



BSI Standards Publication

Space engineering — Simulation modelling platform

National foreword

This British Standard is the UK implementation of EN 16603-40-07:2020.

The UK participation in its preparation was entrusted to Technical Committee ACE/68, Space systems and operations.

A list of organizations represented on this committee can be obtained on request to its secretary.

This publication does not purport to include all the necessary provisions of a contract. Users are responsible for its correct application.

© The British Standards Institution 2020
Published by BSI Standards Limited 2020

ISBN 978 0 539 03261 1

ICS 49.140

Compliance with a British Standard cannot confer immunity from legal obligations.

This British Standard was published under the authority of the Standards Policy and Strategy Committee on 30 June 2020.

Amendments/corrigenda issued since publication

Date	Text affected
------	---------------

English version

Space engineering - Simulation modelling platform

Ingénierie spatiale - Plateforme de modélisation pour
simulationRaumfahrttechnik - Teil 40-07: Modelliersoftware-
Plattform

This European Standard was approved by CEN on 17 May 2020.

CEN and CENELEC members are bound to comply with the CEN/CENELEC Internal Regulations which stipulate the conditions for giving this European Standard the status of a national standard without any alteration. Up-to-date lists and bibliographical references concerning such national standards may be obtained on application to the CEN-CENELEC Management Centre or to any CEN and CENELEC member.

This European Standard exists in three official versions (English, French, German). A version in any other language made by translation under the responsibility of a CEN and CENELEC member into its own language and notified to the CEN-CENELEC Management Centre has the same status as the official versions.

CEN and CENELEC members are the national standards bodies and national electrotechnical committees of Austria, Belgium, Bulgaria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, France, Germany, Greece, Hungary, Iceland, Ireland, Italy, Latvia, Lithuania, Luxembourg, Malta, Netherlands, Norway, Poland, Portugal, Republic of North Macedonia, Romania, Serbia, Slovakia, Slovenia, Spain, Sweden, Switzerland, Turkey and United Kingdom.



CEN-CENELEC Management Centre:
Rue de la Science 23, B-1040 Brussels

Table of contents

European foreword.....	7
Introduction.....	8
1 Scope.....	9
2 Normative references	10
3 Terms, definitions and abbreviated terms.....	11
3.1 Terms from other standards.....	11
3.2 Terms specific to the present standard	11
3.3 Abbreviated terms.....	16
3.4 Nomenclature	16
4 Principles	17
4.1 Objectives.....	17
4.2 Common Concepts and common types	17
4.3 Architecture	18
4.4 Time handling principle.....	19
4.5 Simulation lifecycle	20
4.6 Simulation method.....	21
4.6.1 Discrete-event simulation (DES)	21
4.6.2 Parallelization and distribution.....	22
4.6.3 Inter component communication	22
4.7 Models, Services and Components	23
4.7.1 Objects.....	23
4.7.2 Components.....	25
4.7.3 Factories	27
4.7.4 Models and Services.....	27
4.8 Publication and Persistence.....	28
4.9 Dynamic invocation.....	29
4.10 Components meta data	31
4.10.1 Catalogue	31
4.10.2 Package.....	31

4.10.3	Configuration.....	32
4.11	Model exchanges considerations.....	32
4.11.1	Overview.....	32
4.11.2	SMP Bundle.....	32
5	Interface requirements.....	33
5.1	Common.....	33
5.1.1	Primitive Types specification.....	33
5.1.2	Time Kinds.....	35
5.1.3	Path string.....	36
5.1.4	Universally Unique Identifiers (UUID).....	37
5.1.5	Exception specification.....	37
5.2	Components and Objects interfaces.....	37
5.2.1	Object Specification (IObject).....	37
5.2.2	Collection Specification (ICollection).....	38
5.2.3	Component Specification.....	39
5.2.4	Aggregation.....	42
5.2.5	Composition.....	45
5.2.6	Events.....	47
5.2.7	Entry points.....	50
5.2.8	Dynamic Invocation.....	50
5.2.9	Persistence (IPersist).....	54
5.2.10	Failures.....	55
5.2.11	Field interfaces.....	56
5.2.12	Requirements on utilization of Simulation Environments interfaces by components.....	62
5.3	Simulation Environment interfaces.....	63
5.3.1	Logger (ILogger interface).....	63
5.3.2	Time Keeper (ITimeKeeper).....	65
5.3.3	Scheduler (IScheduler).....	67
5.3.4	Event Manager (IEventManager).....	75
5.3.5	Resolver (IResolver).....	79
5.3.6	Link Registry (ILinkRegistry).....	80
5.3.7	Simulator (ISimulator).....	82
5.3.8	Persistence.....	94
5.3.9	Publication.....	95
5.3.10	Type Registry.....	102
5.3.11	Component Factory (IFactory).....	107

