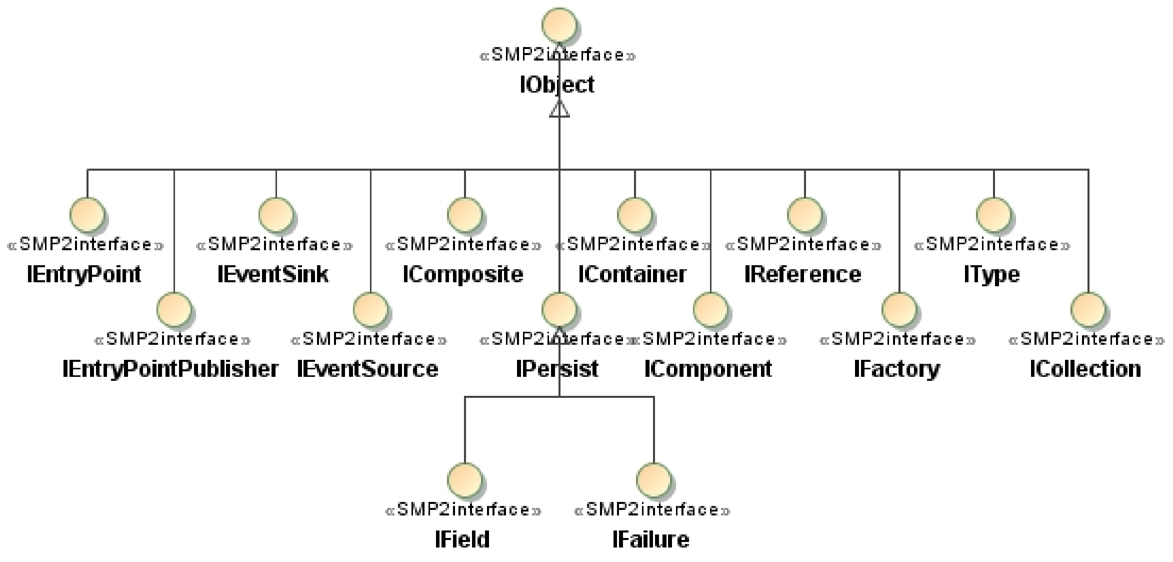
«SMP2interface» IObject
«SMP2interface» IEntryPoint
«SMP2interface» IEventSink
«SMP2interface» IComposite
«SMP2interface» IContainer
«SMP2interface» IReference
«SMP2interface» IType
«SMP2interface» IEntryPointPublisher
«SMP2interface» IEventSource
«SMP2interface» IPersist
«SMP2interface» IComponent
«SMP2interface» IFactory
«SMP2interface» ICollection
«SMP2interface» IField
«SMP2interface» IFailure

Figure 4-4: Object mechanisms

4.7.2 Components

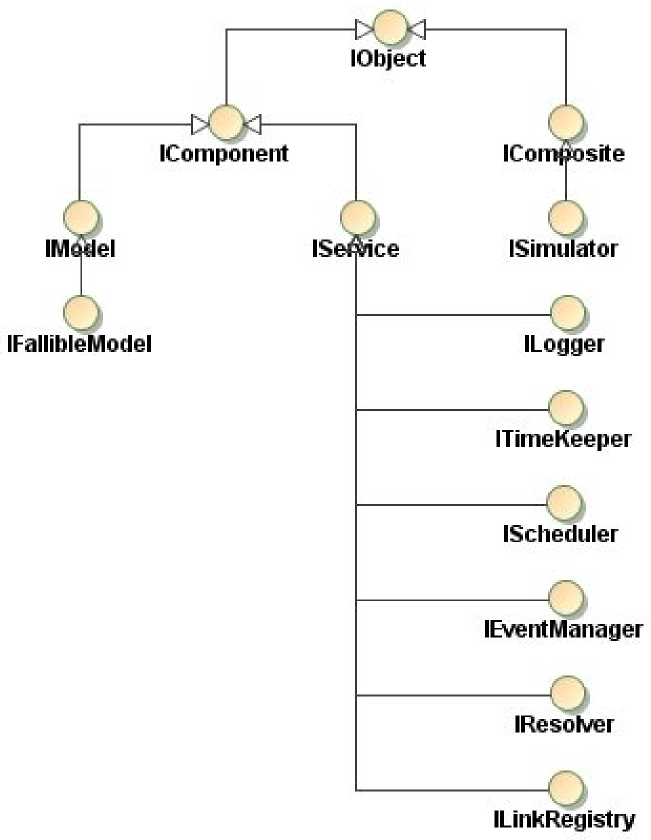

Figure 4-5: Overview of components hierarchy

The functionality of an SMP based simulator is implemented in elements that implement the IComponent (see 5.2.3.1) interface. A component represents an implementation of a self-contained feature with well-defined interfaces to other components. At initialization time, a simulation is built by assembling a set of instances of components. In addition, to implementing the IComponent interface, a number of additional optional component mechanisms are specified (see Figure 4-6):

- Component aggregation (IAggregate),
- Inter-component events (IEventProvider and IEventConsumer),
- Dynamic invocation (IDynamicInvocation),
- Link management features (ILinkingComponent),

See clause 5.2 for details on Component Mechanisms.

The Simulator itself is an object, in particular a composite. All its direct children are components, namely models and services.

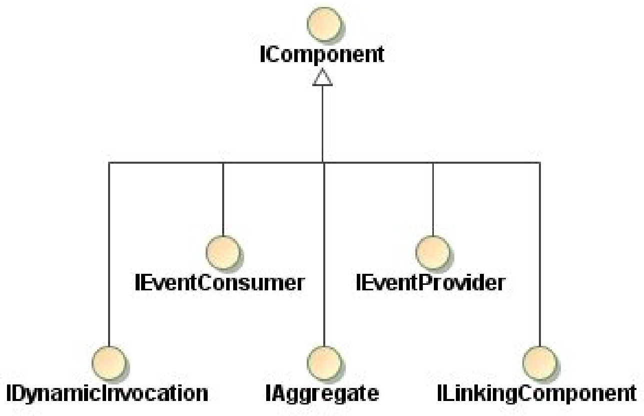

Figure 4-6: Component Mechanisms

All SMP components goes via a lifecycle as defined in Figure 4-7. Each component is responsible to ensure that this state diagram is followed. It is controlled via the IComponent interface (see clause 5.2.3.1).

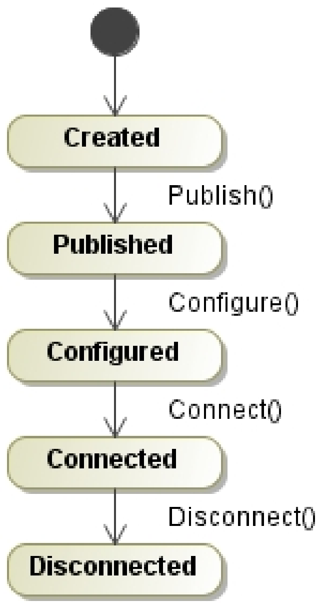

Figure 4-7: Component State machine

Each state in Figure 4-7 has its own purpose and behaviour as explained in Table 5-2.

4.7.3 Factories

A Factory is an Object that creates Components. The type of the Component instantiated by the Factory is identified by a Universally Unique Identifier (UUID). The UUID is a 128-bit number which for practical purposes is unique, without depending for uniqueness on a central registration authority or coordination between the parties generating them. The purpose of the Factory is to hide the implementation details of how a Component is instantiated. For example, the base class of the Component implementation is hidden by the Factory.

A Factory is an Object that implements the IFactory interface. The Factory is registered with the simulator by calling the ISimulator::RegisterFactory method. Instances of the Component identified by the Component's UUID can then be created by calling the ISimulator::CreateInstance method which uses the registered Factory to instantiate the Component.

4.7.4 Models and Services

Two main flavours of components are specified by SMP: Models and Services. The main differences are:

- Models implement the IModel interface while Services implement the IServices interface. Both interfaces are empty and do not add any additional capabilities, but the difference allows to efficiently differentiate models and services.
- Models are added to the simulation in a hierarchical tree, while services live in the global scope of the simulation.
- The Models can be fallible by implementing the IFallibleModel interface but Services are not.
- Models are added to the Simulations via the ISimulator AddModel method, while Services are added via the AddService method.
- It is possible to get a reference to a service from the ISimulator interface via the GetService method by its name. This implies that all components in a simulation can easily obtain a reference to a service.
- Services can only be added to the simulation during the first startup in building phase, while models can be dynamically added also later in stand-by state.

The mandatory features of an SMP runtime environment are specified as services. See clause 5.3 and Figure 4-5.

4.8 Publication and Persistence

SMP components publish their state information to the simulation environment to:

- Allow visualization of the simulation state.
- Allow the simulation environment to interact with the state of the component.
- Allow the simulation environment to store and restore the state of the component via the SMP persistence mechanism.

All published fields are annotated with a set of attributes provided by the SMP component to the Simulation Environment:

- A View Kind attribute indicating which kind of user this information is intended for. The values and intended interpretation of these values by the Simulation Environment is given in Table 4-2.
- If the field is part of the breakpoint or not (State attribute of field).
- If it is an Input, or an Output, or an Input/Output field (Input and Output attributes of field).

Additional meta information can also be provided via the SMP Catalogue (see 5.4.1).

Table 4-2: ViewKind values

Name	Intended interpretation
VK_None	The element is not made visible to the user.
VK_Debug	The element is made visible for debugging purposes. The element is not visible to end users. If the simulation environment supports the selection of different user roles, then the element is intended to be visible to "Debug" users only.
VK_Expert	The element is made visible for expert users. The element is not visible to end users. If the simulation environment supports the selection of different user roles, then the element is intended to be visible to "Debug" and "Expert" users.
VK_All	The element is made visible to all users. (this is the default)

From the list of published fields, the simulation environment is able to determine the state of a simulation and store it into a breakpoint (or to restore it when needed). This is called persistence. Persistence of SMP components can be handled in one of two ways:

- **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 (See 5.3.9.1) interface.
- **Self-Persistence**: The component can implement the IPersist (See 5.2.9) interface, which allows it to perform special operations during store and restore in addition to external persistence. Typically, self-persistence allows the persistence of dynamic data structure (e.g. events on the

simulation schedule). Two approaches exist in this case for models to store their data:

- Its state or parts of it can be stored/restored in the storage that is provided by the simulation environment via the IStorageReader (see 5.3.8.1) and IStorageWriter (see 5.3.8.2) interfaces provided by the simulation environment.
- The component can query the filename and location of the storage file from the environment via the IStorageReader (see 5.3.8.1) and IStorageWriter (see 5.3.8.2) interfaces and store additional files in the same location. This mechanism is usually only needed by specialised models, for example embedded models that need to load on-board software from a specific file.

SMP Runtime Environments supports both External and Self-Persistence. For models and components, only external persistence (via the Store() and Restore() methods of the ISimulator interface) is a mandatory feature, while self-persistence is an additional optional mechanism.

4.9 Dynamic invocation

SMP supports dynamic invocation allowing interaction between simulation environments and simulation models. This is typically used during execution allowing to control a simulation via scripting. It is a mechanism that makes the operations of a component available via a standardised interface.

In order to allow calling a named method with any number of parameters, a request object is created which contains all information for the method invocation. This request object is also used to transfer back a return value. The dynamic invocation concept standardises the request objects (IRequest interface, see 5.2.8.2). 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 as 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 4-8 shows all steps involved when using the CreateRequest() and DeleteRequest() methods.

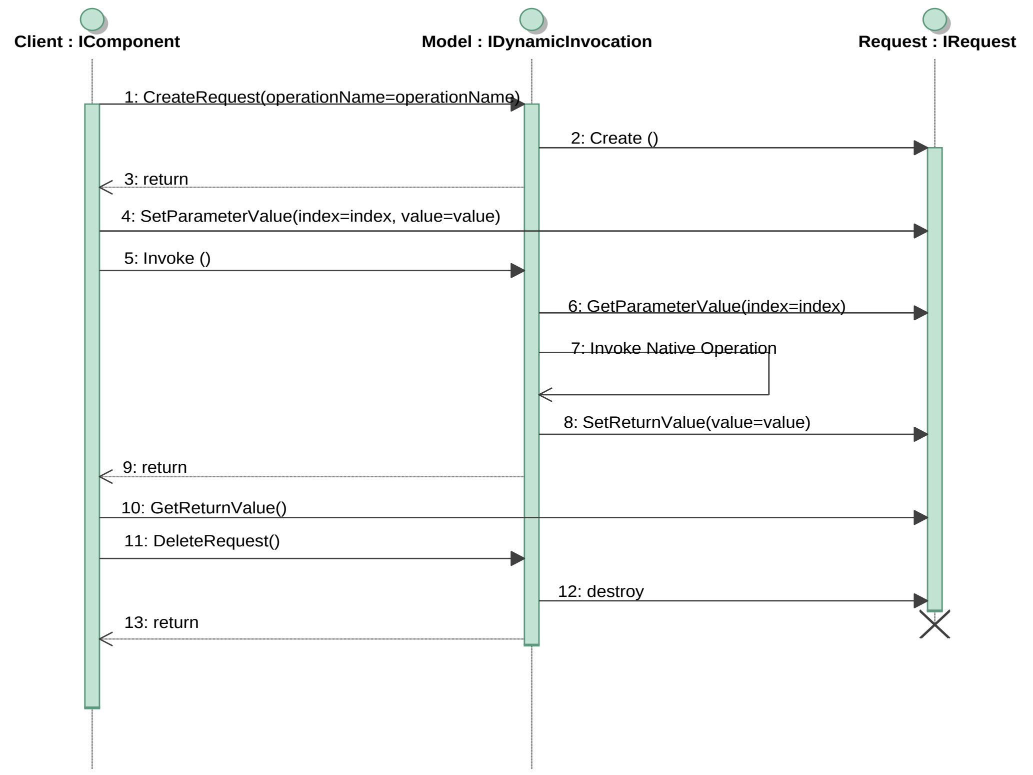
Client : IComponent
Model : IDynamicInvocation
Request : IRequest
1: CreateRequest(operationName=operationName)
2: Create ()
3: return
4: SetParameterValue(index=index, value=value)
5: Invoke ()
6: GetParameterValue(index=index)
7: Invoke Native Operation
8: SetReturnValue(value=value)
9: return
10: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

Figure 4-8: Sequence of calls for dynamic invocation

The sequence diagram in Figure 4-8, 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.

4.10 Components meta data

4.10.1 Catalogue

Meta data for SMP objects are stored in XML documents called the Catalogue. Having the SMP objects described in XML catalogues allows taking benefit from the XML language, for example:

- Generation of the catalogues from UML diagrams
- Generation of models documentation from the catalogue
- Generation of models skeleton code from the catalogue (See clause 6.1).

The content of a catalogue is hierarchically ordered in namespaces that may be nested. Inside each namespace many uniquely named instance of the following SMP features can be found:

- Types definitions including:
  - Constants, Fields and Properties
  - Exceptions
  - Data Types
- Interfaces specifications
- Component and model specifications including:
  - Event Sinks and sources
  - Fields and properties
  - Entry Points
  - Operations
  - Containment and inheritance
  - Interfaces, associations and references
- Attributes that can be attached to elements:
  - Fallible and Forcible
  - Min and Max limits for types
  - View/ViewKind information determines the visibility of the element

For all the elements above, meta data can be added like the description of each element or the engineering unit for type definitions. From this, it can be seen that the Catalogue definition provides a rich capability to describe the complete external interface of all SMP components. In fact, the interfaces as described in the SMP standard can as well be expressed in a catalogue. (See ecss.smp.smpcat as referenced in clause 5.4.1.2.1a).

4.10.2 Package

A package describes how implementations of types defined in catalogues are packaged. This includes not only models, which may have different implementations in different packages, but as well all other user-defined types.

4.10.3 Configuration

A configuration document allows specifying arbitrary field values of component instances in the simulation hierarchy. This can be used to initialise or reinitialise the simulation.

4.11 Model exchanges considerations

4.11.1 Overview

One of the primary goal of SMP is to allow model exchanges based on the Package concept.

Model source code exchange are considered easier than binary exchange as some considerations are important to be taken into account when exchanging binary models.

The mapping of a Package to C++ defines which symbols a static or dynamic library of SMP has to expose. This enables binary distribution of models, where only the catalogues and/or header files (for the compiler) and the libraries (for the linker) are provided, but no implementation source code. Nevertheless, binary compatibility depends on a number of other constraints, which may even vary between operating systems and compilers.

4.11.2 SMP Bundle

For distribution of a binary package SMP bundles are used. A SMP Bundle is an archive (e.g. a tar file on Linux, or a zip file on Windows) which provides the following elements:

- One or more SMDL packages.
- One or more package dynamic libraries, directly related to the SMDL packages.
- One or more package static libraries, directly related to the SMDL packages.
- All the SMP catalogues related to the SMDL packages.
- Optionally include other artefacts (SMDL configurations) and/or the related source code for all or parts of the included SMDL packages.

The related structure of folders and files within the bundle, and the names of folders and files are not standardised.

The added value of a Bundle is the additional SMP.MF Bundle Manifest file.

This Manifest is an ASCII file (aligned with the OSGi bundle manifest format) which contains key-value pairs with important meta data for the bundle.

5 Interface requirements

5.1 Common

5.1.1 Primitive Types specification

a. All SMP fields, parameters, constants and properties shall be of either a Primitive Type as per PrimitiveTypes.h in [SMP_FILES], or a User Defined Type published to the Type Library.

NOTE This specification is compliant with the types specified in Table 5-1.

b. Mapping between SMP types, XML types and ISO/ANSI C++ types shall be as per Table 5-1.

NOTE C++ mapping for primitive types is provided by PrimitiveTypes.h in [SMP_FILES].

Table 5-1: Primitive Types

SMP Type	XML mapping	C++ mapping	Description
Char8	xsd:string	char	8 bit character type to represent textual characters
String8	xsd:string	const char*	8-bit character strings based on UTF-8 encoding, which is commonly used in XML
Bool	xsd:boolean	bool	Bool is a binary logical type with values true or false
Int8	xsd:byte	int8_t	8 bit signed integer
UInt8	xsd:unsignedByte	uint8_t	8 bit unsigned integer
Int16	xsd:short	int16_t	16 bit signed integer
UInt16	xsd:unsignedShort	uint16_t	16 bit unsigned integer
Int32	xsd:int	int32_t	32 bit signed integer
UInt32	xsd:unsignedInt	uint32_t	32 bit unsigned integer
Int64	xsd:long	int64_t	64 bit signed integer
UInt64	xsd:unsignedLong	uint64_t	64 bit unsigned integer
Float32	xsd:float	float	IEEE 754 single-precision floating-point type with a length of 32 bits.

SMP Type	XML mapping	C++ mapping	Description
Float64	xsd:double	double	IEEE 754 double-precision floating-point type with a length of 64 bits.
Duration	xsd:duration	int64_t	Duration in nanoseconds. See 5.1.1c for detailed specification.
DateTime	xsd:dateTime	int64_t	Absolute time in nanoseconds. See 5.1.1d for detailed specification

c. The Duration type as per Table 5-1 shall be used for specifying a duration, as follows:

1. It is expressed in nanoseconds;
2. It is stored in a signed 64 bit integer;
3. Positive values correspond to positive durations;
4. Negative values correspond to negative durations.

NOTE 1 Nanoseconds is the lowest level of granularity supported for time in SMP.

NOTE 2 The duration type allows specifying values roughly between -290 years and 290 years.

NOTE 3 The duration type allows expression of relative time, hence "negative duration" implies a relative time in the past.

d. The DateTime type as per Table 5-1 shall be used for absolute time values, as follows:

1. It is expressed in nanoseconds, relative to the reference time of 01.01.2000, 12:00, Modified Julian Date (MJD) 2000+0.5;
2. It is stored in a signed 64 bit integer;
3. Positive values correspond to times after the reference time;
4. Negative values correspond to time values before the reference time.

NOTE 1 Nanoseconds is the lowest level of granularity supported for time in SMP.

NOTE 2 DateTime allows specifying time values roughly between 1710 and 2290.

e. A SMP Simple Field shall be of a type that maps directly to a Primitive Type.

f. The AnySimple type shall hold a Primitive Type as per AnySimple.h in [SMP_FILES].

g. The AnySimpleArray type shall be an array of AnySimples as per AnySimpleArray.h in [SMP_FILES].

5.1.2 Time Kinds

a. Simulation time shall be used for keeping the progress of time with respect to the start of the simulation, with the following properties:

1. Simulation time is a non-negative Duration type;
2. Simulation time is initialised to 0 at the beginning of the Building state as per Table 4-1;
3. Simulation time changes only when:
   (a) The simulation is progressing in the Executing state;
   (b) As a result of a restore of a breakpoint in restoring state;
   (c) As a result of ITimeKeeper SetSimulationTime.
4. It is not specified how quickly simulation time is progressed when the simulator is in Executing state;
5. Simulation time is stored and restored in the storing and restoring states;

b. Mission time shall be used for keeping the progress of relative time with respect to a Mission Start time, with the following properties:

1. Mission time is initialised to 0 at the beginning of the Building state as per Table 4-1;
2. Mission Time is calculated as a fixed offset between the current Epoch time and the given Mission start time according to the following formula: MissionTime = EpochTime – MissionStartTime;
3. The Mission time progresses with Epoch time, which progresses with Simulation time, and is hence affected by the ITimeKeeper SetEpochTime method.
4. The Mission time offset from Epoch time, the Mission start time changes by calls to:
   (a) the ITimeKeeper SetMissionTime method;
   (b) the ITimeKeeper SetMissionStartTime method.
5. Mission time only progresses when the simulation environment is in Executing state;
6. Mission time is stored and restored in the storing and restoring states;
7. Mission time is stored as a Duration type.

c. Zulu time shall be time dependent on the system clock of the host machine or an external clock source expressed using the DateTime type.

NOTE High Real Time systems sometimes uses an external clock source instead of the local system clock of host machine.

d. Epoch time shall be time dependent on the Simulation time with a fixed offset using the DateTime type.

NOTE 1 Epoch time progresses with Simulation time.

NOTE 2 Epoch time is changed with the ITimeKeeper SetEpochTime (See 5.3.2).

5.1.3 Path string

a. An SMP path string shall be a representation of a valid route from an SMP object in the hierarchy to another SMP object.

NOTE 1 Examples of valid path strings:

- /Satellite/Receivers/Receiver1
- /Logger
- /Logger/
- ../../Transmitters/Transmitter4
- ./Satellite/../Satellite//Receivers/

NOTE 2 Examples of invalid path strings:

- "/..", parent of root object do not exist
- "…", meaning of triple dots not defined.

b. Both Absolute and Relative path strings shall be supported and distinguished as follows:

1. Paths starting with a delimiter are absolute paths from the simulation root object.
2. Paths not starting with a delimiter are relative paths from the current object.

c. The delimiter between component names in the path string shall be "/".

d. The delimiters between components and its children objects that are not components shall be either "/" or ".".

NOTE This allows "Component.Operation()" to be used as path.

e. Trailing delimiters shall be allowed in path strings.

f. It shall be possible to reference the parent object by the ".." string.

g. It shall be possible to reference the current object by:

1. the "." string
2. an empty string ""

NOTE This allows the following to be used as path to operations of current object:

- .Operation()
- ./Operation()

h. The path string shall allow an element in an array to be identified by "[n]" trailing the array name where "n" is the zero based element index, with no delimiter.

NOTE This allows the following to be used for addressing element 2 of an array "MyArray" in MyModel:

- MyModel/MyArray[2]
- MyModel.MyArray[2]

5.1.4 Universally Unique Identifiers (UUID)

a. All SMP types shall have a unique UUID as per Uuid.h in [SMP_FILES].

