



Space engineering

Simulation modelling platform - Volume 5: SMP usage

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

Foreword

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

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

Disclaimer

ECSS does not provide any warranty whatsoever, whether expressed, implied, or statutory, including, but not limited to, any warranty of merchantability or fitness for a particular purpose or any warranty that the contents of the item are error-free. In no respect shall ECSS incur any liability for any damages, including, but not limited to, direct, indirect, special, or consequential damages arising out of, resulting from, or in any way connected to the use of this Standard, whether or not based upon warranty, business agreement, tort, or otherwise; whether or not injury was sustained by persons or property or otherwise; and whether or not loss was sustained from, or arose out of, the results of, the item, or any services that may be provided by ECSS.

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

Copyright: 2011© by the European Space Agency for the members of ECSS

Change log

ECSS-E-TM-40-07 Volume 5A 25 January 2011	First issue
---	-------------

Table of contents

Change log	3
Introduction	11
1 Scope	13
2 Normative references	15
3 Terms, definitions and abbreviated terms	16
3.1 Terms defined in other standards.....	16
3.2 Terms specific to the present document	16
3.3 Abbreviated terms	18
4 Processes and Supporting Tools	20
4.1 SMP and E-40	20
4.2 ECSS-E-TM-40-07 Phases	21
4.2.1 System engineering processes related to software.....	21
4.2.2 Software management process.....	22
4.2.3 SW requirements and architectural engineering	22
4.2.4 SW design and implementation.....	23
4.2.5 SW validation.....	23
4.2.6 SW delivery and acceptance	24
4.2.7 SW verification.....	24
4.2.8 SW operations	24
4.2.9 SW maintenance	24
4.3 ECSS-E-TM-40-07 artefacts provided by SMP	24
4.3.1 SMP as a tool chain.....	24
4.3.2 SMP artefacts and the lifecycle	26
4.4 Comparison of SMP artefacts to traditional methods	29
5 Getting started with SMP	30
5.1 Introduction.....	30
5.2 Define a catalogue	30
5.3 Generate the SMP wrapper code.....	31

5.4	Add custom model code	32
5.5	Define a package	32
5.6	Generate a simulator component	32
5.7	Define an assembly	33
5.8	Define a schedule	34
5.9	Launch the simulation.	34
6	The Modelling Tasks	35
6.1	Introduction.....	35
6.2	Simulator architect tasks	35
6.2.1	Identify major sub-systems and their components	35
6.2.2	Identify data and control flows	37
6.2.3	Incorporate existing SMP models	38
6.2.4	Incorporate legacy models	39
6.2.5	Decide how to connect simulation components	40
6.2.6	Determine model scheduling	41
6.2.7	Decide how to configure simulation components	42
6.3	Model developer tasks	43
6.3.1	Implement new sub-systems and models	43
6.3.2	Prepare models for re-use.....	44
6.4	Simulator integrator/tester tasks.....	44
6.4.1	Set-up integration environment	45
6.4.2	Verify models.....	46
6.4.3	Integrate models.....	47
6.4.4	Test complete simulator	48
6.4.5	Verify simulator.....	49
7	The example simulator	51
7.1	Introduction.....	51
7.2	Functional Description of Example Simulator.....	51
7.3	Software architecture of example simulator	52
8	The simulator architect tutorial	56
8.1	Introduction.....	56
8.2	Task: Identify major sub-systems and their components	56
8.2.1	Overview.....	56
8.2.2	Capture the architecture in a catalogue.....	57
8.2.3	Organise the catalogue through namespaces.....	58
8.2.4	Structure the simulator with component-based design.....	59

8.2.5	Describe containment structure through interfaces	60
8.2.6	Define the sub-system models	60
8.3	Task: Identify data and control flows and their modelling	62
8.3.1	Overview.....	62
8.3.2	Interface-based design	62
8.3.3	Properties	67
8.3.4	Dataflow-based design	68
8.3.5	Event-based design.....	70
8.4	Task: Decide how to connect components.....	71
8.4.1	Overview.....	71
8.4.2	Assemblies	71
8.4.3	Interface-based design	72
8.4.4	Dataflow-based design	73
8.4.5	Event-Based Design.....	74
8.5	Determine model scheduling.....	75
8.5.1	Overview.....	75
8.5.2	Schedules.....	75
8.5.3	Scheduling – Using triggers.....	76
8.5.4	Scheduling – Using transfers.....	77
8.5.5	Scheduling - Triggering events.....	78
9	The model developer tutorial.....	79
9.1	Introduction.....	79
9.2	The SMP simulation environment and SMP models	79
9.3	Task: Implement new sub-systems and models.....	80
9.3.1	Generate code from catalogue	80
9.3.2	SMP MDK.....	81
9.3.3	Implementing basic SMP model features	81
9.3.4	Implementing sub-systems.....	87
9.3.5	Implement model interfaces	90
9.3.6	Implement event emitters and handlers	91
9.4	Task: Model re-use - Incorporate existing SMP models.....	94
9.4.1	Introduction.....	94
9.4.2	Sharing source code.....	94
9.4.3	Share binary component	94
9.4.4	Re-using models without changing functionality.....	95
9.4.5	Re-using models by extending existing functionality	95
9.5	Task: Model re-use incorporating non-SMP models	98