5.4	Meta data	108
5.4.1	Catalogue	108
5.4.2	Package.....	112
5.4.3	Configuration data.....	112
6	Implementation mapping	113
6.1	Catalogue to C++	113
6.1.1	Mapping templates.....	113
6.1.2	Namespaces and files.....	116
6.1.3	Element and Type Visibility Kind	116
6.1.4	Mapping of elements.....	117
6.1.5	Basic Value Types	126
6.1.6	Compound Value Types.....	128
6.1.7	Reference Types.....	130
6.2	Package to library.....	133
6.2.1	Mapping templates.....	133
6.2.2	Common to Unix and Windows	133
6.2.3	Unix (Shared object)	134
6.2.4	Addendum for Windows Dynamic Link Library (DLL)	135
6.2.5	SMP Bundle.....	136
Annex A	(normative) Catalogue file - DRD	137
A.1	Catalogue DRD	137
A.1.1	Requirement identification and source document.....	137
A.1.2	Purpose and objective.....	137
A.2	Expected response	137
A.2.1	Scope and content	137
A.2.2	Special remarks	137
Annex B	(normative) Package file - DRD	138
B.1	Package DRD.....	138
B.1.1	Requirement identification and source document.....	138
B.1.2	Purpose and objective.....	138
B.2	Expected response	138
B.2.1	Scope and content	138
B.2.2	Special remarks	138
Annex C	(normative) Configuration file - DRD	139
C.1	Configuration DRD.....	139
C.1.1	Requirement identification and source document.....	139

C.1.2	Purpose and objective.....	139
C.2	Expected response	139
C.2.1	Scope and content	139
C.2.2	Special remarks	139
Annex D (normative) Manifest file - DRD		140
D.1	Configuration DRD.....	140
D.1.1	Requirement identification and source document.....	140
D.1.2	Purpose and objective.....	140
D.2	Expected response	140
D.2.1	Scope and content	140
D.2.2	Special remarks	142
Bibliography.....		143

Figures

Figure 4-1: Common Concepts and Type System	18
Figure 4-2: SMP Architecture	18
Figure 4-3: SMP State machine.....	20
Figure 4-4: Object mechanisms.....	24
Figure 4-5: Overview of components hierarchy	25
Figure 4-6: Component Mechanisms	26
Figure 4-7: Component State machine	26
Figure 4-8: Sequence of calls for dynamic invocation	30

Tables

Table 4-1: Overview of simulation states	21
Table 4-2: ViewKind values	28
Table 5-1: Primitive Types.....	33
Table 5-2: Component states	39
Table 5-3: Semantically equivalent types for connections.....	61
Table 5-4: Default Log Message Kinds.....	64
Table 5-5: Condition for emitting predefined global events	78
Table 6-1: C++ declaration templates.....	114
Table 6-2: C++ definition templates	116
Table 6-3: C++ mapping for the Visibility kind attribute	116
Table 6-4: C++ mapping of Association depending on ByPointer attribute.....	119
Table 6-5: C++ mapping for the Direction kind attribute.....	120

Table 6-6: C++ mapping for Property depending on ByPointer attribute	121
Table 6-7: C++ mapping for the Operator attribute kinds	124
Table 6-8: C++ declaration templates for packages.....	133
Table D-1 : SMP Manifest Key	141

European foreword

This document (EN 16603-40-07:2020) has been prepared by Technical Committee CEN-CENELEC/TC 5 "Space", the secretariat of which is held by DIN.

This standard (EN 16603-40-07:2020) originates from ECSS-E-ST-40-07C.