NOTE 1 The UUID follows the specification from Open Group (http://pubs.opengroup.org/onlinepubs/9629399/apdxa.htm)

NOTE 2 The UUID is a 128 bit long unique identifier.

NOTE 3 The UUID allows for example to:

- Uniquely identify types defined in catalogues so that can be bound with implementations defined within packages.
- Uniquely identify linked elements within a Catalogue.

5.1.5 Exception specification

a. All SMP exceptions shall inherit from the Exception class as per Exception.h in [SMP_FILES] providing the following information:

1. The description of the exception;
2. The name of the exception;
3. The exception message;
4. The sender of the exception when the exception originates from an SMP Object.

NOTE This covers both exceptions defined in this standard and user defined exceptions.

5.2 Components and Objects interfaces

5.2.1 Object Specification (IObject)

a. All SMP objects shall provide the following features as per IObject.h in [SMP_FILES]:

1. If the object is not an array element, a name of the object as follows:
   (a) Not be empty;
   (b) Start with a letter;
   (c) Contain only letters, digits, and underscore ("_");
   (d) Not be an ISO/ANSI C++ keyword.
2. If the object is an array element, the name shall be the array name appended by “[i]“ where “i” is a zero based element index;
3. A description of the object;
4. The parent object as follows:
   (a) An IObject pointer to the parent if the object has a parent;

(b) A nullptr if the object does not have a parent.

NOTE 1 The Object description may be empty.

NOTE 2 All SMP elements inherit from the IObject interface including:

- Entry Points
- Event Sinks and Sources
- Fields
- Containers
- References
- Failures
- Components
- Composites
- Collections
- Factories
- Types

NOTE 3 to item 5.2.1a.1(d): See ISO/IEC 9899:2011 [C11 Standard] and ISO/IEC 14882:2011 [C++11 Standard] for the actual list of keywords.

b. All SMP objects with the same parent that are to be resolved by the Resolver shall have a unique name.

NOTE Containers and References cannot be resolved via the resolver, hence they do not need a unique name.

c. The validity of the SMP name shall be checked when an SMP object is created, with the following behaviour:

1. If an object with an invalid name is created, it throws a InvalidObjectName exception as per InvalidObjectName.h in [SMP_FILES].

5.2.2 Collection Specification (ICollection)

a. All SMP Collections of SMP elements shall implement the ICollection interface as per ICollection.h in [SMP_FILES].

b. The ICollection at method shall return the element with the given position or name, with the following behaviour:

1. If no element exists with the given position or name, it returns nullptr.

c. The ICollection size method shall return the number of elements in the collection.

5.2.3 Component Specification

5.2.3.1 Component (IComponent)

a. All SMP Components shall implement the IComponent interface as per IComponent.h in [SMP_FILES].

b. The IComponent GetState method shall return the current state of the component as per ComponentStateKind.h in [SMP_FILES], specified in Table 5-2.

Table 5-2: Component states

Name	Description
CSK_Created	The Created state is the initial state of a component. Component creation is done by an external mechanism, e.g. by factories. This state is entered automatically after the component has been created. This state is left via the Publish() state transition.
CSK_Publishing	In Publishing state, the component is allowed to publish features. This includes publication of fields, operations and properties. In addition, the component is allowed to create other components. This state is entered via the Publish() state transition. This state is left via the Configure() state transition.
CSK_Configured	In Configured state, the component has performed initial configuration. This configuration can be done by external components, or internally by the component itself, e.g. by reading data from an external source. This state is entered via the Configure() state transition. This state is left via the Connect() state transition
CSK_Connected	In Connected state, the component is connected to the simulator. In this state, neither publication nor creation of other components is allowed anymore. Configuration performed via loading of SMDL configuration file and/or calling of initialisation entry point are performed in this state. This state is entered via the Connect() state transition. This state is left via the Disconnect() state transition or on simulation termination.
CSK_Disconnected	In Disconnected state, the component is disconnected from the simulator, and all references to it are deleted, so that it can be deleted. This state is entered via the Disconnect() state transition. This is the final state of a component, and only left on deletion.

c. The IComponent Publish method shall be used by components to publish all publishable fields, properties and operations, with the following argument and behaviour:

1. Argument:
   (a) "receiver" giving a pointer to the IPublication instance for the component.
2. Behaviour:
   (a) If the component is not in Created state, then it throws an InvalidComponentState exception as per InvalidComponentState.h in [SMP_FILES];
   (b) If the component is in Created state, then it enters the Publishing state;
   (c) After entering Publishing state, it publishes its fields, properties and operations using the provided receiver argument;
   (d) While in publishing state, it can create new components;

NOTE 1 Components can override the implementation of operations and properties from their parents, hence it is possible that the same property and operation are published multiple times. In this case, the last call to published overrides the previous calls.

NOTE 2 Newly created components are in Created state. The simulator is responsible for the triggering of state transitions of new components.

d. The IComponent Configure method shall be used to perform initial configuration of the component, with the following arguments and behaviour:

1. Arguments:
   (a) "logger" giving a pointer to the ILogger instance for the component, to provide the possibility to log messages during its configuration;
   (b) "linkRegistry" giving a pointer the ILinkRegistry instance for the component, to provide the possibility to register links.
2. Behaviour:
   (a) If the component is not in Publishing state, it throws an InvalidComponentState exception as per InvalidComponentState.h in [SMP_FILES];
   (b) If the component is in Publishing state, it creates and configures other features and even other components using the field values of its published fields as sole source of configuration information for the creation of such components;
   (c) After completing the configuration actions, the component enters Configured state.

e. The IComponent Connect method shall allow the components to connect to the simulator environment and other components, with the following argument and behaviour:

   1. Argument:

      (a) "simulator" giving a pointer to the ISimulator interface as per ISimlator.h in [SMP_FILES] to access services from the simulation environment.

   2. Behaviour:

      (a) If the Component is not in Configured state, it throws an InvalidComponentState exception as per InvalidComponentState.h in [SMP_FILES];

      (b) If called in Configured state, the component enters Connected state;

      (c) After entering Connected state, it connects to simulation services used by the component, if any.

      NOTE It is guaranteed that all models have been created, published and configured before the Connect method of any component is called.

f. The IComponent Disconnect method shall disconnect the component from the simulation environment and any other components, with the following behaviour:

   1. If the Component is not in Connected state, it throws an InvalidComponentState exception as per InvalidComponentState.h in [SMP_FILES];

   2. If called in Connected state, the component enters Disconnected state;

   3. After entering Disconnected state, the component disconnects from simulation services by deleting all references of these services to the component.

g. The IComponent GetField method shall provide access to the IField interface for fields of the component, taking the following argument and behaviour:

   1. Argument:

      (a) "fullName" giving the path of the field for whom it returns the IField interface.

   2. Behaviour:

      (a) If the passed fullName does not exist, it throws an InvalidFieldName exception as per InvalidFieldName.h in [SMP_FILES];

      (b) If the passed field name exists and it is a field of simple type it returns its ISimpleField interface;

      (c) If the passed field name exists and it is an array field it returns its IArrayField or ISimpleArrayField interface;

      (d) If the passed field name exists and it is a structure field it returns its IStructureField interface.

      NOTE This includes fields of structures and items of arrays.

h. The IComponent GetFields method shall return a collection of the component fields as per FieldCollection in IField.h in [SMP_FILES].

i. The IComponent GetUuid method shall return a reference to the Uuid of the component, as per Uuid.h in [SMP_FILES].

5.2.3.2 Model (IModel)

a. All SMP Components which contain the implementation of the simulations functional behaviour shall implement the IModel interface as per IModel.h in [SMP_FILES].

5.2.3.3 Service (IService)

a. All SMP components which implement a service to be used by other SMP models shall implement the IService interface as per IService.h in [SMP_FILES].

NOTE This includes both standard services specified in this standard and user defined services.

b. All SMP components which implement the IService interface shall ensure their state is fully persisted in a simulation breakpoint and restored on Restore.

5.2.3.4 Linking Component (ILinkingComponent)

a. All SMP Components which require dynamic removal of links at runtime shall implement the ILinkingComponent interface as per ILinkingComponent.h in [SMP_FILES].

b. The ILinkingComponent RemoveLinks method shall remove all links to the passed component stored in the LinkingComponent itself, taking the following argument:

1. “target” giving the reference to the linked component.

NOTE The result of this removal is that the LinkingComponent can no longer access the target component removed.

5.2.4 Aggregation

5.2.4.1 Aggregation interface (IAggregate)

a. All SMP Components which are referencing other components shall implement the IAggregate interface as per IAggregate.h in [SMP_FILES].

NOTE The IReference interface is the referencing mechanism used by the aggregation interface.

b. The IAggregate GetReference method shall return the reference matching the given name, with the following argument and behaviour:

1. Argument:

(a) “name” giving name identifying the reference.

2. Behaviour:

    (a) If no reference matching the given name is found, it returns a nullptr reference.

c. The IAggregate GetReferences method shall return an ordered collection of all references, with the following behaviour:

    1. If the aggregation does not hold any reference, it returns an empty collection;

    2. If at least one reference is contained, it returns a collection ordered according to the order in which the references have been added to the aggregate.

5.2.4.2 Reference Interface (IReference)

a. All references returned by an aggregate shall implement the IReference interface as per IReference.h in [SMP_FILES].

    NOTE A reference is a named object.

b. The IReference GetComponent method shall return a reference to the component matching the given name with the following argument and behaviour:

    1. Argument:

        (a) "name" giving the name of the referenced component to be returned.

    2. Behaviour:

        (a) If no component matching the given name argument is found, it returns a nullptr reference;

        (b) If multiple components matching the given name argument are found, it returns one of the references.

    NOTE Multiple components with the same name, but with a different parent (and hence path) can end up in a single reference. In this case, retrieving a component by name is not safe, as any of the components that match the name can be returned.

c. The IReference GetComponents method shall return an ordered collection of all the referenced components with the following behaviour:

    1. If no component is referenced, it returns an empty collection;

    2. If at least one component is contained, it returns a collection ordered according to the order in which the components have been added using the AddComponent method.

d. The IReference AddComponent method shall add a component to the collection of referenced components, with the following argument and behaviour:

    1. Argument:

        (a) "component" giving a reference to the component to be added.

2. Behaviour:

   (a) If the maximum supported number of referenced components is reached, it throws a ReferenceFull exception as per ReferenceFull.h in [SMP_FILES];

   (b) If the reference interface implementation is expecting the given component to inherit from another type it throws an InvalidObjectType exception as per InvalidObjectType.h in [ZIPFLE].

   NOTE A (typed) reference can attempt to type-cast a component to a specific type, to ensure that all components within the reference inherit from this common base type.

e. The IReference RemoveComponent method shall remove a component from the collection of referenced components, with the following argument and behaviour:

1. Argument:

   (a) "component" giving a reference to the component to be removed.

2. Behaviour:

   (a) If the minimum number of component(s) referenced by this object is reached, it throws a CannotRemove exception as per CannotRemove.h in [SMP_FILES];

   (b) If the component to remove is not referenced, it throws a NotReferenced exception as per NotReferenced.h in [SMP_FILES].

   NOTE RemoveComponent ensures that the right component is identified also if several components with the same name exist in the reference, as it takes a reference to the component as argument, and not the name.

f. The IReference GetCount method shall return the number of components in the collection of referenced components.

g. The IReference GetUpper method shall return the upper limit, with the following behaviour:

1. If a maximum number has been defined, it returns the maximum number;

2. If no maximum number has been defined, it returns -1.

   NOTE The usage of -1 is consistent with the use of upper bounds in UML, where a value of -1 represents no limit (typically shown as *)

h. The IReference GetLower method shall return the minimum number of components in the collection or 0 when not defined.

   NOTE The lower bound can be used to validate a model hierarchy. If a collection specifies a Lower value above its current Count, then it is not properly configured. An external component can use this information to validate the configuration before executing it.

5.2.5 Composition

5.2.5.1 Composition interface (IComposite)

a. All SMP Objects which contain Components shall implement the IComposite interface as per IComposite.h in [SMP_FILES].

NOTE 1 The IContainer interface (see 5.2.5.1c.2) is the component container used by the composition interface.

NOTE 2 Composition is the counter part of the IObject GetParent() method and allows traversing the tree of components from parent to child components.

b. The IComposite GetContainer method shall return the container matching the given name with the following argument and behaviour:

1. Argument:

   (a) "name" giving the name of the container to be returned.

2. Behaviour:

   (a) If no container matching the given argument name is found, it returns a nullptr reference.

c. The IComposite GetContainers method shall return an ordered collection of all the containers with the following behaviour:

1. If the composite does not hold any container, it returns an empty collection.

2. If at least one container is contained, it returns a collection ordered according to the order in which the containers have been added to the composite.

5.2.5.2 Container interface (IContainer)

a. All SMP Objects which represent a composition of child Components shall implement the IContainer interface as per IContainer.h in [SMP_FILES].

NOTE 1 The container components life-cycle coincides with its parent one.

NOTE 2 The container is a named Object as per 5.2.1.

NOTE 3 The container allows adding children to a parent object.

NOTE 4 Each container holds objects of only one type.

b. The IContainer GetComponent method shall return the component matching the given name, with the following argument and behaviour:

1. Argument:

(a) "name" giving the name of the component to be returned.

2. Behaviour:

(a) If no component matching the given name is found, it returns nullptr.

NOTE As the container does not support component name duplication, it is not possible to get naming conflict when performing query.

c. The IContainer GetComponents method shall return an ordered collection of all the contained components with the following behaviour:

1. If no component is contained, it returns an empty collection;

2. If at least one component is contained, it returns a collection ordered according to the order in which the components have been added using the AddComponent method.

d. The IContainer AddComponent method shall add a component to the collection of contained components, with the following argument and behaviour:

1. Argument:

(a) "component" giving the component to be added.

2. Behaviour:

(a) If the maximum supported number of components is reached, it throws a ContainerFull exception as per ContainerFull.h in [SMP_FILES];

(b) If a component with the same name and parent already exists, it throws a DuplicateName exception as per DuplicateName.h in [SMP_FILES];

(c) If the container interface implementation is expecting the given component to inherit from another type, it throws an InvalidObjectType exception as per InvalidObjectType.h in [SMP_FILES].

NOTE A (typed) container can attempt to type-cast a component to a specific type, to ensure that all components within the container inherit from this common base type.

e. The IContainer GetCount method shall return the number of components contained in the collection.

f. The IContainer GetUpper method shall return the maximum number of components in the collection, with the following behaviour:

1. If the maximum number of elements for the collection has been defined, it returns the maximum number;

2. If the maximum number of elements for the collection has not been defined, it returns -1.

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

g. The IContainer GetLower method shall return the minimum number of components in the collection or 0 when not defined.

NOTE The lower bound can be used to validate a model hierarchy. If a collection specifies a Lower value above its current Count, then it is not properly configured. An external component can use this information to validate the configuration before executing it.

h. The IContainer DeleteComponent method shall delete a component from the collection of contained components, with the following argument and behaviour:

1. Argument:
   (a) "component" giving a reference to the component to be deleted.
2. Behaviour:
   (a) If the minimum number of component(s) contained by this object is reached, it throws a CannotDelete expection as per CannotDelete.h in [SMP_FILES];
   (b) If the component to delete is not contained, it throws a NotContained exception as per NotContained.h in [SMP_FILES];
   (c) If the component to delete is included, and the minimum number is not reached, then the component is removed from the collection, and its destructor is called.

5.2.6 Events

5.2.6.1 Sink of events interface (IEventSink)

a. All SMP Objects which receive event notifications shall implement the IEventSink interface as per IEventSink.h in [SMP_FILES].

NOTE The specification of event sinks ensures that notifications from the event sources they are subscribed to can be managed.

b. The IEventSink GetEventArgType method shall provide the primitive type kind of the argument expected by the event sink when it is notified about a given event, with the following behaviour:

1. If no argument is expected, it returns PTK_None.

NOTE 1 See 5.2.6.1c for the specification of how event sinks are notified.

NOTE 2 This operation allows for type checking when subscribing (see 5.2.6.2b) event sinks to event sources.

c. The IEventSink Notify method shall inform the object about the event, with the following arguments:

1. "sender" giving the reference to the event source calling the method;
2. "arg" giving context data together with the event notification.

NOTE See 5.2.6.2d for the specification of how event sources call this method.

5.2.6.2 Source of events interface (IEventSource)

a. All SMP Objects which represent the source of event notifications shall implement the IEventSource interface as per IEventSource.h in [SMP_FILES].

NOTE The specification of event sources ensures that event sinks (see 5.2.6.1) that wish to receive their notifications can subscribe to them.

b. The IEventSource Subscribe method shall add the given event sink to the list of subscribed event sinks, with the following argument and behaviour:

1. Argument:
   (a) "eventSink" giving the reference to the event.
2. Behaviour:
   (a) If the given event sink is already subscribed to the event source, it throws an EventSinkAlreadySubscribed exception as per EventSinkAlreadySubscribed.h in [SMP_FILES];
   (b) If the primitive type kind of the argument expected by the event sink is not semantically equivalent to the one of the event source as per Table 5-3, it throws an InvalidEventSink exception as per InvalidEventSink.h in [SMP_FILES].

NOTE Any event sink can only be subscribed once to each event source.

c. The IEventSource Unsubscribe method shall remove the given event sink from the list of subscribed event sinks, with the following argument and behaviour:

1. Argument:
   (a) "eventSink" giving the event to be unsubscribed.
2. Behaviour:
   (a) If the given event sink is not subscribed to the event source, it throws an EventSinkNotSubscribed exception as per EventSinkNotSubscribed.h in [SMP_FILES].

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

d. When the event source emits the event, it shall call the Notify method of all the subscribed event sinks in the same order as the sinks have been subscribed.

NOTE See 5.2.6.1 for specification of the event sinks interface.

5.2.6.3 Consumer of events interface (IEventConsumer)

a. All SMP Components which hold event sinks and want to allow external access to them shall implement the IEventConsumer interface as per IEventConsumer.h in [SMP_FILES].

NOTE The publication of event sinks ensures that they can subscribe to other component's event sources.

b. The IEventConsumer GetEventSinks method shall return a collection of all the contained event sinks, with the following behaviour:

1. If no event sink is contained, it returns an empty collection.

c. The IEventConsumer GetEventSink method shall return the component's event sink corresponding to the given name, with the following argument and behaviour:

1. Argument:
   (a) "name" giving the name of the Event Sink.
2. Behaviour:
   (a) If no event sink with the given name exists, it returns nullptr.

5.2.6.4 Provider of events interface (IEventProvider)

a. All SMP Components which hold event sources and want to allow external access to them shall implement the IEventProvider interface as per IEventProvider.h in [SMP_FILES].

NOTE The publication of event sources ensures that other component's event sinks can subscribe to them.

b. The IEventProvider GetEventSources method shall return a collection of all the contained event sources, with the following behaviour:

1. If no event source is contained, it returns an empty collection.

c. The IEventProvider GetEventSource method shall return the component's event source corresponding to the given name, with the following argument and behaviour:

1. Argument:
   (a) "name" giving the name of event source to be returned
2. Behaviour:
   (a) If no event source with the given name exists, it returns nullptr.

5.2.7 Entry points

5.2.7.1 Entry points calling interface (IEntryPoint)

a. All SMP Objects which represent a schedulable entry point shall implement the IEntryPoint interface as per IEntryPoint.h in [SMP_FILES].

NOTE The specification of entry points ensures that the scheduler or the event manager can trigger them when the relevant events are emitted.

b. The IEntryPoint Execute method shall be called when the triggering event is emitted.

5.2.7.2 Entry Points publisher interface (IEntryPointPublisher)

a. All SMP components which hold entry points and want to allow external access to them shall implement the IEntryPointPublisher interface as per IEntryPointPublisher.h in [SMP_FILES].

b. The IEntryPointPublisher GetEntryPoints method shall return a collection of all the contained entry points, with the following behaviour:

1. If no entry point is contained, it returns an empty collection.

c. The IEntryPointPublisher GetEntryPoint method shall return the component's entry point corresponding to the given name, with the following argument and behaviour:

1. Argument:

(a) "name" giving the name of the EntryPoint to be returned.

2. Behaviour:

(a) If no entry point with the given name exists, it returns nullptr.

NOTE The "name" always identifies a unique EntryPoint, as a component cannot have several EntryPoints with same name.

5.2.8 Dynamic Invocation

5.2.8.1 Dynamic invocation interface (IDynamicInvocation)

a. All SMP Components which allow the simulation environment to invoke operations on them shall implement the IDynamicInvocation interface as per IDynamicInvocation.h in [SMP_FILES].

b. All operations of simulation components callable through dynamic invocation shall be registered by the component using the IPublication interface.

NOTE See 5.2.12.2d for specification of the IPublication PublishOperation method to be used. Parameters of operations need to be of types registered in the type registry, which excludes operations with parameters of other types from dynamic invocation.

c. The IDynamicInvocation CreateRequest method shall return an instance of a request class for identifying the given operation, with the following argument and behaviour:

1. Argument
   (a) "operationName" giving the name of the callable method.
2. Behaviour:
   (a) If the operation with the given name is not callable through dynamic invocation, it returns nullptr;
   (b) If the operation with the given name is callable through dynamic invocation, a fully populated request object with all parameters of the operation shall be created and returned.

NOTE 1 The behaviour of this mechanism in the context of operation overloading is not specified.

NOTE 2 The calling object is responsible for memory management of the request object, and for its deletion via DeleteRequest.

d. The IDynamicInvocation Invoke method shall invoke the method referenced, with the following argument and behaviour:

1. Argument:
   (a) "request" giving the identification of the callable method, as a fully populated request object implementing IRequest (see 5.2.8.2).
2. Behaviour:
   (a) If the operation specified by the request parameter is not callable through dynamic invocation, it throws an InvalidOperationName exception as per InvalidOperationName.h in [SMP_FILES];
   (b) If the number of arguments specified by the request object does not match the number of parameters of the callable operation, it throws an InvalidParameterCount exception as per InvalidParameterCount.h in [SMP_FILES];
   (c) If the types of the arguments specified by the request object do not match the types of parameters of the callable operation, it throws an InvalidParameterType exception as per InvalidParameterType.h in [SMP_FILES];
   (d) If called with a valid request object, it calls the operation identified in the request, passing the parameters provided in the request which are of parameter direction In or InOut;
   (e) After invoking the request, it stores the parameter values of parameters with parameter direction InOut, Out or Return into the requests object.

NOTE The Invoke operation is a void operation as the result of the invocation is stored in the IRequest object (see 5.2.8.2.

e. The IDynamicInvocation DeleteRequest method shall release all resources associated to the given request instance.

f. All requests created with IDynamicInvocation CreateRequest shall be deleted with a call to IDynamicInvocation DeleteRequest.

g. The IDynamicInvocation GetProperties method shall return a collection of the invokable properties of the component as per PropertyCollection in IProperty.h in [SMP_FILES].

h. The IDynamicInvocation GetOperations method shall return a collection of the invokable operations of the component as per OperationCollection in IOperation.h in [SMP_FILES].

5.2.8.2 IRequest

a. All SMP Request objects which are used in dynamic invocation shall implement the IRequest interface as per IRequest.h in [SMP_FILES].

b. The IRequest GetOperationName method shall return the name of the callable operation represented by the request object.

NOTE Requests are usually created by calling the CreateRequest method of Dynamic Invocation (see 5.2.8.1) so the name returned is the string given to the CreateRequest method.

c. The IRequest GetParameterCount method shall return the number of parameters of the request object.

NOTE This operation only considers parameters of direction in, out or in/out, but not of type return.

d. The IRequest GetParameterIndex method shall return the index of a specified parameter, with the following argument and behaviour:

1. Argument:
   (a) "name" giving the name of the parameter for which the index is returned.
2. Behaviour:
   (a) If the name corresponds to the name of a parameter in the parameter collection, it returns the 0-based index of the parameter in this collection;
   (b) If no parameter with the given name exists, it returns -1.