This European Standard shall be given the status of a national standard, either by publication of an identical text or by endorsement, at the latest by December 2020, and conflicting national standards shall be withdrawn at the latest by December 2020.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. CEN [and/or CENELEC] shall not be held responsible for identifying any or all such patent rights.

This document has been prepared under a standardization request given to CEN by the European Commission and the European Free Trade Association.

This document has been developed to cover specifically space systems and has therefore precedence over any EN covering the same scope but with a wider domain of applicability (e.g. : aerospace).

According to the CEN-CENELEC Internal Regulations, the national standards organizations of the following countries are bound to implement this European Standard: Austria, Belgium, Bulgaria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, Former Yugoslav Republic of Macedonia, France, Germany, Greece, Hungary, Iceland, Ireland, Italy, Latvia, Lithuania, Luxembourg, Malta, Netherlands, Norway, Poland, Portugal, Romania, Serbia, Slovakia, Slovenia, Spain, Sweden, Switzerland, Turkey and the United Kingdom.

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 constraints 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.

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.

EN reference	Reference in text	Title
EN 16601-00-01	ECSS-S-ST-00-01	ECSS system - Glossary of terms
EN 16603-40	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

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 components implemented by storing their references

NOTE Each component in such a relationship keeps its own lifecycle and it does not depend on that of other components.

3.2.2 association

relationship between two instances of any data-type, where each instance 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 contract to its environment

3.2.5 composite

component implementing composition

3.2.6 composition

hierarchical relationship where child component is destroyed if the parent component 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.

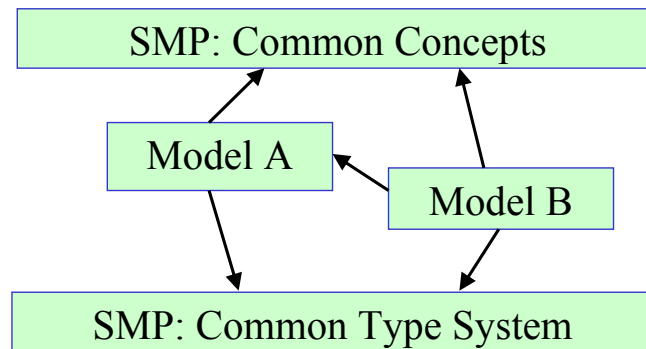


Figure 4-1: Common Concepts and Type System

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.

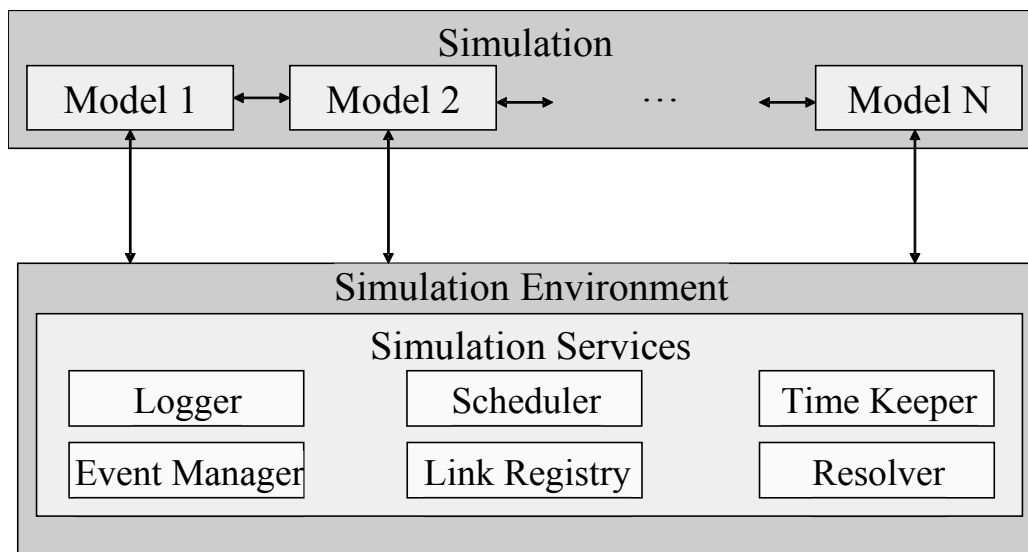


Figure 4-2: SMP Architecture

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).