NOTE This operation only considers parameters of direction in, out or in/out, but not of type return.

e. The IRequest SetParameterValue method shall store the value for a parameter, with the following arguments and behaviour:

1. Arguments:

(a) "index" giving the location of the parameter to be set;

(b) "value" giving the new value of the parameter.

2. Behaviour:

(a) If the index is less than zero, it throws an InvalidParameterIndex exception as per InvalidParameterIndex in [SMP_FILES];

(b) If the index is greater than or equal to the number of parameters of the request object, it throws an InvalidParameterIndex exception as per InvalidParameterIndex in [SMP_FILES];

(c) If the type of the given value is different than the type of the parameter at the given index, it throws an InvalidParameterValue exception as per InvalidParameterValue.h in [SMP_FILES];

(d) If both index and value are valid, it stores the new value into the parameter with the given index, so that its new value can be returned with future calls to GetParameterValue.

NOTE This operation only considers parameters of direction in, out or in/out, but not of type return.

f. The IRequest GetParameterValue method shall return the value stored at the given index in the parameters collection, with the following argument and behaviour:

1. Argument:

(a) "index" of the parameter for which the value is returned.

2. Behaviour:

(a) If the given index is less than zero, it throws an InvalidParameterIndex exception as per InvalidParameterIndex.h in [SMP_FILES];

(b) If the index is greater than or equal to the number of parameters of the request object, it throws an InvalidParameterIndex exception as per InvalidParameterIndex.h in [SMP_FILES];

(c) If the index is valid, it returns the current value of the parameter.

NOTE 1 The current value is either the initial value from creation of the request object, or the value provided to the last successful call of the SetParameterValue method for the same index.

NOTE 2 This operation only considers parameters of direction in, out or in/out, but not of type return.

g. The IRequest SetReturnValue method shall allow to set a return value in the request with the following argument and behaviour:

1. Argument:

   (a) "value" giving the new value to be set for the return parameter.

2. Behaviour:

   (a) If the operation does not return a value, it throws a VoidOperation exception as per VoidOperation.h in [SMP_FILES];

   (b) If the type of the provided value does not match the type of the return value of the operation, it throws an InvalidReturnValue exception as per InvalidReturnValue.h in [SMP_FILES];

   (c) If the operation does return a value of the given type, the return value is stored into the request object, so that it can be retrieved with later calls to GetReturnValue.

h. The IRequest GetReturnValue method shall return the return value of the callable operation in the request, with the following behaviour:

1. If the operation does not return a value, it throws a VoidOperation exception as per VoidOperation.h in [SMP_FILES];

2. If the operation does return a value, it returns the current value of the return parameter.

   NOTE The current value is either the initial value from creation of the request object, or the value provided to the last successful SetReturnValue call.

5.2.9 Persistence (IPersist)

a. All SMP Objects which need self-persistence of data shall implement the IPersist interface as per IPersist.h in [SMP_FILES].

   NOTE Self-persistence is an optional interface as external persistence by the simulation environment is sufficient for most components.

b. All Simulation objects which implement self-persistence shall read from the IStorageReader interface exactly the same amount of data and in the same order as it writes it to the IStorageWriter interface.

c. The IPersist Restore method shall read persisted data from storage through the IStorageReader interface with the following argument and behaviour:

1. Argument:

   (a) "reader" giving a pointer to a IStorageReader interface where data can be read from.

2. Behaviour:

(a) The operation restores exactly the same amount of data from the reader that was stored by the writer on Store;

(b) If the operation cannot restore the data, it throws a CannotRestore exception as per CannotRestore.h in [SMP_FILES].

d. The IPersist Store method shall write persisted data to the storage through the IStorageWriter interface with the following argument and behaviour:

1. Argument:

   (a) "writer" giving a pointer to a IStorageWriter interface where data can be written to.

2. Behaviour:

   (a) The operation stores exactly the amount of data to the writer than what it restores from a reader on Restore;

   (b) If the operation cannot store the data, it throws a CannotStore exception as per CannotStore.h in [SMP_FILES].

5.2.10 Failures

5.2.10.1 Failure interface (IFailure)

a. All SMP Objects which represent a failure shall implement the IFailure interface as per IFailure.h in [SMP_FILES].

b. The IFailure Fail method shall set the state of the failure to "failed".

c. The IFailure Unfail method shall set the state of the failure to "not failed".

d. The IFailure IsFailed method shall return the failure state of the failure with the following behaviour:

1. If the state is "failed", it returns true;
2. If the state is not "failed", it returns false.

5.2.10.2 Model failure state interface (IFallibleModel)

a. All Simulation models which can be failed through a list of possible failures shall implement the IFallibleModel interface as per IFallibleModel.h in [SMP_FILES].

NOTE 1 This is an optional interface.

NOTE 2 The simulation environment does not automatically persist the state of each failure, as it is the responsibility of the models to store the failure state in persisted data.

b. The IFallibleModel GetFailures method shall return the list of possible failures for this simulation model.

c. The IFallibleModel GetFailure method shall return a failure instance from the list of possible failures, with the following argument and behaviour:

1. Argument:
    (a) "name" giving the name of the failure.
2. Behaviour:
    (a) If none of the failures in the list of possible failures matches the given name, it returns nullptr.
    (b) If a failure matching the given name exists, it returns the pointer to the IFailure instance.

d. The IFallibleModel IsFailed method shall return the failure state of the model, with the following behaviour:
1. If at least one of the failures returns true for its IFailure::IsFailed among the list of possible failures for this simulation model, it returns true;
2. If none of the failures returns true for its IFailure::IsFailed among the list of possible failures for this simulation model, it returns false.

5.2.11 Field interfaces

5.2.11.1 ISimpleField

a. All SMP Fields which represent a primitive type shall implement the ISimpleField interface as per ISimpleField.h in [SMP_FILES].

b. The ISimpleField GetValue method shall return the field value stored in an AnySimple as per AnySimple.h in [SMP_FILES].

c. The ISimpleField SetValue method shall store the value in the field with the following argument and behaviour:
1. Argument:
    (a) "value" giving the new value to the field as an AnySimple as per AnySimple.h in [SMP_FILES].
2. Behaviour:
    (a) If the given value simple type kind does not match the simple type kind of the field, then it throws the InvalidFieldValue exception as per InvalidFieldValue.h in [SMP_FILES];
    (b) If the given value simple type kind does match the simple type kind of the field, then it changes the field value to the given value.

d. The ISimpleField GetPrimitiveTypeKind method shall return the primitive type kind of the field.

5.2.11.2 IStructureField

a. All SMP Fields which represent a structured data shall implement the IStructureField interface as per IStructureField.h in [SMP_FILES].

b. The IStructureField GetField shall return the field as an IField, with the following argument and behaviour:

1. Argument:
   (a) "name" given the field name for which the IField interface is returned.
2. Behaviour:
   (a) If the field name is unknown to the structure, it returns nullptr.

c. The IStructureField GetFields shall return the list of fields of the structure as per FieldCollection in IField.h in [SMP_FILES].

5.2.11.3 IArrayField

a. All SMP fields which represent an array where each array item is to be retrieved individually as a Field shall implement the IArrayField interface as per IArrayField.h in [SMP_FILES].

b. The IArrayField GetField shall return the array item as an IField as per IField.h in [SMP_FILES] with the following argument and behaviour:

1. Argument:
   (a) "index" giving the location of the item for which the IField pointer is returned.
2. Behaviour:
   (a) If the given index is outside the array size, it throws an InvalidArrayIndex as per InvalidArrayIndex.h in [SMP_FILES].
   (b) Otherwise, return the array item at the given index as an IField

c. The IArrayField GetSize method shall return the number of array items.

5.2.11.4 ISimpleArrayField

a. All SMP fields which represent an array of simple type items where individual array items are not to be retrieved as Field shall implement the ISimpleArrayField interface as per ISimpleArrayField.h in [SMP_FILES].

NOTE This enables an efficient implementation especially of large arrays as a single object, rather than having each array item represented by an individual object. The implications are that such array items cannot be retrieved as Object or Field, e.g. via GetField(), and are hence not available in operations that require individual objects or fields.

b. The ISimpleArrayField GetSize method shall return the number of array items.

c. The ISimpleArrayField GetValue shall return the corresponding array item value stored in an AnySimple as per AnySimple.h in [SMP_FILES] with the following argument and behaviour:

   1. Argument:
      (a) "index" giving the location of the item for witch the value is returned.
   2. Behaviour:
      (a) If the given index is outside the array size, it throws an InvalidArrayIndex as per InvalidArrayIndex.h in [SMP_FILES].
      (b) Otherwise, return the item value corresponding to the given index as an AnySimple.

d. The ISimpleArrayField SetValue shall set the corresponding array item value stored with the following arguments and behaviour:

   1. Arguments:
      (a) "index" giving the location of the item for witch the value is set;
      (b) "value" giving the new value for the array item.
   2. Behaviour:
      (a) If the given index is outside the array size, it throws an InvalidArrayIndex as per InvalidArrayIndex.h in [SMP_FILES].
      (b) If the given value simple type kind does not match the simple type kind of the corresponding array item, then it throws an InvalidFieldValue as per InvalidFieldValue.h in [SMP_FILES].
      (c) If the given value simple type kind does match the simple type kind of the corresponding array item, then it stores the given AnySimple value into the item value corresponding to the given index.

e. The ISimpleArrayField GetValues method shall return all the array item values stored in an AnySimpleArray as per AnySimpleArray.h in [SMP_FILES] with the following arguments and behaviour:

   1. Arguments:
      (a) "length" giving the length of the return array for values;
      (b) "values" giving an array of allocated storage for which the return values are put.
   2. Behaviour:
      (a) If the given value array size does not match the ArrayField size, it throws an InvalidArraySize as per InvalidArraySize.h in [SMP_FILES].
      (b) Otherwise, copy all the item values into the given AnySimpleArray.

f. The ISimpleArrayField SetValues method shall allow setting all values of an array with the following arguments and behaviour:

   1. Arguments:

      (a) "length" giving the length of the array with values to be set;

      (b) "values" giving an array of values to be set in the array.

   2. Behaviour:

      (a) If the given length does not match the ArrayField length, it throws a InvalidArraySize as per InvalidArraySize.h in [SMP_FILES];

      (b) If any of the given values simple type kind does not match the array item simple type kind, then it throws an InvalidFieldValue as per InvalidFieldValue.h in [SMP_FILES].

      (c) If any of the given values simple type kind does match the array item simple type kind, then it stores the given values into the corresponding array item values.

5.2.11.5 IField

a. All SMP fields shall implement the IField interface as per IField.h in [SMP_FILES].

b. The IField GetView method shall return the View Kind for the field.

   NOTE See Table 4-2 for specification of View Kind.

c. The IField IsState method shall return true if the field State property is true, false otherwise.

d. The IField IsInput method shall return true if the field is an Input field, false otherwise.

e. The IField IsOutput method shall return true if the field is an Output field, false otherwise.

f. The IField GetType shall return the associated field type or nullptr if the field type has not been published in the Type Registry.

5.2.11.6 IForcibleField

a. All SMP simple fields which allow forcing of the field value shall implement the IForcibleField interface as per IForcibleField.h in [SMP_FILES].

b. The IForcibleField Force method shall force the field value so that it does not change until Unforce is called with the following argument and behaviour:

   1. Argument:

      (a) "value" giving the forced value to be returned by GetValue until Unforce is called.

2. Behaviour:
   (a) If the given value simple kind does not match the field simple type kind, then it throws an InvalidFieldValue as per InvalidFieldValue.h in [SMP_FILES];
   (b) If the given value simple kind does match the field simple type kind, then it stores the given value as the value to return by GetValue.

   NOTE The handling of the forced field value within a model is undefined.

c. The IForcibleField Unforce method shall remove the forcing or freezing condition on the field so that GetValue called on the field returns the current field value.

   NOTE The handling of the forced field value within a model is undefined.

d. The IForcibleField Freeze method shall force the field to its current field value so that it no longer changes until Unforce is called.

e. The IForcibleField IsForced shall return true if field is forced or freezed, otherwise it returns false.

5.2.11.7 IDataflowField

a. All SMP fields which support actively pushing their values to connected fields shall implement the IDataflowField interface as per IDataflowField.h in [SMP_FILES].

   NOTE Dataflow connections are allowed for array items, structure fields and any sub-field of complex type fields.

b. The IDataflowField Connect method shall connect the field to an input field to create a dataflow connection between the two fields giving the following argument and behaviour:
1. Argument:
   (a) “target” giving the input field this output data flow field is connected to.
2. Behaviour:
   (a) If the target is already connected to this output field, it throws a FieldAlreadyConnected exception as per FieldAlreadyConnected.h in [SMP_FILES];
   (b) If Connect is called several times for an output field, it connect the output field to a list of input fields allowing the same output to push values to several input fields;
   (c) If the input and output field have the same type UUID, then the connection is considered to be strict compatible and it connects successfully;
   (d) If the input and output field are of semantically equivalent types as per Table 5-3, then the connection is considered to be of equivalent types and it connects successfully;

(e) If the input and output are of non-equivalent and non-strict compatible types, it throws an InvalidTarget exception as per InvalidTarget.h in [SMP_FILES];

(f) If connection is successful, it invokes the Push methods immediately, triggering an update of the connected input field with the current value of the output field.

NOTE 1 The specification of semantically equivalent type ensures that no information can be lost in transfer of data from output to input.

NOTE 2 The input field is a passive part of the transfer since the output is pushing the values to the input.

NOTE 3 The input field can be connected to several output fields.

NOTE 4 The call of the Push allows to synchronise the Input value with the Output value immediately after the connection is established.

NOTE 5 For arrays and structs, each array and struct element can implement IDataFlowField. In this case, each element can be connected with its own Connect call.

c. The IDataflowField Push method shall push the field value to all connected input fields.

NOTE 1 This is also called propagation of the value to all the connected consumer models.

NOTE 2 Since the responsibility of calling the Push operation is delegated to the component owning the field, the propagation happens “automatic” as seen from the viewpoint of the simulation environment, hence this interaction method is called “automatic data propagation”.

Table 5-3: Semantically equivalent types for connections

Type	Semantically equivalent types
Char8	Char8
String	String of same length
Bool	Bool
Signed integers	Signed integer with same size
Unsigned integers	Unsigned integer with same size
Float	Float with same size
Array	Array with same length and each element are of semantically equivalent types.

Type	Semantically equivalent types
Struct	Struct with: • identical number of elements • same order of elements • each element is of semantically equivalent types
Duration	Duration
DateTime	DateTime
Enumeration	Same enumeration type definition

5.2.12 Requirements on utilization of Simulation Environments interfaces by components

5.2.12.1 ILogger interface utilization

a. LogMessageKind type as per Services/LogMessageKind.h in [SMP_FILES] shall be used to store the Log Message Kind as returned from the results of ILogger::QueryLogMessageKind

NOTE The LogMessageKind returned is guaranteed to be always the same, even after a simulation state save or restore.

b. All SMP models and services shall use the predefined LogMessageKinds as defined in Table 5-4 for messages of message type Information, Event, Warning, Error or Debug .

5.2.12.2 IPublication interface

a. All Arrays published as a single array via the IPublication PublishArray method shall be without any padding.

NOTE This implies that array element with index i (0-based) is assumed to be stored at address of index 0 + i*sizeof(primitiveType).

b. When publishing arrays via the IPublication PublishArray method that require each element to be published individually, the following steps shall be followed:

1. Call the IPublication PublishArray method giving the following arguments:
   (a) Name of array
   (b) Description of Array
2. The PublishArray method returns a pointer to a IPublication interface
3. Use the returned IPublication interface to publish each of the elements of the array.

NOTE In case of a multi-dimensional array, step 1-3 in 5.2.12.2b can be repeated iteratively.

c. When publishing a structure via the IPublication PublishStructure method, the following steps shall be followed:

   1. Call the IPublication PublishStructure method giving the following arguments:

      (a) Name of Structure

      (b) Description of Structure

   2. The PublishStructure method returns a pointer to a IPublication interface

   3. Use the returned IPublication interface to publish each of the elements of the structure.

      NOTE In case of nested structures, steps 1-3 above can be repeated iteratively.

d. The IPublication PublishOperation method shall allow publishing an operation as per following procedure:

   1. Call the IPublication PublishOperation method giving the following arguments:

      (a) Name of Operation

      (b) Description of Operation

      (c) Its view state

   2. The IPublication PublishOperation method returns a pointer to a IPublicationOperation interface

   3. Use the returned IPublicationOperation interface to publish each of the parameters and the return value of the operation.

      NOTE See clause 5.3.9.2 for specification of the IPublicationOperation interface

5.2.12.3 ISimulator interface

a. All user defined services shall be added to the simulation using the ISimulator AddService method.

5.3 Simulation Environment interfaces

5.3.1 Logger (ILogger interface)

a. The Simulation Environment shall provide a component implementing the ILogger interfaces as per Services/ILogger.h in [SMP_FILES].

b. The component implementing the ILogger interface shall maintain a list mapping the defined Log Message Kinds names and IDs, including and eventually extending Table 5-4.

c. The ILogger QueryLogMessageKind method shall translate from the name of the message kind to the identifier of the message kind, with the following argument and behaviour:

1. Argument:
   (a) "messageKindName" giving a case sensitive string containing the name of the log message kind.
2. Behaviour:
   (a) If the given name matches one of the predefined LogMessageKind as specified in Table 5-4, it returns the corresponding LogMessageKind ID as per Table 5-4;
   (b) If the given name does not match any LogMessageKind in Table 5-4 nor any of the log message kinds in the maintained mapping, it returns a new LogMessageKind ID as a unique identifier matching the given name;
   (c) If the given name does not match any LogMessageKind in Table 5-4 but it matches one of the entries in the maintained mapping of log message kinds, it returns the corresponding LogMessageKind ID from the mapping.

Table 5-4: Default Log Message Kinds

Name	ID	Description
Debug	4	To be used for messages that can help during investigations of anomalous behaviours, but that regular users in nominal situations are not interested in seeing.
Error	3	To be used for error messages that the simulation or the model developer thinks are to be conveyed to the user when anomalous situations happen, that almost surely can lead to an anomalous simulation.
Warning	2	To be used for messages that the simulation or the model developer thinks are to be conveyed to the user when anomalous situations happen, that deserves users' attention, but that non necessarily lead to an anomalous simulation.
Event	1	To be used for log messages that the simulation or the model developer thinks are to be conveyed to the user upon certain events (the definition of 'event' is open and simulation or model developer driven).
Information	0	The message contains general information.

d. The list mapping the defined log message kinds to the defined LogMessageKind IDs shall be part of persisted data and saved/restored to/from breakpoints.

e. The list of log message kinds mapping shall be restored upon breakpoint restoring.

NOTE This implies that the list of log message kinds and associated names are part of the breakpoints and that models can store log message kinds they need and not continuously ask which LogMessageKind corresponds to a given LogMessageKind name. This leads to more efficient implementations.

f. The ILogger Log method shall log a message, with the following arguments:

1. "sender" giving the originator of the message;
2. "message" giving the text to be logged;
3. "kind" giving the registered log message kind for this message as returned from ILogger::QueryLogMessageKind method.
   (a) If the LogMessageKind ID was not previously registered by using ILogger::QueryLogMessageKind, then it registers the passed LogMessageKind with text set to the passed number, followed by the sender's Name and the string "undefined log message kind".

NOTE This implicit registration of a new LogMessageKind ID allows to quickly identify models in a simulation that are logging using custom unregistered LogMessageKinds.

5.3.2 Time Keeper (ITimeKeeper)

a. The simulation environment shall provide a component implementing the ITimeKeeper interface as per ITimeKeeper.h in [SMP_FILES].

NOTE 1 The ITimeKeeper gives access to the Time Keeper Service.

NOTE 2 The ITimeKeeper is used to maintain all the simulation times.

b. The ITimeKeeper SetEpochTime method shall set the simulation Epoch Time, with the following argument and behaviour:

1. Argument:
   (a) "epochTime" giving the new epoch time;
2. Behaviour:
   (a) After setting the EpochTime, it emits a SMP_EpochTimeChanged global SMP event.

NOTE 1 See Table 5-5 for details on EpochTimeChanged global Event.

NOTE 2 This method changes the offset between the Simulation time and the Epoch time.

c. The ITimeKeeper SetMissionStartTime method shall set a new start time for Mission time, with the following argument and behaviour:

1. Argument:
   (a) "missionStart" giving the new Epoch time for which the Mission time is zero.
2. Behaviour:
   (a) After changing the MissionStartTime, it emits the SMP_MissionTimeChanged global SMP Event.

NOTE This method changes the offset between the Epoch time and the Mission time.

d. The ITimeKeeper SetMissionTime method shall set the Mission time, with the following argument and behaviour:

1. Argument:
   (a) "MissionTime" giving the new Mission time at the current Epoch time.
2. Behaviour:
   (a) After changing the MissionTime, it emits the SMP_MissionTimeChanged global SMP Event.

NOTE This method changes the offset between the Epoch time and the Mission time.

e. The ITimeKeeper SetSimulationTime method shall advance the Simulation time, the following argument and behaviour:

1. Argument:
   (a) "SimulationTime" giving the new simulation time
2. Behaviour:
   (a) If SetSimulationTime method is called outside a PreSimTimeChange as per Table 5-5, then the method returns without updating the Simulation Time;
   (b) If the given simulation time is less than the current simulation time, it throws an InvalidSimulationTime as per Services/InvalidSimulationTime.h in [SMP_FILES] and the Simulation Time is not updated;
   (c) If the new simulation time is larger than the time of the next event on the scheduler, it throws an InvalidSimulationTime as per Services/InvalidSimulationTime.h in [SMP_FILES] and the Simulation Time is not updated;
   (d) If SetSimulationTime method is called inside a PreSimTimeChange event as per Table 5-5, then the simulation time is updated to the given simulation time.

NOTE 1 SetSimulationTime method should only called during a PreSimTimeChange global event as per Table 5-5.

NOTE 2 SetSimulationTime method does not result in emissions of PreSimTimeChange and PostSimTimeChange global events as per Table 5-5.

f. When the Time Keeper updates the Simulation time in response to the Scheduler executing a new event, the update shall be performed as per the following procedure:

1. First the PreSimTimeChange event is emitted;
2. If applicable, the simulation environment performs any activities related to maintain synchronization with Zulu time;

3. Then the Simulation time is changed to the time of the Event that is about to be executed;
4. Finally, the PostSimTimeChange event is emitted.

NOTE 1 Depending on the timing constraints of the simulation, the Simulation Environment may perform actions (like delays) to keep the desired correlation between simulation time and Zulu time after the PreSimTimeChange event and the update of the Simulation Time for next event. How this synchronization is performed is outside the scope of this standard.

NOTE 2 This method only sets the Simulation time between the current time and the time that is about to be set by the procedure described above.

g. The ITimeKeeper GetSimulationTime method shall return the Simulation time.

h. The ITimeKeeper GetEpochTime method shall return the current Epoch time.

i. The ITimeKeeper GetMissionTime method shall return the Mission time.

j. The TimeKeeper GetMissionStartTime method shall return the Mission Start Time.

k. The ITimeKeeper GetZuluTime method shall return the Zulu time.

5.3.3 Scheduler (IScheduler)

a. The simulation environment shall provide a Scheduler implementing the IScheduler in Services/IScheduler.h in [SMP_FILES].

b. The Scheduler shall allow Events to be added to the scheduler with a repeat count with the following behaviour:
1. An Event with repeat=0 is non-cyclic and executes only once;
2. An Event with repeat=0 is removed automatically after its triggering;
3. An Event with repeat>0 is cyclic, and repeats ‘repeat’ times;
4. An Event with repeat>0 is removed automatically after it has been triggered ‘repeat+1’ times;
5. An Event with repeat<0 is cyclic forever;
6. An Event with repeat<0 is never removed from the scheduler unless explicitly requested using the RemoveEvent() method.

c. The Scheduler shall allow to specify the cycle time between each call for cyclic Events with the following behaviour:
1. For non-cyclic Events, the cycle time parameter is stored, but not used;
2. For cyclic Events, the cycle time is a positive duration;

3. For cyclic Events, an InvalidCylceTime exception as per Services/InvalidCycleTime.h in [SMP_FILES] is thrown if the cycle time is negative or zero.

   NOTE The cycle time can become relevant if a subsequent call to SetEventCount is received before the Event is removed from the scheduler.

d. Events added to the scheduler by AddSimulationTimeEvent, AddMissionTimeEvent, AddEpochTimeEvent and AddZuluTimeEvent shall be executed according to a "first posted, first executed" strategy where the posting order of Events are determined based on the order of the Add call.

   NOTE This implies that the posting order is not affected by a change in Epoch time or Mission Time.

e. The IScheduler AddSimulationTimeEvent method shall add an Event to the scheduler, with the following arguments and behaviour:

   1. Arguments:

      (a) "entryPoint" giving the Entry Point to be called when the Event is executed;

      (b) "simulationTime" giving the relative time from now until the first call of the Entry Point;

      (c) "cycleTime" giving the cycle time of the Event as specified in 5.3.3c;

      (d) "repeat" giving the Event repetition count as specified in 5.3.3b.

   2. Behaviour:

      (a) If the Simulation Time is less than zero, it throws an InvalidEventTime exception as per Services/InvalidEventTime.h in [SMP_FILES], the Event is not added to the scheduler and never executed.

      (b) If Repeat is not zero and CycleTime is not positive, it throws an InvalidCycleTme exception as per Services/InvalidCycleTime.h in [SMP_FILES], the Event is not added to the scheduler and never executed;

      (c) After adding the new Event to the scheduler, it returns the EventId as per Services/EventId.h in [SMP_FILES] identifying the added Event.

   NOTE The execution order follows the general priority rules given in requirement 5.3.3d.

f. The IScheduler AddMissionTimeEvent method shall add an Event to the scheduler with the following arguments and behaviour:

   1. Arguments:

      (a) "entryPoint" giving the Entry Point to be called when the Event is executed;

(b) "missionTime" giving the mission time of the first call of the Entry Point;

(c) "cycleTime" giving the cycle time of the Event as specified in 5.3.3c;

(d) "repeat" giving the Event repetition count as specified in 5.3.3b.

2. Behaviour:

(a) If the Mission Time is less than the current mission time, it throws an InvalidEventTime exception as per Servives/InvalidEventTime.h in [SMP_FILES], the Event is not added to the scheduler and never executed;

(b) If Repeat is not zero and CycleTime is not positive, it throws an InvalidCycleTime exception as per Services/InvalidCycleTime.h in [SMP_FILES], the Event is not added to the scheduler and never executed;

(c) After adding the new Event to the scheduler, it returns the EventId as per Services/EventId.h in [SMP_FILES] identifying the added Event.

NOTE The execution order follows the general priority rules given in requirement 5.3.3d.

g. The IScheduler AddEpochTimeEvent method shall add an Event to the scheduler, with the following arguments and behaviour:

1. Arguments:

(a) "entryPoint" giving the Entry Point to be called when the Event is executed;

(b) "epochTime" giving the epoch time of the first call of the Entry Point;

(c) "cycleTime" giving the cycle time of the Event as specified in 5.3.3c;

(d) "repeat" giving the Event repetition count as specified in 5.3.3b.

2. Behaviour:

(a) If the Epoch Time is less than the current epoch time it throws an InvalidEventTime exception as per Services/InvalidEventTime.h in [SMP_FILES], the Event is not added to the scheduler and never executed;

(b) If Repeat is not zero and CycleTime is not positive, it throws an InvalidCycleTime exception as per Services/InvalidCycleTime.h in [SMP_FILES], the Event is not added to the scheduler and never executed;

(c) After adding the new Event to the scheduler, it returns the EventId as per Services/EventId.h in [SMP_FILES] identifying the added Event.

NOTE The execution order follows the general priority rules given in requirement 5.3.3d.

h. The IScheduler AddZuluTimeEvent method shall add an Event to the scheduler, with the following arguments and behaviour:

1. Arguments:

(a) "entryPoint" giving the Entry Point to be called when the Event is executed;

(b) "zuluTime" giving the Zulu time of the first call of the Entry Point;

(c) "cycleTime" giving the cycle time of the Event as specified in 5.3.3c;

(d) "repeat" giving the Event repetition count as specified in 5.3.3b.

2. Behaviour:

(a) If the given Zulu Time is less than the current Zulu time, it throws an InvalidEventTime exception as per Services/InvalidEventTime.h in [SMP_FILES], the Event is not added to the scheduler and never executed;

(b) If Repeat is not zero and CycleTime is not positive, it throws an InvalidCycleTime exception as per Services/InvalidCycleTime.h in [SMP_FILES], the Event is not added to the scheduler and never executed;

(c) After adding the new Event to the scheduler, it returns the EventId as per Services/EventId.h in [SMP_FILES] identifying the added Event.

NOTE The execution order follows the general priority rules given in requirement 5.3.3d.

i. The IScheduler AddImmediateEvent method shall add an immediate simulation time event to the scheduler with the current simulation time as execution time returning an EventId as per Services/EventId.h in [SMP_FILES], with the following argument and behaviour:

1. Argument:

(a) "entryPoint" giving the Entry Point to be called when the Event is executed.

2. Behaviour:

(a) The scheduled event is inserted at the front of the list of events scheduled for the current simulation time making it the next event to be executed;

(b) After adding the new Event to the scheduler, it returns the EventId identifying the added Event.

NOTE 1 Calls to AddImmediateEvent differs to calls to AddSimulationTimeEvent method with repeat=0, cycleTime=0 and simulationTime=0 since the event is scheduled at the font instead of the end of the list of scheduled events for the current simulation time.

NOTE 2 It cannot be assumed that Events added via AddImmediateEvent are the next Event executed, as other Events can be scheduled with AddImmediateEvent prior to its execution, hence executed first.

NOTE 3 To execute an entry point immediately without going through the scheduler, the Execute() method of the EntryPoint can be called directly.

j. The EventId returned when adding an event shall be unique throughout the entire duration of the simulation implying that EventIds cannot be reused after the Event has been executed.

NOTE The EventId must only be unique within the Scheduler context; the Event Manager service uses the same EventId type, but uniqueness across services is not required.

k. The IScheduler RemoveEvent method shall remove an already scheduled Event from the Scheduler, with the following argument and behaviour:

1. Argument:
   (a) "eventId" giving the unique identifier of the Event.
2. Behaviour:
   (a) If the given EventId does not identify an even currently in the Scheduler, it throws an InvalidEventId exception as per InvalidEventId.h in [SMP_FILES];
   (b) If the EventId is identical to the current executing Event in the schedule, then the call is functionally equivalent to setting the repeat count to 0 via the SetEventCount.

NOTE Setting the repeat count to 0 implies that the Event is removed from the scheduler immediately after it is executed.

l. The IScheduler SetEventSimulationTime method shall update the Simulation time of the next execution of an Event with the following arguments and behaviour:

1. Arguments:
   (a) "eventId" giving the unique identifier of the Event;
   (b) "simulationTime" giving the relative time from now until the next execution of the Event.
2. Behaviour:
   (a) If the Simulation Time is negative, the Event is never executed but instead removed immediately from the scheduler;
   (b) If the given EventId is not currently on the scheduler, it throws an InvalidEventId exception as per InvalidEventId.h in [SMP_FILES];
   (c) If the Event identified by the given EventId is not scheduled on Simulation time, it throws an InvalidEventId exception as per InvalidEventId.h in [SMP_FILES];

(d) When the Simulation time of the next execution of an Event is updated, it takes effect on all future repeats of this Event as per the remaining "repeat" count and respecting the given cycle-time between each repeat.

NOTE Events scheduled with AddImmediateEvent are also considered to be scheduled based on Simulation Time.

m. The IScheduler SetEventMissionTime method shall update the Mission time of the next execution of an Event with the following arguments and behaviour:

1. Arguments:

(a) "eventId" giving the unique identifier of the Event;

(b) "missionTime" giving the time of the next execution of the Event.

2. Behaviour:

(a) If the given EventId is not currently on the scheduler, it throws an InvalidEventId exception as per InvalidEventId.h in [SMP_FILES];

(b) If the Event identified by the given EventId is not scheduled on Mission time, it throws an InvalidEventId exception as per InvalidEventId.h in [SMP_FILES];

(c) If the mission time is before the current mission time, the Event is never executed but instead removed immediately from the scheduler;

(d) When the Mission time of the next execution of an Event is updated, it takes effect on all future repeats of this Event as per the remaining "repeat" count and respecting the given cycle-time between each repeat.

n. The IScheduler SetEventEpochTime method shall update the Epoch time of the next execution of an Event, with the following arguments and behaviour:

1. Arguments:

(a) "eventId" giving the unique identifier of the Event;

(b) "epochTime" giving the time of the next execution of the Event.

2. Behaviour:

(a) If the given EventId is not currently on the scheduler, it throws an InvalidEventId exception as per Services/InvalidEventId.h in [SMP_FILES];

(b) If the Event identified by the given EventId is not scheduled on Epoch time, it throws an InvalidEventId exception as per Services/InvalidEventId.h in [SMP_FILES];

(c) If the epoch time is before the current epoch time, the Event is never executed but instead removed immediately from the scheduler;

(d) When the Epoch time of the next execution of an Event is updated, it takes effect on all future repeats of this Event as per the remaining "repeat" count and respecting the given cycle-time between each repeat.

o. The IScheduler SetEventZuluTime method shall update the Zulu time of the next execution of an Event, with the following arguments and behaviour:

1. Arguments:
   (a) "eventId" giving the unique identifier of the Event;
   (b) "zuluTime" giving the time of the next execution of the Event.
2. Behaviour:
   (a) If the given EventId is not currently on the scheduler, it throws an InvalidEventId exception as per Services/InvalidEventId.h in [SMP_FILES];
   (b) If the Event identified by the given EventId is not scheduled on Zulu time, it throws an InvalidEventId exception as per Services/InvalidEventId.h in [SMP_FILES];
   (c) If the Zulu time is before the current Zulu time, the Event is never executed but instead removed immediately from the scheduler;
   (d) When the Zulu time of the next execution of an Event is updated, it takes effect on all future repeats of this Event as per the remaining "repeat" count and respecting the given cycle-time between each repeat.

p. The IScheduler SetEventCycleTime method shall allow to update the cycle time of an already scheduled Event, with the following arguments and behaviour:

1. Arguments:
   (a) "eventId" giving the unique identifier of the Event;
   (b) "cycleTime" giving the new cycle time of the Event as specified in 5.3.3c;
2. Behaviour:
   (a) If the given EventId is not currently on the scheduler, it throws an InvalidEventId exception as per Services/InvalidEventId.h in [SMP_FILES];
   (b) If the Repeat count of the Event is not zero and CycleTime is not positive, it throws an InvalidCycleTime exception as per Services/InvalidCycleTime.h in [SMP_FILES] and the CycleTime is not updated.

NOTE The CycleTime can be set also for immediate events and events with repeat count equal to 0, as the repeat can be updated with SetEventCount afterwards.

q. The IScheduler SetEventCount method shall allow to update the repeat count of an Event already scheduled with the following arguments and behaviour:

1. Arguments:
   (a) "eventId" as a unique identifier of the Event;
   (b) "count" giving the number of the Event repetitions as specified in 5.3.3b.
2. Behaviour:
   (a) If the given EventId that is not currently on the scheduler, it throws an InvalidEventId exception as per Services/InvalidEventId.h in [SMP_FILES];
   (b) If Count is not zero and CycleTime of the Event is zero, it throws an InvalidCycleTime exception as per Services/InvalidCycleTime.h in [SMP_FILES] and the Count is not updated;
   (c) If the given Count is greater than 0 and the given EventId is identical to the one currently executing, then the scheduler executes the Event for the given Count, excluding the current execution;
   (d) If the given Count is 0, the Event is removed immediately after its execution is finished.

r. The IScheduler GetCurrentEventId method shall return an EventId as per Services/EventId.h in [SMP_FILES], with the following behaviour:

1. If an Event is currently executing, it returns the EventId of the currently executing Event;
2. If no scheduled Event is currently executing, it returns -1.

NOTE A scheduled Event may not be executing if the GetCurrentEventId is called as part of the SMP global events (See clause 5.3.4)

s. The IScheduler GetNextScheduledEventTime method shall return the Simulation Time of the execution of the next scheduled Simulation Time, Epoch Time or Mission Time Event.

NOTE 1 Events scheduled in Zulu Time are not considered, as these Events do not have a fixed defined Simulation Time.

NOTE 2 In case of Zulu Events executed, other Events may schedule Events prior to the time returned, hence the Scheduler does not guarantee that no other Events may be executed prior to the time returned from GetNextScheduledEvent().

t. The complete state of the Scheduler, with the exception of Events scheduled using ZuluTime, shall be part of persisted data and saved/restored to/from breakpoints.

u. When the SMP_EpochTimeChanged global SMP event is emitted, the events already scheduled with Epoch time shall behave as follows:

(a) Non-cyclic events with Epoch Time equal to or in the future of the new Epoch Time, are executed according to the Epoch Time they were originally scheduled;

(b) Non-cyclic events with Epoch Time prior the new Epoch Time, are removed from the scheduler and not executed;

(c) For Cyclic Events, any repeat that falls prior the new Epoch Time is not executed and any positive repeat count is reduced according to the number of skipped executions;

(d) For Cyclic Events, any repeat that are equal or after the new Epoch Time is executed according to the original Epoch Time of the repeats.

v. When the SMP_MissionTimeChanged global SMP event is emitted, the events already scheduled with Mission time shall behave as follows:

(a) Non-cyclic events with Mission time equal to or in the future of the new Mission Time, are executed according to the Mission Time they were originally scheduled;

(b) Non-cyclic events with Mission time prior the new Mission Time, are removed from the scheduler and not executed;

(c) For Cyclic Events, any repeat that falls prior the new Epoch Time is not executed and any positive repeat count is reduced according to the number of skipped executions;

(d) For Cyclic Events, any repeat that are equal or after the new Mission Time is executed according to the original Mission Time of the repeats.

w. When the simulator is in Standby state, the scheduler shall behave as follows:

1. Events scheduled on simulation time including immediate Events, epoch and mission time are not processed;

2. Events scheduled on Zulu time are executed.

5.3.4 Event Manager (IEventManager)

a. The simulation environment shall provide an Event Manager implementing the IEventManager interface as Services/IEventManager.h in [SMP_FILES].

b. The IEventManager QueryEventId method shall return the Event identifier for an Event, with the following argument and behaviour:

1. Argument:

(a) “eventName” giving the name of the Event.

2. Behaviour:

(a) If called with an empty name, it throws an InvalidEventName exception as per Services/InvalidEventName.h in [SMP_FILES];

(b) If called with the name of one of the pre-defined Event types as in Table 5-5 , it returns the corresponding EventId as in Table 5-5;

(c) If called with a non-empty event name different from all pre-defined event types as in Table 5-5, it returns an event identifier different from all pre-defined event identifiers in Table 5-5;

(d) If called with the same name again in the context of a restored simulation, it returns always the same event identifier.

NOTE This implies that the EventManager maintains a global list of events that is persisted in the breakpoint and restored when needed.

c. The Event Manager shall maintain a list of pairs of unique event identifiers and entry points.

NOTE The event identifier must only be unique within the Event Manager context; the Scheduler service uses the same EventId type, but uniqueness across services is not required.

d. The Event Manager shall initialise the list of pairs to be empty at creation time.

e. The IEventManager Subscribe method shall allow to subscribe an entry point to a global event identifier, with the following arguments and behaviour:

1. Arguments:

(a) “event” giving the ID of the event to be subscribed;

(b) “entryPoint” giving a pointer to the entry point to be called when the event is emitted.

2. Behaviour:

(a) If called with a pair of event identifier and entry point that is not currently in the internal list, it adds this pair to the internal list;

(b) If called with a pair of event identifier and entry point that is already in the internal list, it throws an EntryPointAlreadySubscribed exception as per Services/EntryPointAlreadySubscribed.h in [SMP_FILES];

(c) If called with an event ID that does not exist, it throws an InvalidEventId exception as per Services/InvalidEventId.h in [SMP_FILES].

f. The IEventManager Unsubscribe method shall remove a pair from the list, with the following arguments and behaviour:

1. Arguments:

(a) “event” giving the ID of the event to be unsubscribed;

(b) “entryPoint” giving a pointer to the entry point to be unsubscribed.

2. Behaviour:
   (a) If called with a pair of event identifier and entry point that is currently in the internal list, it removes this pair to the internal list;
   (b) If called with a pair of event identifier and entry point that is not in the internal list, it throws an EntryPointNotSubscribed exception as per Services/EntryPointNotSubscribed.h in [SMP_FILES];
   (c) If called with an invalid event id, it throws an InvalidEventId exception as per Services/InvalidEventId.h in [SMP_FILES].

g. The IEventManager Emit method shall emit a specific global event to all the subscribed entry points, with the following arguments and behaviour:
   1. Arguments:
      (a) "eventId" giving the ID of the event to be emitted;
      (b) "synchronous" giving if the event is emitted synchronous to all subscribed entry points.
   2. Behaviour:
      (a) If called with an event identifier for which pairs with entry points exist in the list, then these entry points are called;
      (b) If more than one entry point is called, then the order of the calls are not guaranteed;
      (c) If called with the synchronous flag set to false, the calls to the entry points are asynchronous, such that the call to the Emit method is not blocked from returning while waiting for calls to subscribed entry points to return;
      (d) If called with the synchronous flag set to true, the calls to the entry points are synchronous, such that the call to the Emit method is blocked from returning until the calls to all subscribed entry points return.

h. The SMP predefined global events shall only be emitted in the conditions outlined in Table 5-5 and only by the Simulation Environment.

i. The SMP predefined global events shall be emitted with the synchronous flag set as per Table 5-5.

Table 5-5: Condition for emitting predefined global events

Name	EventId	Condition for emitting	Synchronous flag
SMP_LeaveConnecting	1	When leaving the Connecting state with an automatic state transition to Initializing state	True
SMP_EnterInitialising	2	When entering the Initialising state with an automatic state transition from Connecting state, or with the Initialise() state transition.	True
SMP_LeaveInitialising	3	When leaving the Initialising state with an automatic state transition to Standby state.	True
SMP_EnterStandby	4	When entering the Standby state with: • an automatic state transition from Initialising, Storing or Restoring state, • the Hold() state transition command from Executing state.	True
SMP_LeaveStandby	5	When leaving the Standby state with: • the Run() state transition command to Executing state. • the Store() state transition command to Storing state, • the Restore() state transition command to Restoring state • the Initialise() state transition command to Initialising state	True
SMP_EnterExecuting	6	When entering the Executing state with the Run() state transition command from Standby state	True
SMP_LeaveExecuting	7	When leaving the Executing state with the Hold() state transition command to Standby state.	True
SMP_EnterStoring	8	When entering the Storing state with the Store() state transition command from Standby state	True
SMP_LeaveStoring	9	When leaving the Storing state with an automatic state transition to Standby state	True
SMP_EnterRestoring	10	When entering the Restoring state with the Restore() state transition command from Standby state	True
SMP_LeaveRestoring	11	When leaving the Restoring state with an automatic state transition to Standby state	True

Name	EventId	Condition for emitting	Synchronous flag
SMP_EnterExiting	12	When entering the Exiting state with the Exit() state transition command from Standby state	True
SMP_EnterAborting	13	When entering the Aborting state with the Abort() state transition command from any other state	True
SMP_EpochTimeChanged	14	When changing the epoch time with the SetEpochTime() method of the time keeper service	True
SMP_MissionTimeChanged	15	When changing the mission time with one of the SetMissionTime() and SetMissionStartTime() methods of the time keeper service.	True
SMP_EnterReconnecting	16	When entering the Reconnecting state with the Reconnect() state transition from Standby state	True
SMP_LeaveReconnecting	17	When leaving the Reconnecting state with an automatic state transition to Standby state.	True
SMP_PreSimTimeChange	18	When all events have been executed by the Scheduler for a specific Simulation Time, but before the TimeKeeper changes the Simulation time to the time of next event.	False
SMP_PostSimTimeChange	19	When the simulation time has been changed by the Time Keeper, but before any events have been executed by the Scheduler.	False

5.3.5 Resolver (IResolver)

a. The simulation environment shall provide a component implementing the IResolver interface as Services/IResolver.h in [SMP_FILES].

b. The IResolver ResolveAbsolute method shall return a reference to a Component, Field, Failure, Container, Reference, Event Sink, Event Source or Entry Point object in the simulation, with the following argument and behaviour:

1. Argument:
   (a) “absolutePath” giving the absolute path string of the object.
2. Behaviour:
   (a) If the “absolutePath” does not give the path to an object, it returns nullptr;
   (b) If no object with the given path can be found, it returns nullptr;

(c) If "absolutePath" resolves to an object, it returns the IObject reference to the object.

NOTE 1 To allow keeping names as short as possible, and avoid dependency on the name of the simulator itself, absolute paths contain the name of either a top level Model or Service, but not the name of the simulator, although the simulator itself is the top-level object.

NOTE 2 The specification of path string is given in clause 5.1.3.

c. The Resolver ResolveRelative method shall return a reference to an object in the simulation with the following arguments and behaviour:

1. Arguments:

(a) "relativePath" giving a path string representing the relative path to the object;

(b) "sender" giving the reference to the Component issuing the request.

2. Behaviour:

(a) If "relativePath" does not resolve to any object, it returns nullptr;

(b) If "relativePath" resolves to an object, it returns an IObject reference to the object.

NOTE The specification of path string is given in clause 5.1.3.

5.3.6 Link Registry (ILinkRegistry)

a. The simulation environment shall provide a Link Registry service implementing the ILinkRegistry interface as Services/ILinkRegistry.h in [SMP_FILES].

NOTE 1 The link registry maintains a global collection of links between components, supports adding, fetching and removing all links to a given target.

NOTE 2 The links include Interface Links, Event Links and Field Links.

b. The ILinkRegistry AddLink method shall increment the link count between two components, with the following arguments and behaviour:

1. Arguments:

(a) "source" giving the source component;

(b) "target" giving the target component.

2. Behaviour

(a) The link count between both components is incremented by one, taking note of a new link that has been created.

NOTE This method can be called several times with the same arguments, when a source component has several links to the same target component.

c. The ILinkRegistry GetLinkCount method shall return the link count between the given source and target, with the following arguments:

1. "source" giving the source component;
2. "target" giving the target component.

d. The ILinkRegistry RemoveLink method shall decrement the link count between the two components, with the following arguments and behaviour:

1. Arguments:
   (a) "source" giving the source component;
   (b) "target" giving the target component.
2. Behaviour:
   (a) If the link count between both components is positive, it is decremented by one, taking note that a link has been removed, and true is returned.
   (b) If the link count between both components is 0, false is returned.

NOTE 1 Existing links have been previously added to the service using the AddLink() method.

NOTE 2 This method can be called several times with the same arguments, when several links from the source component to the same target component are removed.

e. The ILinkRegistry GetLinkSources method shall return the collection of source components for which a link to the given target component has been added to the registry.

f. The ILinkRegistry CanRemove method shall return whether all source components linking to the given target can be asked to remove their link(s), with the following argument and behaviour:

1. Argument:
   (a) "target" giving the target component of the link.
2. Behaviour:
   (a) If all source components linking to the given target can be asked to remove their link(s), it returns true;
   (b) If at least one of the source components linking to the given target cannot be asked to remove its link(s), it returns false.