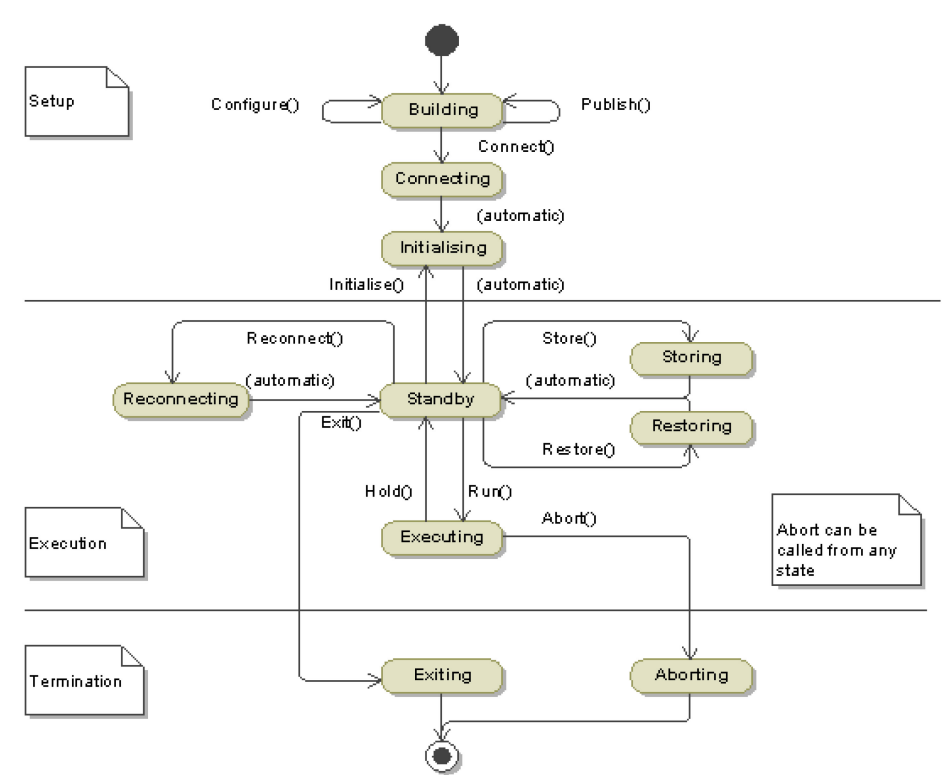


Figure 4-3: SMP State machine

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.