NOTE Components can be asked to remove their links if they implement the optional ILinkingComponent interface.

g. The ILinkRegistry RemoveLinks method shall call the RemoveLinks method of all source components that implement the optional ILinkingComponent interface with the following argument:

1. "target" giving the component from which all links to be removed.

5.3.7 Simulator (ISimulator)

a. The simulation environment shall provide a Simulator Object implementing the ISimulator interface as ISimulator.h in [SMP_FILES].

NOTE 1 The ISimulator gives access to the simulation environment state and state transitions.

NOTE 2 The ISimulator interface provides methods to add models and to add and retrieve simulation services.

b. The Simulator Object shall have two containers as follows:

1. One "Models" container that holds simulation models with no upper limit on the number of Models to hold;
2. One "Services" container that holds simulation services with no upper limit on the number of Services to hold.

c. The ISimulator interface shall be used to setup the simulation as per the following procedure:

1. First the Publish method is called;
2. After returning from the Publish call, the Configure method is called;
3. After returning from the Configure method, the Connect method is called;
4. After returning from the Connect method, the Initialise method is called.

d. The ISimulator Publish method shall call the Publish() method of all service and model instances in the component hierarchy that are in Created state within the simulation as per following procedure:

1. If the simulation is not in Building state, then it returns and no action is taken;
2. If Publish method is called during the execution of the global event SMP_LeavingBuilding, then it returns and no action is taken;
3. If the simulation is in Building state, it issues the global event "SMP_LeavingBuilding" via the Event Manager.
4. After returning from the SMP_LeavingBuilding global event, it changes the simulation state to Publishing state.
5. After entering Publishing state, it issues the global event "SMP_EnterPublishing" via the Event Manager.
6. After returning from the "SMP_EnterPublishing" global event, it traverses through the "Service" container of the simulator, as follows:
   (a) It calls the Publish() operation of each component in CSK_Created state;
   (b) After calling Publish() on a service, it calls Publish() immediately on all its child components recursively.

7. After completing the "Service" container, it traverses through the "Models" container of the simulator as follows:
   (a) It calls the Publish() operation of each component in CSK_Created state;
   (b) After calling Publish on a model, it calls Publish immediately on all its child components recursively.
8. After all Publish() operations have been executed, it issues the global event "SMP_LeavingPublishing" via the Event Manager;
9. After returning from the "SMP_LeavingPublishing" global event, it changes the simulation state to Building state;
10. Finally, it issues the global event "SMP_EnteringBuilding" via the Event Manager.

e. The ISimulator Configure method shall call the Configure() method of all service and model instances in the component hierarchy that are in Publishing state as per following procedure:

1. If the simulation is not in Building state, then it returns and no action is taken;
2. If Configure method is called during the execution of the global event SMP_LeavingBuilding, then it returns and no action is taken;
3. If the simulation is in Building state, it issues the global event "SMP_LeavingBuilding" via the Event Manager;
4. After returning from the SMP_LeavingBuilding global event, it changes the simulation state to "Configuring" state;
5. After entering Configuring state, it issues the global event "SMP_EnterConfiguring" via the Event Manager;
6. After returning from the "SMP_EnterConfiguring" global event, it traverses through the "Services" container of the simulator. For each component, it performs the following procedure:
   (a) If the component is still in CSK_Created state, it first calls the Publish() operation;
   (b) If the component is in CSK_Publishing state, it calls the Configure() operation;
   (c) Then it immediately performs the same operation(s) recursively on all child components of the component.
7. After configuring the Services, it traverses through the "Models" container of the simulator. For each component, it performs the following procedure:
   (a) If the component is in CSK_Created state, it first calls the Publish() operation;
   (b) If the component is in CSK_Publishing state, it calls the Configure() operation;
   (c) Then it immediately performs the same operation(s) recursively on all child components of the component.
8. After all Configure() operations have been executed, it issues the global event "SMP_LeavingConfiguring" via the Event Manager;

9. After returning from the “SMP_LeavingConfiguring” global event, it changes the simulation state to Building state;
10. Finally, it issues the global event “SMP_EnteringBuilding” via the Event Manager.

f. The ISimulator Connect method shall call the Connect() method of all service and model instances in the component hierarchy that are in Configure state as per following procedure:

1. If the simulation is not in Building state, then it returns and no action is taken;
2. If Connect method is called during the execution of the global event SMP_LeavingBuilding, then it returns and no action is taken;
3. If the simulation is in Building state, it issues the global event “SMP_LeavingBuilding” via the Event Manager;
4. After returning from the SMP_LeavingBuilding global event, it changes the simulation state to Connecting state;
5. After entering Connecting state, it issues the global event “SMP_EnterConnecting” via the Event Manager;
6. After returning from the “SMP_EnterConnecting” global event, it traverses through the "Services" container of the simulator and performs the following actions:
    (a) If the component is in CSK_Created state, it calls the Publish() operation;
    (b) If the component is in CSK_Publishing state, it calls the Configure() operation;
    (c) If the component is in CSK_Configure state, it calls the Connect() operation;
    (d) Afterwards, it performs the same operation(s) recursively on all child components of the component.
7. After connecting the services, the operation traverses through the "Models" container of the simulator performing the following actions:
    (a) If the component is in CSK_Created state, it calls the Publish() operation;
    (b) If the component is in CSK_Publishing state, it calls the Configure() operation;
    (c) If the component is in CSK_Configure state, it calls the Connect() operation;
    (d) Afterwards, it performs the same operation(s) recursively on all child components of the component.
8. After all Connect() operations have been executed, it issues the global event “SMP_LeavingConnecting” via the Event Manager;
9. After returning from the “SMP_LeavingConnecting” global event, it changes the simulation state to Initialising state;
10. After entering Initialising state, it issues the global event “SMP_EnterInitialising” via the Event Manager;

11. After returning from the "SMP_EnterInitialising" global event, it calls the initialising entry points for all models that has registered an initialising entry point via the ISimulator AddInitEntryPoint method in the order the entry points where added;
12. After executing the entry points, it removes the entry points from the list so that in case Initialise is called again, the same Initialise entry point is not called twice;
13. After calling all initialising entry points, it issues the global event "SMP_LeaveInitialising" via the Event Manager;
14. After returning from the "SMP_LeaveInitialising" global event, it changes the simulation state to Standby state;
15. Finally the global event "SMP_EnteringStandby" is issued via the Event Manager.

g. The ISimulator Initialise method shall call all initialization entry points within the simulation as per the following procedure:

1. If the simulation is not in Standby state, then it returns and no action is taken;
2. If Initialise method is called during the execution of the global event SMP_LeavingStandby, then it returns and no action is taken;
3. If the simulation is in Standby state, it issues the global event "SMP_LeavingStandby" via the Event Manager;
4. After returning from the SMP_LeavingStandby global event, the simulator state changes to Initialising state;
5. After entering Initializing state, it issues the global event "SMP_EnterInitialising" via the Event Manager;
6. After returning from the SMP_EnterInitialising global event, it executes all entry points added via the ISimulator AddInitEntryPoint() method in the order they have been added by the AddInitEntryPoint() call;
7. After executing the entry points, it removes the entry points from the list so that in case Initialise is called again, the same Initialise entry point is not called twice;
8. After all entry points has been executed, it issues the global event "SMP_LeavingInitialising" via the Event Manager;
9. After returning from the "SMP_LeavingInitialising" global event, it changes the simulation state to Standby state;
10. Finally, it issues the global event "SMP_EnteringStandby" via the Event Manager.

h. The ISimulator Run method shall change the state from Standby to Executing as per following procedure:

1. If the simulation is not in Standby state, then it returns and no action is taken;
2. If Run method is called during the execution of the global event SMP_LeavingStandby, then it returns and no action is taken;

3. If Run method is called during the execution of the global event SMP_EnterStandby, then it returns and no action is taken;
4. If the simulation is in Standby state, it issues the global event "SMP_LeavingStandby" via the Event Manager;
5. After returning from the SMP_LeavingStandby global event, it changes the simulation state to "Executing" state;
6. After entering Executing state, it issues the global event "SMP_EnterExecuting" via the Event Manager.

i. The ISimulator Hold method shall change the state from Executing to Standby with the following argument and procedure:

1. Argument:
   (a) "hardHold" giving if the Simulation is halting immediately.
2. Procedure:
   (a) If the simulation is not in Executing state, then it returns and no action is taken;
   (b) If called during the execution of the global event SMP_LeavingExecuting, then it returns and no action is taken;
   (c) If called during the execution of the global event SMP_EnterExecuting, then it returns and no action is taken;
   (d) If the simulation is in Executing state, it waits until the current executing event, if any, completes;
   (e) After the current executing event is completed and if the hardHold argument is "false", it executes all events scheduled for the current simulation time;
   (f) After all events that needs executing is completed, it issues the global event "SMP_LeavingExecuting" via the Event Manager;
   (g) After returning from the SMP_LeavingExecuting global event, it changes the simulation state to "Standby" state;
   (h) After entering Standby state, it issues the global event "SMP_EnterStandby" via the Event Manager.

NOTE 1 Halting the simulation with "hardHold" to "true" can cause the simulation to halt when some models have reached the current simulation time, but others not. This is useful for debugging purposes.

NOTE 2 Halting the simulation with "hardHold" to "false" ensures that all simulation models have executed up until a consistent simulation time. This is useful for hardware in the loop simulations.

j. The ISimulator Store method shall store a breakpoint to file, with the following argument and procedure:

   1. Argument:

      (a) "filename" giving the name including the full path of the breakpoint file to be saved.

   2. Procedure:

      (a) If the simulation is not in Standby state, then it returns and no action is taken;

      (b) If Store method is called during the execution of the global event SMP_LeavingStandby, then it returns and no action is taken;

      (c) If the simulation is in Standby state, it issues the global event "SMP_LeavingStandby" via the Event Manager;

      (d) After returning from the SMP_LeavingStandby global event, it changes the simulation state to Storing state;

      (e) After entering Storing state, it issues the global event "SMP_EnterStoring" via the Event Manager;

      (f) After returning from the "SMP_EnterStoring" event, it performs Self Persistence by calling the IPersist Store method on all simulation objects that implement the IPersist interface;

      (g) After Self Persistence is completed, it performs External Persistence by storing the simulation state in the simulation breakpoint file given by the "filename" argument;

      (h) After Store operation has been completed, it issues the global event "SMP_LeaveStoring" via the Event Manager;

      (i) After returning from the "SMP_LeaveStoring" event, it changes the simulation state to Standby state;

      (j) After entering Standby state, it issues the global event "SMP_EnterStandby" via the Event Manager.

      NOTE Self-Persistence is performed prior to External Persistence during store as it allows models to update its published data prior to storing it.

k. The ISimulator Restore method shall restore a breakpoint from file, with the following argument and procedure:

   1. Argument:

      (a) "filename" giving the name including the full path of the breakpoint file to restore.

   2. Procedure:

      (a) If the simulation is not in Standby state, then it returns and no action is taken;

      (b) If called during the execution of the global event SMP_LeavingStandby, then it returns and no action is taken;

      (c) If the simulation is in Standby state, it issues the global event "SMP_LeavingStandby" via the Event Manager;

(d) After returning from the SMP_LeavingStandby global event, the simulation state is changed to Restoring state;

(e) After entering Restoring state, it issues the global event "SMP_EnterRestoring" via the Event Manager.

(f) After returning from the "SMP_EnterRestoring" event, it performs External Persistence by restoring the simulation state from a breakpoint file given by the "filename" argument;

(g) After completing External Persistence, it performs Self persistence by calling the IPersist Restore method of all simulation objects which implement the IPersist interface;

(h) After Restore operation has been completed, it issues the global event "SMP_LeavingRestoring" via the Event Manager;

(i) After returning from the "SMP_LeavingRestoring" event, the simulation state is changed to Standby;

(j) After entering Standby state, it issues the global event "SMP_EnterStandby" via the Event Manager.

NOTE Self-Persistence is performed after to External Persistence at restore as it allows models to use its published data during the self-persistence.

l. The ISimulator Reconnect method shall reconnect the component hierarchy starting at the root component given as parameter, with the following argument and procedure:

1. Argument:

(a) "root" giving the component in the hierarchy for which the reconnect shall start from.

2. Procedure:

(a) If the simulation is not in Standby state, then the method returns and no action is taken.

(b) If Reconnect method is called during the execution of the global event SMP_LeavingStandby, then the method returns and no action is taken.

(c) If the simulation is in Standby state, the global event "SMP_LeavingStandby" is issued via the Event Manager.

(d) After returning from the SMP_LeavingStandby global event, the simulation state is changed to "Reconnecting" state.

(e) The simulation environment ensures that all models under the given Root component parameter are published, configured and connected.

(f) After Reconnect operation has been completed, the simulation state is changed to Standby.

(g) After entering Standby state, the global event "SMP_EnterStandby" is issued via the Event Manager.

m. The ISimulator Exit method shall trigger a normal termination of a simulation, as per following procedure:

1. If the simulation is not in Standby state, then it returns and no action is taken;
2. If called during the execution of the global event SMP_LeavingStandby, then it returns and no action is taken;
3. If the simulation is in Standby state, it issues the global event "SMP_LeavingStandby" via the Event Manager;
4. After returning from the SMP_LeavingStandby global event, it changes the simulation state to "Exiting" state;
5. After entering Exiting state, it issues the global event "SMP_EnterExiting" via the Event Manager;
6. The Exit method triggers a normal termination of the simulation.

n. The ISimulator Abort method shall trigger an abnormal termination of a simulation, as per following procedure:

1. When called, it issues the global event "SMP_EnterAborting" via the Event Manager;
2. After returning from the "SMP_EnterAborting" event, it changes the simulation state to Aborting state;
3. After entering Aborting state, it triggers an abnormal termination of the simulation.

NOTE This method can be called from any other state.

o. The ISimulator GetState method shall return the current simulator state as per SimulatorStateKind in SimulatorStateKind.h in [SMP_FILES].

p. The ISimulator AddInitEntryPoint method shall add entry points to be executed in the Initialising state, as per following argument and behaviour:

1. Argument:
   (a) "entryPoint" giving a pointer to the entry point interface of the entry point to be added.
2. Behaviour:
   (a) If the simulation is not in Building, Connecting or Standby state, then it returns and no action is taken;
   (b) If the simulation is in Building, Connecting or Standby state, it adds the entry point to the list of entry points to be executed once the simulation reaches Initialising state.

NOTE This allows components to subscribe to a callback during initialization phase since there are only explicit methods defined for Publish, Configure and Connect. This simplifies implementation for models that do not require initialization.

q. The ISimulator AddModel method shall add a model to the models collection of the simulator, with the following argument and behaviour:

1. Argument:
   (a) “model” giving the model to be added.
2. Behaviour:
   (a) If the Simulation is not in Standby, Building, Connecting or Initializing state, it throws an InvalidSimulatorState exception as per InvalidSimulatorState.h in [SMP_FILES];
   (b) If the name of the new model conflicts with the name of an existing model already added via AddModel, it throws a DuplicateName exception as per DuplicateName.h in [SMP_FILES];
   (c) If the name of the new model conflicts with the name of an existing service already added via AddService, it throws a DuplicateName exception as per DuplicateName.h in [SMP_FILES].

NOTE 1 The container for the models has no upper limit and thus the ContainerFull exception will never be thrown.

NOTE 2 The method will never throw the InvalidObjectType exception, as it gets a component implementing the IModel interface.

r. The ISimulator AddService method shall add a user-defined service to the services collection, with the following argument and behaviour:

1. Argument:
   (a) “service” giving the service to be added.
2. Behaviour:
   (a) If the Simulation is not in Building state, it throws an InvalidSimulatorState exception as per InvalidSimulatorState.h in [SMP_FILES];
   (b) If the name of the new service conflicts with the name of an existing model already added via AddModel, it throws a DuplicateName exception as per DuplicateName.h in [SMP_FILES];
   (c) If the name of the new service conflicts with the name of an existing service already added via AddService, it throws a DuplicateName exception as per DuplicateName.h in [SMP_FILES].

NOTE 1 The container for the services has no upper limit and thus the ContainerFull exception is never thrown.

NOTE 2 The method never throw the InvalidObjectType exception, as it gets a component implementing the IService interface.

NOTE 3 It is recommended that custom services include a project or company acronym as prefix in their name, to avoid collision of service names.

s. The ISimulator GetService method shall return the interface of a service with the following argument and behaviour:

1. Argument:
   (a) "name" giving the name of the service.
2. Behaviour:
   (a) If no service with the given name , it returns nullptr;
   (b) If a service with the given name is found, it returns a reference to that service.

t. The ISimulator GetLogger method shall return the interface of the mandatory logger service.

NOTE This is a type-safe convenience method, to avoid having to use the generic GetService() method. For the standardised services, it is recommended to use the convenience methods, which are guaranteed to return a valid reference.

u. The ISimulator GetTimeKeeper method shall return the interface to the mandatory time keeper service.

NOTE This is a type-safe convenience method, to avoid having to use the generic GetService() method. For the standardised services, it is recommended to use the convenience methods, which are guaranteed to return a valid reference.

v. The ISimulator GetScheduler method shall return the interface to the mandatory scheduler service.

NOTE This is a type-safe convenience method, to avoid having to use the generic GetService() method. For the standardised services, it is recommended to use the convenience methods, which are guaranteed to return a valid reference.

w. The ISimulator GetEventManager method shall return the interface to the mandatory event manager service.

NOTE This is a type-safe convenience method, to avoid having to use the generic GetService() method. For the standardised services, it is recommended to use the convenience methods, which are guaranteed to return a valid reference.

x. The ISimulator GetResolver method shall return the interface to the mandatory resolver service.

NOTE This is a type-safe convenience method, to avoid having to use the generic GetService() method. For the standardised services, it is recommended to use the convenience methods, which are guaranteed to return a valid reference.

y. The ISimulator GetLinkRegistry method shall return the interface to the mandatory link registry service.

NOTE This is a type-safe convenience method, to avoid having to use the generic GetService() method. For the standardised services, it is recommended to use the convenience methods, which are guaranteed to return a valid reference.

z. The ISimulator RegisterFactory method shall register a component factory, with the following argument and behaviour:

1. Argument:
   (a) “componentFactory” giving the factory to be registered.
2. Behaviour:
   (a) If another factory has been registered using the same implementation identifier already, it raises a DuplicateUuid exception as per DuplicateUuid.h in [SMP_FILES].

NOTE 1 The simulator can use this factory to create component instances of the component implementation in its CreateInstance() method.

NOTE 2 This method is typically called early in the Building state to register the available component before the hierarchy of model instances is created.

aa. The ISimulator CreateInstance method shall create an instance of a component, with the following arguments and behaviour:

1. Arguments:
   (a) “uuid” giving a unique identifier of the component implementation to create;
   (b) “name” giving the name of the new instance;
   (c) “description” giving the description of the new instance;
   (d) “parent” giving the parent object of the new instance.
2. Behaviour:
   (a) If the uuid provided does not corresponds to a registered factory, it returns nullptr;
   (b) If the name provided is not a valid object name, it raises an InvalidObjectName exception as per InvalidObjectName.h in [SMP_FILES];

(c) If the uuid provided corresponds to a registered model, and the name is a valid object name, it returns a reference to the newly created model with name, description and parent set as provided.

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

bb. The ISimulator GetFactory method shall return the factory of the component with the following argument and behaviour:

1. Argument:

(a) “uuid” giving a unique identifier of the component implementation.

2. Behaviour:

(a) If a factory has been registered with the given “uuid”, it returns a pointer to the registered factory;

(b) If no factory for the given “uuid” has been registered, it returns nullptr.

cc. The ISimulator GetFactories method shall return a collection of all registered facories as per FactoryCollection in IFactory.h in [SMP_FILES].

dd. The ISimulator GetTypeRegistry method shall return a reference to the Type Registry.

ee. The ISimulator LoadLibrary method shall load a library for a Package, with the following argument and behaviour:

1. Argument:

(a) "libraryPath" to the library to load.

2. Behaviour:

(a) If called with an invalid libraryPath, it throws an LibraryNotFound exception as per LibraryNotFound.h in [SMP_FILES];

(b) If called with an libraryPath pointing to a library without initialise function, it throws and InvalidLibrary exception as per InvalidLibrary.h in [SMP_FILES];

(c) If called with the file name of a library, it loads this library into memory and calls the dynamic "Initialise()" function of this library;

(d) If called with the file name of a library, it calls the dynamic "Finalise()" function of this library when in Exiting or Aborting state.

ff. The ISimulator GetContainers method shall return a ContainerCollection with two containers as follows:

1. One container called “Models” with all the models added via ISimulator AddModel method;

2. One container called “Services” with all the services added via the ISimulator AddService method.

gg. The ISimulator GetContainer method shall return the IContainer interface to the container, with the following argument and behaviour:

1. Argument:
   (a) "name" of the container to be returned.
2. Behaviour:
   (a) If called with "Models" as argument, it returns the container reference to the models container.
   (b) If called with "Services" as argument, it returns the container reference to the Services container.
   (c) If called with anything else than "Models" or "Services", it returns nullptr.

hh. The ISimulator GetParent shall return nullptr.

NOTE The Simulator is the root object in the simulator tree.

ii. The ISimulator GetName shall return a valid name.

5.3.8 Persistence

5.3.8.1 Storage Reader Interface (IStorageReader)

a. The simulation environment shall provide a component implementing the IStorageReader interface as per IStorageReader.h in [SMP_FILES].

NOTE 1 The IStoragerReader interface provides functionality to read data from storage.

NOTE 2 The IStoragerReader interface allows objects implementing the IPersist interface to restore their state.

b. The IStorageReader Restore method shall restore data from storage, with the following arguments and behaviour:

1. Arguments:
   (a) "address" giving the address of memory block;
   (b) "size", giving the size of the memory block.
2. Behaviour:
   (a) It reads from the breakpoint a memory block of the given size at the given address.

c. The IStorageReader GetStateVectorFileName method shall return the full name including the absolute path of the breakpoint file currently in use by the Storage Reader.

d. The IStorageReader GetStateVectorFilePath method shall return a full absolute path to the directory of the breakpoint file currently in use.

NOTE The path can be used when reading additional files that correspond to the breakpoint file read.

5.3.8.2 Storage Writer Interface (IStorageWrite)

a. The simulation environment shall provide a component implementing the IStorageWriter interface as per IStorageWriter.h in [SMP_FILES].

NOTE 1 The IStoragerWriter interface provides functionality to write data from storage.

NOTE 2 The IStoragerWriter interface allows objects implementing the IPersist interface to store their state.

b. The IStorageWriter Store method shall store data to storage by writing a memory block of data to the breakpoint file with the following arguments:

1. “address” giving the address of memory block;
2. “size” giving the size of the memory block.

c. The IStorageWriter GetStateVectorFileName method shall return the full name including the absolute path of the breakpoint file currently in use by the Storage Writer.

d. The IStorageWriter GetStateVectorFilePath method shall return a full absolute path to the directory of the breakpoint file currently in use.

NOTE The path can be used when writing additional files that correspond to the breakpoint file written.

5.3.9 Publication

5.3.9.1 IPublication

a. The simulation environment shall provide a component implementing the IPublication interface as per IPublication.h in [SMP_FILES].

NOTE The IPublication interface provides functionality to allow publishing simulation model members, including fields, properties and operations.

b. The IPublication GetTypeRegistry method shall return a reference to the Type Registry.

NOTE See clause 5.3.10 for details on the Type Registry.

c. The IPublication PublishField method shall allow publishing of a field, with the following arguments and behaviour:

1. Arguments:
   (a) “name” giving the field name;
   (b) “description” giving the field description;
   (c) “address” giving the pointer to the address where the value of the field is found supporting the following pointer types:
      (1) Char8,

(2) Bool,

(3) Int8,

(4) Int16,

(5) Int32,

(6) Int64,

(7) UInt8,

(8) UInt16,

(9) UInt32,

(10) UInt64,

(11) Float32,

(12) Float64.

(d) "view" giving the fields view attribute as per ViewKind.h in [SMP_FILES];

(e) "state" given if the field is part of the simulation state when storing or restoring or not;

(f) "input" giving if the field is an input field or not;

(g) "output" giving if the field is an output field or not.

2. Behaviour:

(a) If the name of the new field to be published is already used by another published field by the same Component, it throws DuplicateName as per DuplicateName.h in [SMP_FILES];

(b) If the name of the new field to be published is not a valid object name, it throws InvalidObjectName as per InvalidObjectName.h in [SMP_FILES].

NOTE 1 The view kind attribute is specified in Table 4-2.

NOTE 2 There is no publishing call for String8 as it relies on dynamically allocated memory areas, hence cannot be published like the other primitive types.

NOTE 3 Duration and DateTime cannot be supported in the same way, as they are not strong types (they are defined to be identical to Int64, but with a different semantic). For publication of Duration and DateTime, PublishField with Uuid is used.

d. The IPublication PublishField method shall allow publishing a field, with the following arguments and behaviour:

1. Arguments:

(a) "name" giving the field name;

(b) "description" giving the field description;

(c) "address" giving the field memory address;

(d) "uuid" giving the field type;

(e) "view" giving the fields view attribute as per ViewKind.h in [SMP_FILES];

(f) "state" given if the field is part of the simulation state when storing or restoring or not;

(g) "input" giving if the field is an input field or not;

(h) "output" giving if the field is an output field or not.

2. Behaviour:

(a) If the name of the new field to be published is already used by another published field by the same Component, it throws DuplicateName as per DuplicateName.h in [SMP_FILES];

(b) If the name of the new field to be published is not a valid object name, it throws InvalidObjectName as per InvalidObjectName.h in [SMP_FILES];

(c) If the given Uuid is not a valid Uuid of a registered type, it throws InvalidUuid as per InvalidUuid.h in [SMP_FILES].

NOTE The view kind attribute is specified in Table 4-2.

e. The IPublication PublishField method shall allow publishing a field, with the following argument and behaviour:

1. Argument:

(a) "field" giving a pointer to the field IField interface.

2. Behaviour:

(a) If the name of the new field to be published is already used by another published field by the same Component, it throws DuplicateName as per DuplicateName.h in [SMP_FILES].

NOTE All additional data defining the field is available via the operations supported by the IField interface.

f. The IPublication PublishArray method shall publish an array of simple types that can be mapped to a primitive type, with the following arguments and behaviour:

1. Arguments:

(a) "name" giving the array name;

(b) "description" giving the array description;

(c) "count" giving the size of an array;

(d) "address" giving the array memory address of the first element;

(e) "type" giving the type of each array item;

(f) "view" giving the array view attribute as per ViewKind.h in [SMP_FILES];

(g) "state" given if the array is part of the simulation state when storing or restoring or not;

(h) "input" giving if the array is an input field or not;

(i) "output" giving if the array is an output field or not.

2. Behaviour:
   - (a) If the name of the new Array to be published is already used by another published field by the same Component, it throws DuplicateName as per DuplicateName.h in [SMP_FILES];
   - (b) If the name of the new field to be published is not a valid object name, it throws InvalidObjectName as per InvalidObjectName.h in [SMP_FILES].

g. The IPublication PublishArray method shall allow to publish arrays of any type by allowing each element of the array to be published individually, with the following arguments and behaviour:

1. Arguments:
   - (a) "name" giving the array name;
   - (b) "description" giving the array description.
2. Behaviour:
   - (a) If the name of the new Array to be published already used by another published field by the same Component, it throws DuplicateName as per DuplicateName.h in [SMP_FILES];
   - (b) If the name of the new field to be published is not a valid object name, it throws InvalidObjectName as per InvalidObjectName.h in [SMP_FILES];
   - (c) A pointer to an IPublication object is returned.

     NOTE 1 The returned IPublication interface allows callers of PublishArray to publish each element of the array individually.

     NOTE 2 See clause 5.2.12.2 for details on how to public each element individually.

h. The IPublication PublishStructure method shall allow publishing a structure by allowing each child element to be published individually, with the following arguments and behaviour:

1. Arguments:
   - (a) "name" giving the struct name;
   - (b) "description" giving the struct description.
2. Behaviour:
   - (a) If the name of the new Structure to be published is already used by another published field by the same Component, it throws DuplicateName as per DuplicateName.h in [SMP_FILES];
   - (b) If the name of the new Struct to be published is not a valid object name, it throws InvalidObjectName as per InvalidObjectName.h in [SMP_FILES];
   - (c) A pointer to an IPublication object is returned.

NOTE 1 The returned IPublication interface allows callers of PublishStructure to publish each element of the struct individually.

NOTE 2 See clause 5.2.12.2 for details on how to publish each element individually.

i. The IPublication PublishOperation method shall allow publishing of an operation, with the following arguments and behaviour:

1. Arguments:
   - (a) "name" giving the operation name;
   - (b) "description" giving the operation description;
   - (c) "view" giving the visibility of the operation
2. Behaviour:
   - (a) If an Operation with the same Name is already published, it updates the "Description" and "View" of the previous publication and it returns the same IPublishOperation of the previously published Operation;
   - (b) If an Operation with the same Name is not published, it creates a new IPublishOperation instance and returns it.

NOTE 1 The returned IPublishOperation interface allows callers of PublishOperation to publish parameters and return value of the operation.

NOTE 2 See clause 5.2.12.2 for details on how to publish a complete operation including its parameters.

j. The IPublication PublishProperty method shall allow publishing a property, with the following arguments and behaviour:

1. Arguments:
   - (a) "name" giving the property name;
   - (b) "description" giving the property description.
   - (c) "uuid" giving the property type.
   - (d) "accessKind" giving the property access restrictions as per AccessKind.h in [SMP_FILES] allowing the following values:
     - (1) Read and write;
     - (2) Read only;
     - (3) Write only.
   - (e) "view" giving its view kind attribute as per ViewKind.h in [SMP_FILES].
2. Behaviour:
   - (a) If a Property with the same Name is already published, it updates the "description", "uuid", "accessKind" and "view" of the previous Property;
   - (b) If the given Uuid is not a valid Uuid of a registered type, it throws a TypeNotRegistered exception as per TypeNotRegistered.h in [SMP_FILES].

k. The IPublication GetField method shall return an interface to a field, with the following argument and behaviour:

1. Argument:

   (a) "fullName" giving the path relative to the component.

2. Behaviour:

   (a) If no field exists with the given fully qualified name, it throws an InvalidFieldName exception as per InvalidFieldName.hin [SMP_FILES];

   (b) If the field matching the given fully qualified name has a simple type, it returns an ISimpleField instance;

   (c) If the field matching the given fully qualified name is an Array Field, it returns an IArrayField instance;

   (d) If the field matching the given fully qualified name is a Structure Field, it returns an IStructureField instance.

NOTE The path relative to the component is constructed as per clause 5.1.3. Examples:

- MyStructuredField.InnerField
- MyArrayField[2]
- MyStructuredField.ArrayInnerField[2]

l. The IPublication GetFields method shall return a collection of published fields as per FieldCollection in IField.h in [SMP_FILES].

m. The IPublication GetProperties method shall return a collection of published properties as per PropertyCollection in Property.h in [SMP_FILES].

n. The IPublication GetOperations method shall return a collection of published operations as per OperationCollection in Operation.h in [SMP_FILES].

o. The IPublication CreateRequest method shall return a request object allowing dynamic invocation of a published operation with the following argument and behaviour:

1. Argument:

   (a) "operationName" giving the name of operation.

2. Behaviour:

   (a) If no operation with the given name can be found, it returns nullptr.

NOTE 1 See clause 5.2.8.2 for specification of the returned request object.

NOTE 2 When the request object is no longer needed, destroyed with a call to IPublication DeleteRequest.

p. The IPublication DeleteRequest method shall delete request object that has been created with the CreateRequest() method, with the following argument:

1. "request" giving the object to be deleted.

NOTE The request object cannot be used anymore after DeleteRequest has been called for it.

q. The IPublication Unpublish method shall release all data published earlier via the Publish operations.

NOTE This is called prior to deleting the component that has called into a specific IPublication instance

5.3.9.2 IPublishOperation

a. The simulation environment shall provide a component implementing the IPublishOperation interface as per Publication/IPublishOperation.h in [SMP_FILES].

b. The IPublishOperation PublishParameter method shall allow publishing parameters of an operation, with the following arguments and behaviour:

1. Arguments:
   (a) “name” giving the parameter name;
   (b) “description” giving the parameter description;
   (c) “uuid” giving the parameter type identifier in the Type Registry;
   (d) “direction” giving the parameter direction as per Publication/ParameterDirectionKind.h in [SMP_FILES] allowing the following values:
      (1) “In” for read only parameters that are not changed by the operation;
      (2) “Out” for write only parameters where no initial value is specified but the operation provides an output value;
      (3) “InOut” for both read and write parameters;
      (4) “Return” for the operation return value.
2. Behaviour:
   (a) If the name of the new parameter to be published is already used by another published parameter by the same Operation, it throws DuplicateName as per DuplicateName.h in [SMP_FILES];
   (b) If the given Uuid is not a valid Uuid of a registered type, it throws TypeNotRegistered as per TypeNotRegistered.h in [SMP_FILES];
   (c) If the name of the new parameter to be published is not a valid object name, it throws InvalidObjectName as per InvalidObjectName.h in [SMP_FILES].

5.3.10 Type Registry

5.3.10.1 ITypeRegistry

a. The simulation environment shall provide, via the IPublication interface, a Type Registry publication implementing the ITypeRegistry interface as Publication/ITypeRegistry.h in [SMP_FILES].

NOTE This interface defines a registration mechanism for user defined types.

b. The Type Registry shall contain all pre-defined SMP value types with their pre-defined universally unique identifiers as per ecss.smp.smpcat in [SMP_FILES].

NOTE It is not mandatory for the models to make use of the Type Registry.

c. The ITypeRegistry GetType method shall return the interface to the requested primitive type, with the following argument:

1. "type" giving a primitive type kind.

NOTE This method can be used to map primitive types to the IType interface to treat all types identically.

d. The ITypeRegistry GetType method shall return the interface to the requested type, with the following argument and behaviour:

1. Argument:
   (a) "typeUuid" giving the Uuid for which the type are returned.
2. Behaviour:
   (a) If no type with the registered Uuid are found, it returns nullptr.

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

e. The ITypeRegistry AddFloatType method shall return the interface to a new Float type, with the following arguments and behaviour:

1. Arguments:
   (a) "name" giving the name of the registered type;
   (b) "description" giving the description of the registered type;
   (c) "uuid" giving the universally unique identifier of the registered type;
   (d) "minimum" giving the minimum value for float;
   (e) "maximum" giving the maximum value for float;
   (f) "minIncluded" giving whether the minimum value is valid or not;
   (g) "maxIncluded" giving whether the maximum value is valid or not;
   (h) "unit" giving the unit of the type;
   (i) "type" giving the primitive type to use for Float type.

2. Behaviour:
   (a) If the Primitive Type given is not a Float type, it throws an InvalidPrimitiveType exception as per InvalidPrimitiveType.h in [SMP_FILES];
   (b) If another type with the same uuid already is registered, it throws a TypeAlreadyRegistered exception as per TypeAlreadyRegistered.h in [SMP_FILES].

NOTE 1 IComponent and IDynamicInvocation support fields, parameters and operations of Float types via the PTK_Float32 and PTK_Float64 primitive type, as a Float is mapped either to Float32 or Float64.

NOTE 2 In type registry, name duplication is possible as long as the uuid is unique.

f. The ITypeRegistry AddIntegerType method shall return the interface to a new Integer type, with the following arguments and behaviour:
1. Arguments:
   (a) “name” giving the name of the registered type;
   (b) “description” giving the description of the registered type;
   (c) “uuid” giving the universally unique identifier of the registered type;
   (d) “minimum” giving the minimum allowed value for integer;
   (e) “maximum” giving the maximum allowed value for integer;
   (f) “unit” giving the unit of the type;
   (g) “primitiveType” giving the primitive type to use for Integer type.
2. Behaviour:
   (a) If the Primitive Type given is not an Integer type, it throws an InvalidPrimitiveType exception as per InvalidPrimitiveType.h in [SMP_FILES];
   (b) If another type with the same uuid already is registered, it throws a TypeAlreadyRegistered exception as per TypeAlreadyRegistered.h in [SMP_FILES].

NOTE IComponent and IDynamicInvocation support fields, parameters and operations of Integer types via the PTK_Int primitive types, as an Integer is mapped to one of Int8 / Int16 / Int32 / Int64 / UInt8 / UInt16 / UInt32 / UInt64.

g. The ITypeRegistry AddEnumerationType method shall return the interface to a new Enumeration type, with the following arguments and behaviour:
1. Arguments:
   (a) “name” giving the name of the registered type;
   (b) “description” giving the description of the registered type;

(c) "uuid" giving the universally unique identifier (UUID) of the registered type;

(d) "size" giving the size of an instance of this enumeration in bytes. Valid values are 1, 2, 4 and 8.

2. Behaviour:

(a) If another type with the same uuid already is registered, it throws a TypeAlreadyRegistered exception as per TypeAlreadyRegistered.h in [SMP_FILES].

h. The ITypeRegistry AddArrayType method shall return the interface to a new Array type, with the following arguments and behaviour:

1. Arguments:

(a) "name" giving the name of the registered type;

(b) "description" giving the description of the registered type;

(c) "typeUuid" giving the universally unique identifier of the registered type;

(d) "itemTypeUuid" giving the universally unique identifier of the Type of the array items;

(e) "itemSize" giving the size of an array item in bytes, taking possible padding into account, as it can be used by the simulation environment to calculate the memory offset between array items;

(f) "arrayCount" giving the number of elements in the array;

(g) "simpleArray" giving a flag whether a field of this array type is be implemented as ISimpleArrayField or as IArrayField.

2. Behaviour:

(a) If another type with the same uuid already is registered, it throws a TypeAlreadyRegistered exception as per TypeAlreadyRegistered.h in [SMP_FILES].

i. The ITypeRegistry AddStringType method shall return the interface to a new String type, with the following arguments and behaviour:

1. Arguments:

(a) "name" giving the name of the registered type;

(b) "description" giving the description of the registered type;

(c) "uuid" giving the universally unique identifier of the registered type;

(d) "length" giving the maximum length of the string.

2. Behaviour:

(a) If another type with the same uuid already is registered, it throws a TypeAlreadyRegistered exception as per TypeAlreadyRegistered.h in [SMP_FILES].

j. The ITypeRegistry AddStructureType method shall return the interface to a new Structure type that allows adding fields, with the following arguments and behaviour:

1. Arguments:
   (a) "name" giving name of the registered type;
   (b) "description" giving description of the registered type;
   (c) "uuid" giving the universally unique identifier of the registered type.
2. Behaviour:
   (a) If another type with the same uuid already is registered, it throws a TypeAlreadyRegistered exception as per TypeAlreadyRegistered.h in [SMP_FILES].

k. The ITypeRegistry AddClassType method shall return the interface to a new Class type that allows adding fields, with the following arguments and behaviour:

1. Arguments:
   (a) "name" giving name of the registered type;
   (b) "description" giving description of the registered type;
   (c) "uuid" giving the universally unique identifier of the base class.
2. Behaviour:
   (a) If another type with the same uuid already is registered, it throws a TypeAlreadyRegistered exception as per TypeAlreadyRegistered.h in [SMP_FILES].

5.3.10.2 IType

a. The simulation environment shall provide a class implementing the IType interface as per Publication/IType.h in [SMP_FILES].

b. The IType GetPrimitiveTypeKind method shall return the primitive type kind as per PrimitiveTypes.h in [SMP_FILES] for types in the type registry as follows:

1. If the type cannot be mapped to a primitive type kind, it returns PTK_None;
2. If the type is registered as a derived type of one of the primitive types, it returns the Primitive type kind;
3. If the type is one of the primitive types themselves, it returns the corresponding primitive type kind.

NOTE 1 The primitive types are specified in Table 5-1.

NOTE 2 Types that cannot be mapped to a primitive type include:

- Arrays registered via ITypeRegistry AddArrayType;
- Structures registered via ITypeRegistry AddStructureType.

NOTE 3 Derived types include:

- Enumerations registered via ITypeRegistry AddEnumerationType;

- Strings registered via ITypeRegistry AddStringType;
- Integer types registered via the ITypeRegistry AddIntegerType;
- Float types registered via the ITypeRegistry AddFloatType.

c. The IType GetUuid method shall return the Universally Unique Identifier of the type.

d. The IType Publish method shall allow publishing a new field in a receiver, with the following arguments:

1. "receiver" giving the publishing interface to publish against;
2. "name" giving the name of instance;
3. "description" giving the description of instance;
4. "address" giving the address of instance;
5. "viewKind" giving the visibility of instance;
6. "state" giving if the instance is part of the breakpoint or not;
7. "input" giving if writing to the instance is allowed;
8. "output" giving if reading from the instance is allowed.

NOTE Using the IType Publish method is an alternative method to publish a field than using the IPublication publishing methods.

5.3.10.3 IStructureType

a. The simulation environment shall provide a class implementing the IStructureType interface as per Publication/IStructureType.h in [SMP_FILES].

b. The IStructureType AddField method shall add a field to the structure, with the following arguments:

1. "Name" giving the name of the field;
2. "Description" giving the description of the field;
3. "Uuid" giving the universally unique identifier of field Type, as a value type;
4. "offset" giving the memory offset of field relative to Structure;
5. "ViewKind" giving the visibility of instance;
6. "state" giving if the instance is part of the breakpoint or not;
7. "input" giving if writing to the instance is allowed;
8. "output" giving if reading from the instance is allowed.

5.3.10.4 IClassType

a. The simulation environment shall provide a class implementing the IClassType interface as per Publication/IClassType.h in [SMP_FILES].

5.3.10.5 IArrayType

a. The simulation environment shall provide a class implementing the IArrayType interface as per Publication/IArrayType.h in [SMP_FILES].

b. The IArrayType GetSize method shall return the number of elements in the array.

c. The IArrayType GetItemType method shall return a pointer to the type that all array items have.

5.3.10.6 IEnumerationType

a. The simulation environment shall provide a class implementing the IEnumerationType interface as per Publication/IEnumerationType.h in [SMP_FILES].

b. The IStructureType AddLiteral method shall add a literal entry to the enumeration given the following input arguments and behaviour:

1. Arguments:
   (a) "name" giving the name of the literal;
   (b) "description" giving the description of the field;
   (c) "value" giving the "value" of the literal.
2. Behaviour:
   (a) If the given Name is already added as a literal to the enumeration, it throws a DuplicateName exception as per DuplicateName.h in [SMP_FILES];
   (b) If the given Value is already added as a literal to the enumeration, it throws a DuplicateLiteral exception as per Publication/DuplicateLiteral.h in [SMP_FILES].

5.3.11 Component Factory (IFactory)

a. The simulation environment shall provide a class implementing the IFactory interface as per IFactory.h in [SMP_FILES].

b. The IFactory GetUuid method shall return the UUID of the component that will be created by this factory.

c. The IFactory CreateInstance method shall create an instance of the component with the following arguments:

1. "name" giving the name of the instance to be created;
2. "description" giving the description of the instance to be created;
3. "parent" giving a pointer to the parent object of the instance to be created.

d. The IFactory DeleteInstance method shall delete an existing component with the following argument:

1. "instance" given the IComponent interface to the component to be deleted.

e. The IFactory GetTypeName method shall return the fully qualified C++ type name of the component type.

NOTE The fully qualified type name contains all namespaces and the name of the type, separated by two colons ("::").

5.4 Meta data

5.4.1 Catalogue

5.4.1.1 File format specification

a. The Catalogue file shall be in conformance with thc Catalogue file DRD of Annex A.

5.4.1.2 Validation rules

5.4.1.2.1 General

a. All user defined catalogues shall link to the SMP catalogue in file XML/ecss.smp.smpcat in [SMP_FILES] for all standard SMP elements defined in this standard.

NOTE 1 The ecss.smp.smpcat contains the complete meta data model for all elements of [SMP_FILES] expressed in an SMP catalogue.

NOTE 2 The usage of a common standardized SMP catalogue ensures that common types and other elements have the same UUID across all platform, hence allows model integration.

b. No recursive Types shall be specified.

NOTE Models, interfaces, entry points, fields, etc… are Types in the catalogue, so these Types cannot be typed as, be derived from or use themselves at any level of their specification.

c. Types that are used in another Type shall be visible for that Type.

d. XLinks in documents shall not result in recursively linked documents.

e. The xlink:href attribute shall be a valid URI locator on the form "<Document>[#<Fragment>]", where <Document> is the linked XML file and <Fragment> is an optional named element defined in that file.

NOTE In case the named element is defined within the same file, i.e. the link is local to the file, the <Document> part of the locator can be omitted.

f. The xlink:title attribute shall always contain the Name of the referenced named element.

5.4.1.2.2 Types

a. The size of an array size shall be a positive number.

b. The PrimitiveType for a Float may only point to Float32 or Float64.

c. Float Minimum shall be less than Float Maximum if MinInclusive is false or MaxInclusive is false.

d. Float Minimum shall be less or equal to Float Maximum if MinInclusive is true and MaxInclusive is true.

e. The length of a string shall be larger or equal to zero.

f. The length of a String Value shall not exceed the size of the corresponding String type.

g. The PrimitiveType for an Integer shall point to Int8, Int16, Int32, Int64, UInt8, UInt16, UInt32 or UInt64.

h. For Integer types, the Minimum shall be less or equal to the Integer Maximum.

i. The type for an AttributeType shall point to a ValueType.

j. The Type link for an Attribute shall point to an AttributeType.

k. The default value of an AttributeType shall be not empty.

l. The Value of an Attribute shall not be empty.

5.4.1.2.3 Named Element

a. A Named Element Name shall be unique in its context.

b. A Named Element Id shall be unique in its Document.

c. A Named Element Id shall not be empty.

d. A Named Element Name shall not be an ISO/ANSI C++ keyword.

e. A Named Element Name shall only contain letters, digits, and the underscore, optionally followed by '[' and ']' enclosing a number or a string.

f. Type UUID shall be unique.

5.4.1.2.4 Container and associations

a. Container lower bound shall be a positive number or 0.

b. Container lower bound shall be less or equal to the container upper bound, if present.

c. Container upper bound shall be -1 or larger or equal to the container lower bound.

d. The Type link of an Association shall point to a Model, Interface or Value Type.

e. The Type link of a Container shall point to a Reference Type.

f. The Type link of a Reference shall point to an Interface.

g. For a Reference, the Lower limit shall be less or equal to the Upper limit.

5.4.1.2.5 Enumeration

a. Enumeration Literal Names shall be unique within an Enumeration.

b. Enumeration Literal Values shall be unique within an Enumeration.

5.4.1.2.6 Entry Point

a. Entry Point Output fields shall be output type fields.

b. Entry Point Input fields shall be input type fields.

c. Entry Point Input and Output fields shall be located in the same Model or a base model.

5.4.1.2.7 Properties

a. Property Attached Field shall have the type of Property's Type, or a type derived thereof.

b. Property Attached Field shall be located in the same Class or a base class.

c. The Type link of a Property shall point to a Value Type.

d. The Type of the AttachedField shall match the Type of the Property.

e. A Property of an Interface shall be public.

f. A Property of an Interface shall not be static.

5.4.1.2.8 References

a. Reference lower bound shall be larger than zero.

b. Reference lower bound shall be less or equal to Reference upper bound, if present.

c. Reference upper bound shall be -1, larger or equal to Reference lower bound.