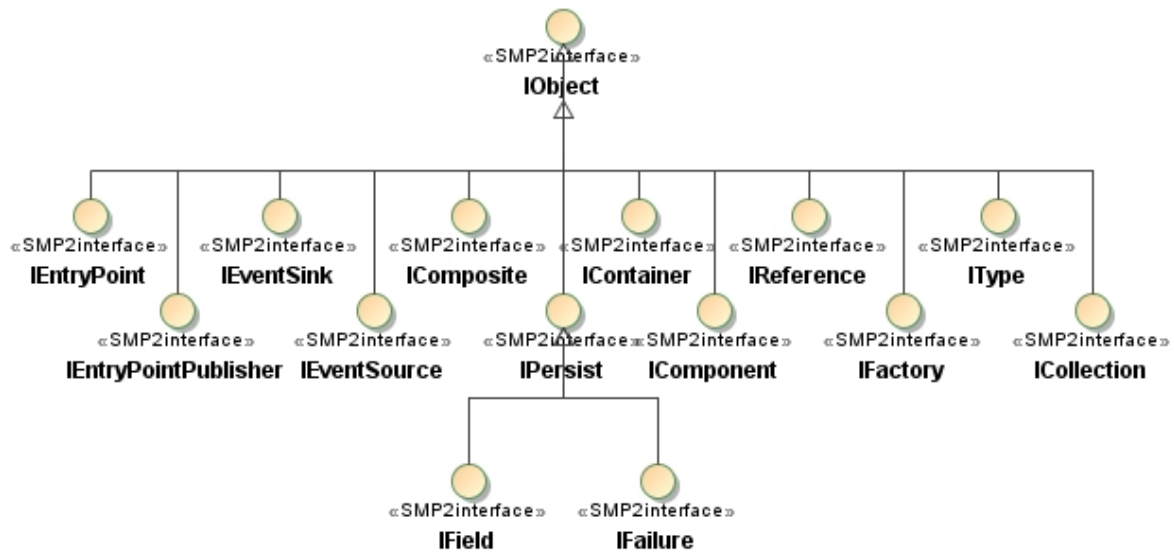


Figure 4-4: Object mechanisms

4.7.2 Components

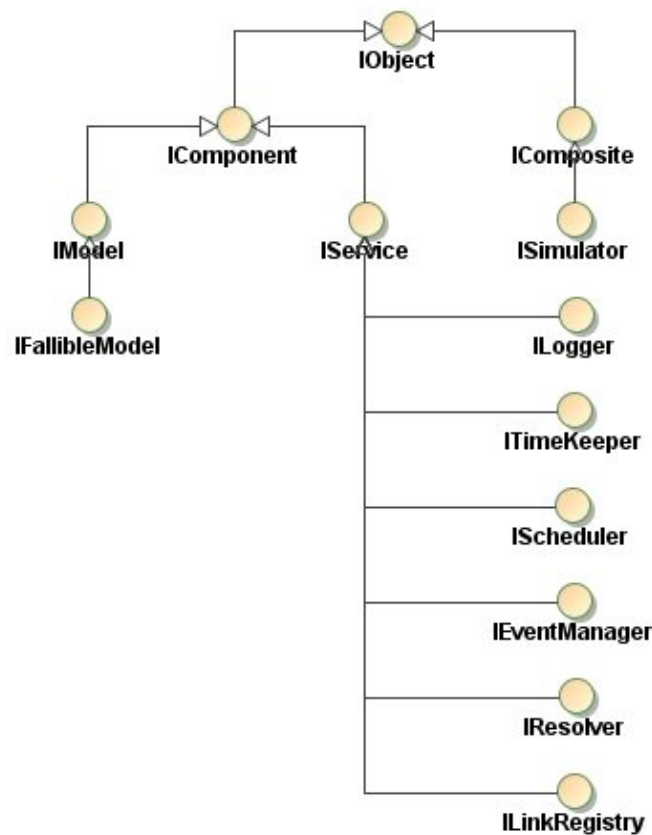


Figure 4-5: Overview of components hierarchy

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.

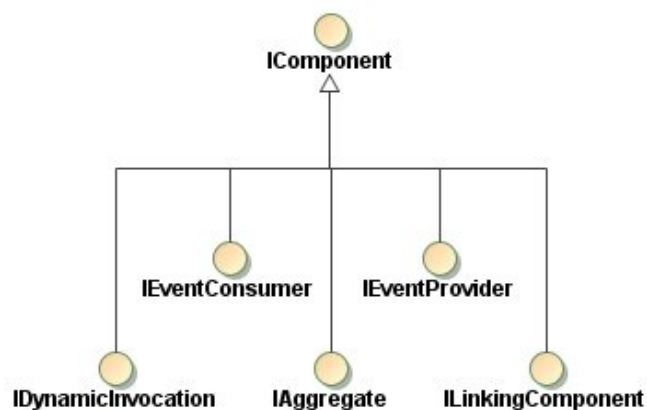


Figure 4-6: Component Mechanisms

All SMP components go 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).

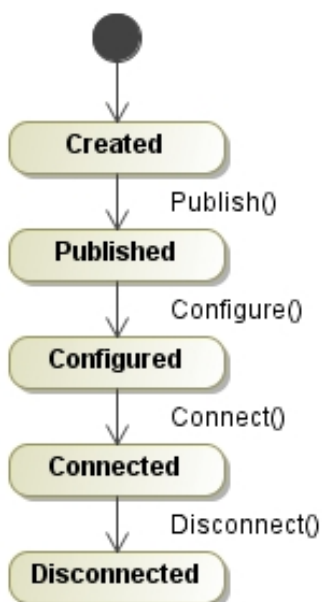


Figure 4-7: Component State machine

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.



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.

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 ISimulator.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.