5.4.1.2.9 Fields

a. The Field link shall point to a Field.

b. The Type link for a Field shall point to a ValueType.

c. The Field value shall not be empty.

d. A Field of a Structure shall be public.

e. The Type link of an Operation shall point to a ValueType.

5.4.1.2.10 Operations

a. An operation of an Interface shall be public.

b. An Operation of an Interface shall not be static.

c. The Type link of a Parameter of the Operation shall point to a Value Type.

d. A Parameter shall only have a default value if its type is Value Type.

e. The value of a parameter shall be inside the range defined for the corresponding type.

f. Each operation shall have only one parameter with the return type attribute set.

5.4.1.2.11 Constructors

a. Constructors shall not have any return parameters.

b. Constructors shall not have Const, Virtual or Static attributes.

5.4.1.2.12 Events

a. Events shall be typed by an EventType.

b. The EventArgs link for an EventType shall only point to a SimpleType.

c. The Type link of an EventSource shall point to an EventType.

d. The Type link of an EventSink shall point to an EventType.

e. An EventSource shall only be linked to an EventSink when both have the same EventType.

5.4.1.3 Requirements on utilization of Catalogue

a. The simulation Models design shall be defined via a catalogue, or a set of catalogues.

b. Catalogues shall not have circular dependencies.

c. Each Model in a simulation shall be defined in a catalogue.

d. Each user-defined Service in a simulator shall be defined in a catalogue.

e. Each Interface between components shall be defined in a catalogue.

f. Each Type used in an interface or component shall be defined in a catalogue.

g. Each Field of a model or service that is of a type defined in a catalogue shall be defined in a catalogue.

h. Each public Property of an interface, model or service shall be defined in a catalogue.

i. Each public Operation of an interface, model or service shall be defined in a catalogue.

j. Each Entry Point of a model or service shall be defined in a catalogue.

k. Each Event Source of a model or service shall be defined in a catalogue.

l. Each Event Sink of a model or service shall be defined in a catalogue.

m. Each Container of a model or service shall be defined in a catalogue.

n. Each Reference of a model or service shall be defined in a catalogue.

5.4.2 Package

5.4.2.1 File format specification

a. The Package file shall be in confromance with the Package file DRD of Annex B.

5.4.2.2 Validation rules

a. There shall be no clashes of Type names in packages.

b. For each Model implementation, a different Uuid shall be used.

5.4.3 Configuration data

5.4.3.1 File format

a. Files containing configuration data for published fields should be in conformance with the Configuration file DRD of Annex C.

NOTE The usage of the SMP Configuration file format is optional.

5.4.3.2 Validation rules

a. All path strings in configuration files shall be valid SMP path strings.

NOTE Valid SMP path strings are specified in clause 5.1.3

b. All field values set shall be valid values for the field type it refers to.

6
Implementation mapping

6.1 Catalogue to C++

6.1.1 Mapping templates

a. Syntax and expression rules used in the specification of C++ mapping templates:

1. Parts omitted to shorten the template and ease the reading are replaced by '…'.

2. Information from the catalogue to be mapped in the C++ code is specified by means of placeholders encased within dollar '$' symbols. For example, $Component.Name$ for the value of the field 'Name' of some 'Component' element referred in the context the template is applicable. In case an element belongs in a sequence with a number 'N' of occurrences, $...Element[i]...$ refers to the 'i-th' occurrence of the sequence where 'i' could take any value between '1' and 'N-1'.

3. Fully qualified names for types are specified by means of the 'TypeName($Type$)' expression. For example, for a given type 'MyType' defined within two levels of nested namespaces would refer to '::Namespace1::Namespace2::MyType'.

4. Optional code is specified encased within the square bracket '[' and ']' symbols. For example, '[static]' where the use of 'static' might be subject to some conditions. Exception is where '[...]' is used for the elements in an array as per rule a.2. above.

5. Alternative code is specified by means of the '|' separator symbol where exactly one of several options is required. For example, 'A|B|C' if either 'A', 'B' or 'C' is to be used in the code.

NOTE Table 6-1 and Table 6-2 contains the C++ declaration and defintition templates and are referred to from requirements of clause 6.1.

Table 6-1: C++ declaration templates

Template	C++ mapping
Constant	`static constexpr TypeName($Constant.Type$) $Constant.Name$ = $Constant.Value$;`
Field	`[static ][mutable ]TypeName($Field.Type$) $Field.Name$;`
Association	`[const ][static ][mutable ]TypeName($Association.Type$)[*] $Association.Name$;`
Parameter	`[const ]TypeName($Parameter.Type$)[*|&] $Parameter.Name$[ = $Parameter.Default$]`
Property Getter	`[virtual ] [static ][const ]TypeName($Property.Type$)[*|&] get_$Property.Name$()[ const][ = 0];`
Property Setter	`[virtual ][static ]void set_$Property.Name$([const ]TypeName($Property.Type$)[*|&] value)[ = 0];`
Operation	`[virtual ][static ] void|TypeName($Operation.Parameter[i].Type$)[*|&] $Operation.Name$(void|...)[ const][ = 0];`
Operator	`[virtual ][static ] void|TypeName($Operation.Parameter[i].Type$)[*|&] operator $Operation.Operator.OperatorKind$(void|...)[ const][ = 0];`
Constructor	`$Owner.Name$(void|...)[= delete];`
Entry Point	`Smp::IEntryPoint* $EntryPoint.Name$;`
Event Sink	`Smp::IEventSink* $EventSink.Name$;`
Event Source	`Smp::IEventSource* $EventSource.Name$`
Container	`Smp::IContainer* $Container.Name$;`
Reference	`Smp::IReference* $Reference.Name$;`
Uuid	`extern const Smp::Uuid Uuid_$Type.Name$;`
Global Registry	`[static] void _Register_$Type.Name$( Smp::Publication::ITypeRegistry* registry);`
Scoped Registry	`[static] void _Register( Smp::Publication::ITypeRegistry* registry);`
Enumeration	`enum class $Enumeration.Name$ : Smp::Int32 { ... };`
Literal	`$Enumeration.Literal.Name$ = $Enumeration.Literal.Value$`
Integer	`typedef $Integer.PrimitiveType$|Smp::Int32 $Integer.Name$;`
Float	`typedef $Float.PrimitiveType$|Smp::Float64 $Float.Name$;`
String	`struct $String.Name$ { Smp::Char8 internalString[$String.Length$+1]; };`
Array	`struct $Array.Name$ { TypeName($Array.ItemType$) internalArray[$Array.Size$]; };`
Structure	`struct $Structure.Name$ { ... };`

Template	**C++ mapping**
Class	class $Class.Name$
	[: public TypeName($Class.Base.Name$)] {
	...
	};
Exception	class $Exception.Name$:
	public TypeName($Exception.Base.Name$)|Smp::Exception {
	...
	};
Interface	class $Interface.Name$
	[: virtual public TypeName($Interface.Base[1].Name$),
	...,
	TypeName($Interface.Base[N].Name$)] {
	...
	};
Model	class $Model.Name$:
	[public TypeName($Model.Base.Name$),]
	[virtual public TypeName($Model.Interface[1].Name$),
	...,
	TypeName($Model.Interface[N].Name$),]
	[virtual public Smp::IEntryPointPublisher,]
	[virtual public Smp::IEventConsumer,]
	[virtual public Smp::IEventProvider,]
	[virtual public Smp::IComposite,]
	[virtual public Smp::IAggregate,]
	virtual public Smp::IModel {
	...
	};
Service	class $Service.Name$:
	[public TypeName($Service.Base.Name$),]
	[virtual public TypeName($Service.Interface[1].Name$),
	...,
	TypeName($Service.Interface[N].Name$),]
	[virtual public Smp::IEntryPointPublisher,]
	[virtual public Smp::IEventConsumer,]
	[virtual public Smp::IEventProvider,]
	virtual public Smp::IService {
	...
	};

Table 6-2: C++ definition templates

Template	C++ mapping
Uuid	`Smp::Uuid Uuid_$Type.Name$ = $Type.Uuid$;`
Simple	`TypeName($Variable.Type$) $Variable.Name$ = $Variable.Value.Value$|$Variable.Value.Literal$;`
Array	`TypeName($Variable.Type$) $Variable.Name$ = {{ $Variable.ItemValue[1].Value$|$Variable.ItemValue[1].Literal$, ..., $Variable.ItemValue[N].Value$|$Variable.ItemValue[N].Literal$ }};`
Structure	`TypeName($Variable.Type$) $Variable.Name$ = { $Variable.FieldValue[1].Value$|$Variable.FieldValue[1].Literal$, ..., $Variable.FieldValue[N].Value$|$Variable.FieldValue[N].Literal$ };`
Property Getter	`return $Property.AttachedField.Name$;`
Property Setter	`$Property.AttachedField.Name$ = value;`

6.1.2 Namespaces and files

a. All elements shall be declared within the exact same namespace as in the Catalogue.

b. Each type shall be declared in a dedicated header file as follows:

1. The hierarchy of namespaces defines the file location with one directory level per namespace level in the hierarchy;
2. The type name defines the file name.

c. Header files shall allow multiple inclusion by implementing '#include' guards.

d. Header files shall avoid circular dependencies by using forward declaration.

6.1.3 Element and Type Visibility Kind

a. Visibility kind attributes shall be mapped to ISO/ANSI C++ member access specifiers as follows:

1. If the attribute is explicitly defined, mapping is as per Table 6-3;
2. If the attribute is undefined, the default "Private" visibility kind is used with mapping as per Table 6-3.

Table 6-3: C++ mapping for the Visibility kind attribute

Visibility kind	Description	C++ mapping
Private	Local to the parent Type.	`private`
Protected	Local to the parent Type and derived Types thereof.	`protected`
Public	Global.	`public`

6.1.4 Mapping of elements

6.1.4.1 Value elements

a. Simple value elements shall be mapped to ISO/ANSI C++ variable's values as follows:

1. Syntax as per "Simple" template in Table 6-2;
2. If the element value is of EnumerationValue type, mapping is done using the Literal attribute instead of the Value one.

b. Array value elements shall be mapped to ISO/ANSI C++ variable's values as follows:

1. Syntax as per "Array" template in Table 6-2;
2. If the element items are of EnumerationValue type, mapping is done using their Literal attribute instead of the Value one.

c. Structure value elements shall be mapped to ISO/ANSI C++ variable's values as follows:

1. Syntax as per "Structure" template in Table 6-2;
2. For the element fields of EnumerationValue type, mapping is done using the Literal attribute instead of the Value one.

6.1.4.2 Constant

a. Constant elements shall be mapped to ISO/ANSI C++ member variables as per "Constant" template in Table 6-1.

b. The value of the Constant member variable shall be defined as per mapping of the Value attribute.

NOTE See clause 6.1.4.1 for details on the mapping of Value attributes.

c. The access specifier of the Constant member variable shall be defined as follows:

1. If the member variable belongs in a C++ structure, the member is public;
2. If the member variable does not belong in a C++ structure, the mapping of the Visibility attribute is used.

NOTE 1 See clause 6.1.3 for details on the mapping of Visibility attributes.

NOTE 2 The access specifier applies to Classes, Models, Services and Interfaces.

6.1.4.3 Field

a. Field elements shall be mapped to ISO/ANSI C++ member variables as per "Field" template in Table 6-1.

b. The initial value of the Field member variable shall be defined as per mapping of the Default attribute.

NOTE See clause 6.1.4.1 for details on the mapping of Value attributes.

c. The access specifier of the Field member variable shall be defined as follows:

   1. If the member variable belongs in a C++ structure, the member is public;
   2. If the member variable does not belong in a C++ structure, the mapping of the Visibility attribute is used.

      NOTE See clause 6.1.3 for details on the mapping of Visibility attributes.

d. The Static attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Field C++ mapping:

   1. If set to "true", then the C++ field includes the 'static' specifier as per "Field" template in Table 6-1;
   2. If not set, then it has no effect;
   3. If set to "false", then it has no effect.

e. The Mutable attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Field C++ mapping:

   1. If set to "true", then the C++ field includes the 'mutable' specifier as per "Field" template in Table 6-1;
   2. If not set, then it has no effect;
   3. If set to "false", then it has no effect.

6.1.4.4 Association

a. Association elements shall be mapped to ISO/ANSI C++ member variables as per "Association" template in Table 6-1;

b. The access specifier of the Association member variable shall be defined by the mapping of the Visibility attribute.

   NOTE See clause 6.1.3 for details on the mapping of Visibility attributes.

c. The ByPointer attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Association C++ mapping:

   1. If set to "true", then the C++ mapping of the type includes the '*' specifier as per "Association" template in Table 6-1;
   2. If not set, then the C++ mapping of the type includes the specifier corresponding to the type referenced in the Type attribute as per Table 6-4;
   3. If set to "false", then the C++ mapping of the type does not include the '*' specifier.

Table 6-4: C++ mapping of Association depending on ByPointer attribute

	C++ mapping			
	Native Type	Value Type	Value Reference	Reference Type
Specifier without ByPointer				*
Specifier with ByPointer="true"	*	*	*	*
Specifier with ByPointer="false"				

d. The Const attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Association C++ mapping:

1. If set to "true", then the C++ mapping includes the 'const' specifier as per "Association" template in Table 6-1;
2. If not set, then it has no effect;
3. If set to "false", then it has no effect.

e. The Static attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Association C++ mapping:

1. If set to "true", then the C++ mapping includes the 'static' specifier as per "Association" template in Table 6-1;
2. If not set, then it has no effect;
3. If set to "false", then it has no effect.

f. The Mutable attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Association C++ mapping:

1. If set to "true", then the C++ mapping includes the 'mutable' specifier as per "Association" template in Table 6-1;
2. If not set, then it has no effect;
3. If set to "false", then it has no effect.

6.1.4.5 Parameter

a. Parameter elements shall be mapped to ISO/ANSI C++ as follows:

1. If the Direction kind attribute is 'return', the parameter is the return type of a C++ member method;
2. If the Direction kind attribute is not 'return', the parameter is an argument of a C++ member method with default value given by the Default attribute;
3. Syntax is for arguments as per "Parameter" and for the return type as per "Operation" templates in Table 6-1;
4. For the C++ type specifier the mapping of the Direction kind attribute corresponding to the type referenced in the Type attribute as per Table 6-5 is used.

Table 6-5: C++ mapping for the Direction kind attribute

Direction kind	C++ mapping			
	Native Type	Value Type	Value Reference	Reference Type
in	const		const	const &
out	*	*		*
inout	*	*		*
return				*

b. The ByReference attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Parameter C++ mapping:

1. If set to "true", then the C++ mapping includes the '&' specifier as per "Parameter" template in Table 6-1, irrespectively of Table 6-5;
2. If not set, then the C++ mapping is done according to Table 6-5;
3. If set to "false", then the C++ mapping does not include the '&' specifier, irrespectively of Table 6-5.

c. The Const attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Parameter C++ mapping:

1. If set to "true", then the C++ mapping includes the 'const' specifier as per "Parameter" template in Table 6-1, irrespectively of Table 6-5;
2. If not set, then the C++ mapping is done according to Table 6-5;
3. If set to "false", then the C++ mapping does not include the 'const' specifier, irrespectively of Table 6-5.

d. The ByPointer attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Parameter C++ mapping:

1. If set to "true", then the C++ mapping includes the '*' specifier as per "Parameter" template in Table 6-1, irrespectively of Table 6-5;
2. If not set, then the C++ mapping is done according to Table 6-5;
3. If set to "false", then the C++ mapping does not include the '*' specifier, irrespectively of Table 6-5.

NOTE It is invalid to have both the ByReference attribute and the ByPointer attribute set to "true" for the same parameter.

6.1.4.6 Property

a. Property elements shall be mapped to ISO/ANSI C++ member methods as follows:

1. If the Access attribute is not defined, or it is defined with value equal to 'readWrite' or 'readOnly', a getter member method is created with syntax as per "Property Getter" template in Table 6-1;
2. If the Access attribute is not defined, or it is defined with value equal to 'readWrite' or 'writeOnly', a setter member method is created with syntax as per "Property Setter" template in Table 6-1;

b. The access specifier of the Property member methods shall be defined as follows:

1. If the Operation belongs in an Interface, the member is public;
2. If the Operation does not belong in an Interface, the mapping of the Visibility attribute is used.

   NOTE See clause 6.1.3 for details on the mapping of Visibility attributes.

c. If the AttachedField element is defined, the body of the Property getter and setter member methods shall be respectively mapped as per "Property Getter" and "Property Setter" templates in Table 6-2.

d. The ByReference attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Property C++ mapping:

1. If set to "true", then the C++ mapping of the type includes the '&' specifier;
2. If not set, or if set to "false", then the C++ mapping of the type does not include the '&' specifier.

e. The ByPointer attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Property C++ mapping:

1. If set to "true", then the C++ mapping of the type includes the '*' specifier;
2. If not set, then the C++ mapping of the type includes the specifier corresponding to the type referenced in the Type attribute as per Table 6-6;
3. If set to "false", then the C++ mapping of the type does not include the '*' specifier.

Table 6-6: C++ mapping for Property depending on ByPointer attribute

	C++ mapping			
	Native Type	**Value Type**	**Value Reference**	**Reference Type**
specifier without ByPointer				*
Specifier with ByPointer="true"	*	*	*	*
Specifier with ByPointer="false"				

f. The Static attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Property C++ mapping:

1. If set to "true", then the C++ mapping includes the 'static' specifier as per "Property Getter" and "Property Setter" template in Table 6-1;
2. If not set, then it has no effect;
3. If set to "false", then it has no effect.

g. The Virtual attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Property C++ mapping:

   1. If set to "true", then the C++ mapping includes the 'virtual' specifier as per "Property Getter" and "Property Setter" template in Table 6-1;
   2. If not set, then the C++ mapping includes the 'virtual' specifier as per "Property Getter" and "Property Setter" template in Table 6-1 if the property belongs to an Interface, Model or Service;
   3. If set to "false", then it has no effect.

h. The Abstract attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Property C++ mapping:

   1. If set to "true", then the C++ mapping includes the '=0' pure specifier as per "Property Getter" and "Property Setter" template in Table 6-1;
   2. If not set, then the C++ mapping includes the '=0' pure specifier as per "Property Getter" and "Property Setter" template in Table 6-1 if the property belongs to an Interface;
   3. If set to "false", then it has no effect.

i. The ConstGetter attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Property C++ mapping:

   1. If set to "true", then the C++ mapping includes the 'const' specifier at the end, as per "Property Getter" template in Table 6-1;
   2. If not set, then it has no effect;
   3. If set to "false", then it has no effect.

j. The Const attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Property C++ mapping:

   1. If set to "true", then the C++ mapping includes the 'const' specifier at the beginning, as per "Property Getter" and "Property Setter" templates in Table 6-1;
   2. 2. If not set, then it has no effect;
   3. 3. If set to "false", then it has no effect.

6.1.4.7 Operation

a. Operation elements shall be mapped to ISO/ANSI C++ member methods as follows:

   1. If neither the Operator nor the Constructor attribute is set, syntax is as per "Operation" template in Table 6-1.
   2. If the Operator attribute is set, syntax is as per "Operator" template in Table 6-1.
   3. If the Constructor attribute is set, syntax is as per "Constructor" template in Table 6-1.

      NOTE 1 Operator and Constructor attributes cannot be both set at the same time for a given Operation element as they are mutually exclusive.

NOTE 2 Constructor methods inherit the name from the element the Operation is member of, therefore their own Name attribute is ignored.

b. Operation elements shall have at maximum one Parameter element, or none in case the Constructor attribute is set, with Direction attribute equal to ‘return’.

c. Parameter elements belonging to the Operation element shall be mapped as follows:

1. Syntax as per mapping of Parameter elements.
2. If there is no Parameter element with Direction attribute equal to ‘return’, the return type of the Operation member method is ‘void’.
3. If there is no Parameter element with Direction attribute different than ‘return’, the only argument of the Operation member method is ‘void’.
4. If there is more than one Parameter element with Direction attribute different than ‘return’, they are mapped in sequence as comma-separated arguments for the Operation member method.

NOTE See clause 6.1.4.5 for details on the mapping of Parameter elements.

d. The access specifier of the Operation C++ member method shall be defined as follows:

1. If the Operation belongs in an Interface, the member is public;
2. If the Operation does not belong in an Interface, the mapping of the Visibility attribute is used.

NOTE See clause 6.1.3 for details on the mapping of Visibility attributes.

e. The Static attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Operation C++ mapping:

1. If set to “true”, then the C++ mapping includes the ‘static’ specifier as per “Operation” or “Operator” template in Table 6-1;
2. If not set, then it has no effect;
3. If set to “false”, then it has no effect.

NOTE Constructor methods are not affected by the Static attribute.

f. The Virtual attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Operation C++ mapping:

1. If set to “true”, then the C++ mapping includes the ‘virtual’ specifier as per “Operation” or “Operator” template in Table 6-1;
2. If not set, then the C++ mapping includes the ‘virtual’ specifier as per “Operation” or “Operator” template in Table 6-1 if the Operation belongs to an Interface, Model or Service;
3. If set to “false”, then it has no effect.

NOTE Constructor methods are not affected by the Virtual attribute.

g. The Abstract attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Operation C++ mapping:

1. If set to "true", then the C++ mapping includes the '=0' pure specifier as per "Operation" or "Operator" template in Table 6-1;
2. If not set, then the C++ mapping includes the '=0' pure specifier as per "Operation" or "Operator" template in Table 6-1 if the Operation belongs to an Interface;
3. If set to "false", then it has no effect.

NOTE Constructor methods are not affected by the Abstract attribute.

h. The Const attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Operation C++ mapping:

1. If set to "true", then the C++ mapping includes the 'const' specifier as per "Operation" or "Operator" template in Table 6-1;
2. If not set, then it has no effect;
3. If set to "false", then it has no effect.

NOTE Constructor methods are not affected by the Const attribute.

i. The Operator attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Operation C++ mapping:

1. If set, then the C++ mapping of the Operator kind referenced in the Operator attribute as per Table 6-7 is used.
2. If not set, it has no effect.

Table 6-7: C++ mapping for the Operator attribute kinds

Operator kind	Description	C++ mapping
None	Undefined.	
Positive	Positive value of instance.	+x
Negative	Negative value of instance.	-x
Assign	Assigns new value to instance.	x = a
Add	Adds value to instance.	x += a
Subtract	Subtracts value to instance.	x -= a
Multiply	Multiplies instance with value.	x *= a
Divide	Divides instance by value.	x /= a
Remainder	Remainder of instance for value.	x %= a
Greater	Compares whether instance is greater than value.	x > a
Less	Compares whether instance is less than value.	x < a
Equal	Compares whether instance is equal to value.	x == a
NotGreater	Compares whether instance is not greater than value.	x <= a
NotLess	Compares whether instance is not less than value.	x >= a
NotEqual	Compares whether instance is not equal to value.	x != a

Operator kind	Description	C++ mapping
Indexer	Returns indexed value of instance.	x[a]
Sum	Returns sum of two values.	a + b
Difference	Returns difference of two values.	a - b
Product	Returns product of two values.	a * b
Quotient	Returns quotient of two values.	a / b
Module	Returns remainder of two values.	a % b

6.1.4.8 EntryPoint

a. EntryPoint elements shall be mapped to ISO/ANSI C++ member pointer variables as per "EntryPoint" template in Table 6-1.

b. The access specifier of the EntryPoint member variable shall be public.

c. The EntryPoint member variable shall point to an implementation of the Smp::IEntryPoint interface.

6.1.4.9 EventSink

a. EventSink elements shall be mapped to ISO/ANSI C++ member pointer variables as per "EventSink" template in Table 6-1.

b. The access specifier of the EventSink member variable shall be public.

c. The EventSink member variable shall point to an implementation of the Smp::IEventSink interface.

d. If the EventType of an EventSink has an EventArgs, the implementation of the Notify method of the Smp::IEventSink interface shall expect to receive an "arg" parameter of simple type as defined by the type of the EventArgs.

NOTE See clause 5.2.6.1 for the details of the Notify method of the Smp::IEventSink interface.

6.1.4.10 EventSource

a. EventSource elements shall be mapped to ISO/ANSI C++ member pointer variables as per "EventSource" template in Table 6-1.

b. The access specifier of the EventSource member variable shall be public.

c. The EventSource member variable shall point to an implementation of the Smp::IEventSource interface.

d. If the EventType of an EventSource has an EventArgs, the implementation of the Emit method of the Smp::IEventSource interface shall expect to pass an "arg" parameter of simple type as defined by the type of the EventArgs.

NOTE See clause 5.2.6.2 for the details of the Emit method of the Smp::IEventSource interface.

6.1.4.11 Container

a. Container elements shall be mapped to ISO/ANSI C++ member pointer variables as per "Container" template in Table 6-1.

b. The access specifier of the Container member variable shall be public.

c. The Container member variable shall point to an implementation of the Smp::IContainer interface.

d. If the Type element of the Container points to a reference type, then the implementation of the AddComponent method of the Smp::IContainer interface shall expect the component parameter to be derived from this Type.

NOTE See clause 5.2.5.2 for the details of the AddComponent method of the Smp::IContainer interface.

6.1.4.12 Reference

a. Reference elements shall be mapped to ISO/ANSI C++ member pointer variables as per "Reference" template in Table 6-1.

b. The access specifier of the Reference member variable shall be public.

c. The Reference member variable shall point to an implementation of the Smp::IReference interface.

d. If the Type element of the Reference points to a reference type, then the implementation of the AddComponent method of the Smp::IReference interface shall expect the component parameter to be derived from this Type.

NOTE See clause 5.2.4.2 for the details of the AddComponent method of the Smp::IReference interface.

6.1.5 Basic Value Types

6.1.5.1 Common specification

a. For each type, a universally unique identifier (UUID) variable shall be declared as per "Uuid" template in Table 6-1.

b. The value of the universally unique identifier (UUID) variable shall be defined as per "Uuid" template in Table 6-2.

c. For each type, a method to register the type in the registry shall be defined as per "Global Registry" template in Table 6-1.

d. If the type belongs to a Reference Type, the access specifier of the C++ member variables, types and methods related to the type shall be defined by the mapping of the Visibility attribute.

NOTE See clause 6.1.3 for details on the mapping of Visibility attributes.

6.1.5.2 Enumeration

a. Enumeration types shall be mapped to ISO/ANSI C++ enumerated types as per "Enumeration" template in Table 6-1.

b. Literal elements shall be mapped to ISO/ANSI C++ enumeration literals with value assignment as per "Literal" template in Table 6-1.

c. Literal elements shall be declared within the exact same Enumeration type as in the Catalogue.

6.1.5.3 Integer

a. Integer types shall be mapped to ISO/ANSI C++ type definitions as follows:

   1. Syntax is as per "Integer" template in Table 6-1;
   2. If it references a specific type, the same is used for the declaration;
   3. If it does not reference a type, the default Int32 primitive type as per Table 5-1 is used for the declaration.

6.1.5.4 Float

a. Float types shall be mapped to ISO/ANSI C++ type definitions as follows:

   1. Syntax is as per "Float" template in Table 6-1;
   2. If it references a specific type, the same is used for the declaration;
   3. If it does not reference a type, the default Float64 primitive type as per Table 5-1 is used for the declaration.

6.1.5.5 String

a. String types shall be mapped to ISO/ANSI C++ structures as per "String" template in Table 6-1.

NOTE 1 Using a structure with a single internalString array field (rather than using an array) allows passing String types by value.

NOTE 2 The extension of one extra character in length ensures that the terminating NULL character fits into the string.

6.1.5.6 Array

a. Array types shall be mapped to ISO/ANSI C++ structures as per "Array" template in Table 6-1.

NOTE Using a structure with a single internalArray array field (rather than using an array) allows passing Array types by value.

6.1.6 Compound Value Types

6.1.6.1 Common specification

a. For each type, a universally unique identifier (UUID) variable shall be declared as per "Uuid" template in Table 6-1.

b. The value of universally unique identifier (UUID) variables shall be defined as per "Uuid" template in Table 6-2.

c. For each type, a method to register the type in the registry shall be defined as follows:

   1. Syntax is as per "Scoped Registry" template in Table 6-1;
   2. Method is declared as member of the C++ structure or class the type is mapped to.

d. Constant and Field elements belonging to the type shall be mapped within the exact same C++ structure or class the type is mapped to.

   NOTE See clause 6.1.4.1c.2 for details on the mapping of Constant elements and clause 6.1.4.3 for details on the mapping of Field elements.

e. If the type belongs to a Reference Type, the access specifier of the C++ member variables, types and methods related to the type shall be defined by the mapping of the Visibility attribute.

   NOTE See clause 6.1.3 for details on the mapping of Visibility attributes.

6.1.6.2 Structure

a. Structure types shall be mapped to ISO/ANSI C++ structures as per "Structure" template in Table 6-1.

6.1.6.3 Class

a. Class types shall be mapped to ISO/ANSI C++ classes as follows:

   1. Syntax as per "Class" template in Table 6-1;
   2. If the Base element is defined, the class inherits from the Base class.

b. Class types shall have a default constructor whose access specifier is defined by the mapping of the Visibility attribute.

   NOTE See clause 6.1.3 for details on the mapping of Visibility attributes.

c. Class types shall have a virtual destructor with the noexcept keyword whose access specifier is defined by the mapping of the Visibility attribute.

   NOTE See clause 6.1.3 for details on the mapping of Visibility attributes.

d. If the Class type has the NoConstructor attribute as per ecss.smp.smpcat in [SMP_FILES] set to "true", the constructor shall be declared with the delete keyword.

e. If the Class type has the NoDestructor attribute as per ecss.smp.smpcat in [SMP_FILES] set to "true", the destructor shall be declared with the default keyword.

f. Association, Property and Operation elements belonging to the Class type shall be mapped within the exact same C++ class the type is mapped to.

NOTE See clause 6.1.4.4 for details on the mapping of Association elements, clause 6.1.4.6 for details on the mapping of Property elements and clause 6.1.4.7 for details on the mapping of Operation elements.

g. If the Class type has the Abstract attribute set to “true”, the destructor shall be declared as pure virtual.

h. The BaseClass attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Class C++ mapping:

1. If set, then the class includes an inheritance link to the base class that the attribute points to;
2. If not set, then it has no effect.

6.1.6.4 Exception

a. Exception types shall be mapped to ISO/ANSI C++ classes as follows:

1. Syntax as per “Exception” template in Table 6-1;
2. If the Base element is defined, the class inherits from the Base class;
3. If the Base element is not defined, the class inherits from the default Exception class.

b. Exception classes shall have a default constructor whose access specifier is defined by the mapping of the Visibility attribute.

NOTE See clause 6.1.3 for details on the mapping of Visibility attributes.

c. Exception classes shall have a copy constructor whose access specifier is defined by the mapping of the Visibility attribute.

NOTE 1 Copy constructors are required to be able to catch exceptions by value.

NOTE 2 See clause 6.1.3 for details on the mapping of Visibility attributes.

d. Exception classes shall have a virtual destructor whose access specifier is defined by the mapping of the Visibility attribute.

NOTE See clause 6.1.3 for details on the mapping of Visibility attributes.

e. Association, Property and Operation elements belonging to the Exception type shall be mapped within the exact same C++ class the type is mapped to.

NOTE See clause 6.1.4.4 for details on the mapping of Association elements, clause 6.1.4.6 for details on the mapping of Property elements and clause 6.1.4.7 for details on the mapping of Operation elements.

f. If the Exception type has the Abstract attribute set to "true", the destructor shall be declared as pure virtual.

g. The BaseClass attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Exception C++ mapping:

1. If set, then the Exception includes an inheritance link to the base class that the attribute points to;
2. If not set, then it has no effect.

6.1.7 Reference Types

6.1.7.1 Common specification

a. For each type, a universally unique identifier (UUID) variable shall be declared as per "Uuid" template in Table 6-1.

b. The value of universally unique identifier (UUID) variables shall be defined as per "Uuid" template in Table 6-2.

c. Constant, Property and Operation elements belonging to the type shall be mapped within the exact same C++ class the type is mapped to.

NOTE See clause 6.1.4.2 for details on the mapping of Constant elements, clause 6.1.4.6 for details on the mapping of Property elements and clause 6.1.4.7 for details on the mapping of Operation elements.

d. The access specifier of class constructors and destructors within the C++ class a type is mapped to shall be defined by the mapping of the type Visibility attribute.

NOTE See clause 6.1.3 for details on the mapping of Visibility attributes.

6.1.7.2 Interface

a. Interface types shall be mapped to ISO/ANSI C++ abstract classes as follows:

1. Syntax as per "Interface" template in Table 6-1;
2. If Base elements are defined, the class inherits from the Base classes;
3. All class member methods are declared as pure virtual.

b. Interface classes shall have a virtual destructor with an empty implementation.

6.1.7.3 Model

a. Model types shall be mapped to ISO/ANSI C++ classes as follows:

1. Syntax as per "Model" template in Table 6-1;
2. If Base element is defined, the class inherits from the Base class;
3. If Interface elements are defined, the class inherits from the Interface classes;
4. If at least one EntryPoint is defined, the class inherits from the Smp::IEntryPointPublisher class;
5. If at least one EventSink element is defined, the class inherits from the Smp::IEventConsumer class;
6. If at least one EventSource element is defined, the class inherits from the Smp::IEventProvider class;
7. If at least one Container element is defined, the class inherits from the Smp::IComposite class;
8. If at least one Reference element is defined, the class inherits from the Smp::IAggregate class.

b. Model classes shall have a default constructor.

c. Model classes shall have a virtual destructor.

d. Field and Association elements belonging to the Model type shall be mapped within the exact same C++ class the Model type is mapped to.

NOTE See clause 6.1.4.3 for details on the mapping of Field elements and clause 6.1.4.4 for details on the mapping of Association elements.

e. EntryPoint, EventSink, EventSource, Container and Reference elements belonging to the Model type shall be mapped within the exact same C++ class the Model type is mapped to.

NOTE See clause 6.1.4.8 for details on the mapping of EntryPoint elements, clause 6.1.4.9 for details on the mapping of EventSink elements, clause 6.1.4.10 for details on the mapping of EventSource elements, clause 6.1.4.11 for details on the mapping of Container elements and clause 6.1.4.12 for details on the mapping of Reference elements.

f. The Fallible attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect:

1. If set to "true", then the C++ class implements the IFallibleModel interface;
2. If not set, then it has no effect;
3. If set to "false", then it has no effect.

g. The BaseClass attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Model C++ mapping:

1. If set, then the class includes an inheritance link to a base class that the attribute points to;
2. If not set, then it has no effect.

6.1.7.4 Service

a. Service types shall be mapped to ISO/ANSI C++ classes as follows:

   1. Syntax as per “Service” template in Table 6-1;
   2. If Base element is defined, the class inherits from the Base class;
   3. If Interface elements are defined, the class inherits from the Interface classes.
   4. If at least one EntryPoint is defined, the class inherits from the Smp::IEntryPointPublisher class;
   5. If at least one EventSink element is defined, the class inherits from the Smp::IEventConsumer class;
   6. If at least one EventSource element is defined, the class inherits from the Smp::IEventProvider class;

b. Service classes shall have a default constructor.

c. Service classes shall have a virtual destructor.

d. Field and Association elements belonging to the Service type shall be mapped within the exact same C++ class the Service type is mapped to.

   NOTE See clause 6.1.4.3 for details on the mapping of Field elements and clause 6.1.4.4 for details on the mapping of Association elements.

e. EntryPoint, EventSink and EventSource elements belonging to the Service type shall be mapped within the exact same C++ class the Service type is mapped to.

   NOTE See clause 6.1.4.8 for details on the mapping of EntryPoint elements, clause 6.1.4.9 for details on the mapping of EventSink elements and clause 6.1.4.10 for details on the mapping of EventSource elements.

f. The BaseClass attribute as per ecss.smp.smpcat in [SMP_FILES] shall have the following effect for the Model C++ mapping:

   1. If set, then the class includes an inheritance link to a base class that the attribute points to;
   2. If not set, then it has no effect.

6.2 Package to library

6.2.1 Mapping templates

a. Syntax and expression rules used in the specification of C++ mapping templates:

1. Information from the package to be mapped in the C++ code is specified by means of placeholders encased within dollar '$' symbols. For example, $Package.Name$ for the value of the field 'Name' of some 'Package' referred in the context the template is applicable.

NOTE Table 6-8 contains the C++ declaration templates for packages and is referred to from requirements of clause 6.2.

Table 6-8: C++ declaration templates for packages

Template	C++ mapping

Static Initialise	`extern "C" bool Initialise_$Package.Name$(`
	`   Smp::ISimulator* simulator,`
	`   Smp::Publication::ITypeRegistry* typeRegistry);`

Static Finalise	`extern "C" bool Finalise_$Package.Name$();`

Dynamic Initialise	`extern "C" bool Initialise(`
	`   Smp::ISimulator* simulator,`
	`   Smp::Publication::ITypeRegistry* typeRegistry);`

Dynamic Finalise	`extern "C" bool Finalise(Smp::ISimulator* simulator);`

DLL Initialise	`extern "C" DLL_EXPORT bool Initialise(`
	`   Smp::ISimulator* simulator,`
	`   Smp::Publication::ITypeRegistry* typeRegistry);`

DLL Finalise	`extern "C" DLL_EXPORT bool Finalise(Smp::ISimulator*`
	`simulator);`

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

6.2.2 Common to Unix and Windows

a. The SMDL Package Provider shall implement the Package as a Static or Dynamic Library file.

NOTE The Library file can be materialized differently on different Operating Systems.

b. The Library shall contain an Initialise function as per Initialise template in Table 6-8.

c. The Library shall contain a Finalise method as per Finalise template in Table 6-8.

d. The Finalise method function shall release memory allocated during Initialise method, unless ownership has been handed over.

e. The Initialise function shall register all user-defined Types in the library with the Type Registry using the provided Type Registry interface.

NOTE This is done by calling the global register function (for Enumeration, Integer, Float, Array, String) or method (Structure, Class, Exception) of the type.

f. The Initialise function shall register the class Factory of all implemented models in the library using the ISimulator RegisterFactory method.

NOTE The ownership of the class factory is handed over to the object implementing ISimulator.

g. The Initialise function shall register an instance of all Services in the library using the ISimulator AddService method.

NOTE The ownership of the service is handed over to the object implementing ISimulator.

6.2.3 Unix (Shared object)

a. The SMDL Package shall be implementation mapped on UNIX based Operation Systems using on the following two methods:

1. As a Static Library file with extension ".a";
2. As a Dynamic Shared Object file with extension ".so".

b. The Static Library shall contain an Initialise method as per the "Static Initialise" template in Table 6-8.

c. The Dynamic Shared Object shall contain an Initialise method as per the "Dynamic Initialise" template in Table 6-8.

d. The Static Library shall contain a Finalise method as per the Static Finalise template in Table 6-8.

e. The Dynamic Shared Object shall contain a Finalise method as per the "Dynamic Finalise" template in Table 6-8.

f. The Initialise function shall call the Initialise$Package.Name$ ( ) function.

g. The Finalise function shall call the Finalise$Package.Name$ ( ) function.

h. The Initialise$Package.Name$ function shall call the initialization functions of the Packages which are referenced as Dependencies of the Package.

NOTE 1 Dependency indicates that a type referenced from an implementation in the package needs a type implemented in the referenced package.

NOTE 2 There are no rules on the order in which packages are initialised, as the type registration process via Universally Unique Identifiers (UUIDs) does not introduce dependencies on the order.

i. The Initialise and Finalise functions shall be implemented so that multiple calls are possible.

NOTE 1 The Initialise and Finalise functions may get called several times during initialization when a library is referenced from more than one package.

NOTE 2 Ensuring that types are only registered once and memory is only allocated once allows multiple calls to Initialise.

j. Packages shall map to either static or dynamic libraries.

NOTE 1 Two dynamic library implementations are currently mapped

- Unix Shared Object (SO)
- Windows Dynamic Link Library (DLL)

NOTE 2 The requirements for the static library are common to all the dynamic library implementations, therefore they are not repeated in the corresponding clauses. The clauses on the dynamic library implementations cover only the specific delta specifications applicable to the case at hand.

6.2.4 Addendum for Windows Dynamic Link Library (DLL)

a. A package shall be mapped to a single DLL file.

b. A single DLL file shall implement a single package.

c. All functions exported by a DLL file shall be exported with platform-specific decorations based on the calling convention.

NOTE This is typically achieved by using the 'C' linkage (`extern "C"`) along with the `__declspec(dllexport)` storage-class attributes.

d. A DLL file shall export the function Initialise() with the following "DLL Initialise" template in Table 6-8 where DLL_EXPORT is as per "DLL_EXPORT" template in Table 6-8.

e. A DLL file shall export the function Finalise() with the following DLL Finalise template in Table 6-8 where DLL_EXPORT is as per DLL_EXPORT template in Table 6-8.

6.2.5 SMP Bundle

a. A SMP bundle shall be composed by one or more SMDL packages.

b. A SMP bundle shall be composed by one or more package dynamic libraries, directly related to the SMDL packages.

c. A SMP bundle may be composed by one or more package static libraries, directly related to the SMDL packages.

d. A SMP bundle shall be composed by all the SMP catalogues related to the SMDL packages.

e. A SMP Bundle shall include a SMP manifest file in conformace with the Manifiest file DRD of Annex D.

Annex A (normative) Catalogue file - DRD

A.1 Catalogue DRD

A.1.1 Requirement identification and source document

This DRD is called from ECSS-E-ST-40-07 requirement 5.4.1.1a.

A.1.2 Purpose and objective

The purpose of the Catalogue file is to hold the model meta data.

A.2 Expected response

A.2.1 Scope and content

a. The suffix for catalogue files shall be “smpcat”.

b. The document shall be compliant with the Catalogue XML XSD in XML/Smdl/Catalogue.xsd in [SMP_FILES] and the files referred from it:

   1. XML/Core/Types.xsd in [SMP_FILES]
   2. XML/Core/Elements.xsd in [SMP_FILES]

A.2.2 Special remarks

None.

Annex B (normative) Package file - DRD

B.1 Package DRD

B.1.1 Requirement identification and source document

This DRD is called from ECSS-E-ST-40-07 requirement 5.4.2.1a.

B.1.2 Purpose and objective

The purpose of the Package file is to contain all metamodel elements that are needed in order to define how implementations of types defined in catalogues are packaged.

B.2 Expected response

B.2.1 Scope and content

a. The suffix for package files shall be “smppkg”.

b. The document shall be compliant with the Package XML XSD in xml/Smdl/Package.xsd in [SMP_FILES] and the files referred from it:

   1. xml/Smdl/Types.xsd in [SMP_FILES]

   2. xml/Smdl/Elements.xsd in [SMP_FILES]

B.2.2 Special remarks

None.

Annex C (normative) Configuration file - DRD

C.1 Configuration DRD

C.1.1 Requirement identification and source document

This DRD is called from ECSS-E-ST-40-07 requirement 5.4.3.1a

C.1.2 Purpose and objective

The purpose of the Configuration file is to hold configuration data for a simulation.

C.2 Expected response

C.2.1 Scope and content

a. The suffix for configuration files shall be "smpcfg".

b. The document shall be compliant with the Configuration XML XSD in xml/Smdl/Configuration.xsd in [SMP_FILES] and the files referred from it:

   1. xml/Smdl/Types.xsd in [SMP_FILES]

   2. xml/Smdl/Elements.xsd in [SMP_FILES]

C.2.2 Special remarks

None.

Annex D (normative) Manifest file - DRD

D.1 Configuration DRD

D.1.1 Requirement identification and source document

This DRD is called from ECSS-E-ST-40-07 requirement 6.2.5e.

D.1.2 Purpose and objective

The purpose of the Manifest file is to hold meta data for a bundle.

D.2 Expected response

D.2.1 Scope and content

a. The SMP Manifest files name shall be “SMP.MF”.

b. The SMP Manifest file shall be an ASCII file which contains key-value pairs in the following format: “Key: Value”

c. In the SMP Manifest file the Key and Value shall be separated by a colon.

d. In the SMP Manifest file the, the Key shall only contain alpha-numerical characters, underscore (“_”) or dash (“-“).

e. In the SMP Manifest file the, the Value shall start at the first non-whitespace character after the colon (“:”), and is terminated by the end of line.

f. The SMP Manifest file shall contain the Mandatory Keys listed in Table D-1 as indicated in the Mandatory column.

g. The SMP Manifest file shall conform to the OSGi Core Release 6 Bundle Manifest file format.

NOTE Internet link to the OSGI Core manifest: https://osgi.org/download/r6/osgi.core-6.0.0.pdf

Table D-1: SMP Manifest Key

Key	Meaning	Mandatory
Bundle-Copyright	Copyright statement for the bundle.	Yes
Bundle-ContactAddress	Full address of a person or company that can be contacted.	No
Bundle-DocURL	URL where documentation for the bundle can be retrieved from.	No
Bundle-Description	Textual description of the bundle and its content.	Yes
Bundle-ManifestVersion	A bundle manifest may express the version of the OSGi manifest header syntax in the Bundle-ManifestVersion header. If specified, the bundle manifest version must be '2'.	Yes
Bundle-Name	The Bundle-Name header defines a readable name for this bundle. This should be a short, human-readable name that can contain spaces.	Yes
Bundle-SymbolicName	The Bundle-SymbolicName manifest header is a mandatory header. The bundle symbolic name and bundle version allow a bundle to be uniquely identified in the Framework. That is, a bundle with a given symbolic name and version is treated as equal to another bundle with the same (case sensitive) symbolic name and exact version. The installation of a bundle with a Bundle-SymbolicName and Bundle-Version identical to an existing bundle fail.	Yes
Bundle-Vendor	The Bundle-Vendor header contains a human-readable description of the bundle vendor.	Yes
Bundle-Version	Bundle-Version is an optional header; the default value is 0.0.0. A version consists of major, minor and micro version components. If the minor or micro version components are not specified, they have a default value of 0. Versions are comparable. Their comparison is done numerically and sequentially on the major, minor, and micro components. A version is considered equal to another version if the major, minor, and micro components are equal.	Yes
Require-Bundle	The Require-Bundle header specifies the required exports from another bundle. This is a comma-separated list of required bundles, where each bundle is at least specified by its symbolic name, optionally followed by a specific version: <Bundle-SymbolicName>[; Bundle-Version="<Bundle-Version>"]	No

Key	Meaning	Mandatory
Compiler-Name	Name of the compiler that has been used to compile the source code.	No
Compiler-Version	Version of the compiler that has been used to compile the source code.	No
OS-Name	Name of the Operating System.	No
OS-Version	Version of the Operating System.	No

D.2.2 Special remarks

None.

Bibliography

ECSS-S-ST-00	ECSS system – Description, implementation and general requirements
ISO 9000 series	Quality management systems standards International Organization for Standardization (ISO) http://www.iso.org
ISO/IEC 9899:2011	ISO/IEC 9899:2011 Information technology -- Programming languages -- C
ISO/IEC 14882:2011	ISO/IEC 14882:2011 Information technology -- Programming languages -- C++
Open Group UUID	Open Group http://www.opengroup.org
OSGi Manifest	Open Services Gateway initiative http://www.osgi.org
SMP v1.2	Simulation Model Portability Specification version 1.2
XML	Extensible Markup Language World Wide Web Consortium (W3C) http://www.w3.org/XM