9.5.1	Introduction.....	98
9.5.2	Wrapping a non-SMP model with SMP interfaces.....	98
9.5.3	Simple example with stateless external model.....	99
9.5.4	Instantiation issues with “Statefull” external models.....	100
9.6	Task: Prepare models for re-use.....	101
10	The simulation environment.....	103
10.1	Introduction.....	103
10.2	The BUILDING state	104
10.3	The CONNECTING state	105
10.4	The INITIALISING state	105
10.5	The STANDBY state	106
10.6	The EXECUTING state	106
10.7	The STORING state	106
10.8	The RESTORING state	107
10.9	The RECONNECTING state	107
10.10	The EXITING state	107
10.11	The ABORTING state.....	107
11	The simulator integrator and tester tutorial	108
11.1	Introduction.....	108
11.2	Set up integration environment	108
11.3	Model Verification.....	110
11.4	Integrate simulator in stages	111
11.4.1	Overview.....	111
11.4.2	Re-use with containers (composition)	111
11.4.3	Re-use with references (aggregation)	113
11.4.4	Re-use with dataflow	115
11.4.5	Re-use with events.....	117
11.5	Test complete simulation.....	118
11.5.1	Simulator Integration	118
11.5.2	Sub-Assemblies Simplify Integration.....	120
11.6	Verify simulator.....	120
Annex A (informative)	Model Development Guidelines	122
A.1	Model Development Guidelines	122
A.2	Provide Documentation	122
A.3	Use Standard Languages.....	122
A.4	Avoid Compiler Specific Language Extensions.....	123

A.5	Use Standard Libraries Only	123
A.6	Limit Use of Essential Non-standard Libraries	124
A.7	Use the SMP Interfaces to interact with the Simulation Environment	124
A.8	Avoid Direct Input / Output Operations.....	125
A.9	Do Not Rely on Internal Representation of Data	125
A.10	Avoid Global Data Declarations	126
A.11	Avoid Common Global Names	126
A.12	Enable Multiple Instances	126
A.13	Minimise Number of Model Interfaces	127
A.14	Simplify Model Interfaces Provided	127
A.15	Only Select Suitable Candidates for Reuse	128

Bibliography.....129

Figures

Figure 4-1:	Areas of ECSS-E-TM-40-07 – Volume 1A Impacted by SMP	20
Figure 4-2:	SMP Tool Chain Overview.....	26
Figure 5-1:	HelloWorld Catalogue	31
Figure 5-2:	Generated EntryPoint Set Up Code.....	31
Figure 5-3:	Custom HelloWorld Entrypoint Code	32
Figure 5-4:	HelloWorld Package	32
Figure 5-5:	Generated HelloWorld Component.....	33
Figure 5-6:	HelloWorld Assembly.....	33
Figure 5-7:	HelloWorld Schedule	34
Figure 7-1:	Example Simulator Overview.....	52
Figure 7-2:	Example Simulator Main Components.....	53
Figure 7-3:	Space Segment Model.....	54
Figure 7-4:	Example Simulator Satellite Model Overview	54
Figure 8-1:	UML Model of Main Simulator Architectural Elements.....	57
Figure 8-2:	Snapshot SMP Catalogue Editor	58
Figure 8-3:	Namespaces in SMP	59
Figure 8-4:	Basic Schema of an SMP Model	61
Figure 8-5:	AOCS Sub-system Interfaces	63
Figure 8-6:	IAOCSControl Interface Definition	64
Figure 8-7:	AOCS Controller References Thruster Model.....	64
Figure 8-8:	IThrusterControl Interface Definition	65
Figure 8-9:	AOCSController Reference to IThrusterControl.....	65
Figure 8-10:	AOCS Interface Dependencies.....	66

Figure 8-11: Interface Inheritance	67
Figure 8-12: Catalogue View Of a read-Only Property.....	68
Figure 8-13: Read-only Property Code Snippet	68
Figure 8-14: Dataflow Example UML Class Model.....	69
Figure 8-15: Dataflow-based Example HK TM Data collection	69
Figure 8-16: PSS Event-based Design CatalogueView	71
Figure 8-17: Model Instances in an Assembly	72
Figure 8-18: Defining Interface-Links From AOCSController to the Thrusters.....	73
Figure 8-19: Diagram of Dataflow Field Links in Assembly Editor	74
Figure 8-20: Event Links for MainBus Status	75
Figure 8-21: AOCSControl Trigger and Task.....	77
Figure 8-22: Simulation Update Event	77
Figure 8-23: Schedule Task Containing Datalow Transfers.....	78
Figure 8-24: MainBus Events Entrypoint Scheduling	78
Figure 9-1: Simulation State Transitions	80
Figure 9-2: Definition of Fire() Operation.....	82
Figure 9-3: Fire Operation SMP C++ Mapping Definition.....	82
Figure 9-4: Fire() Operation C++ implementation	83
Figure 9-5: Thruster Model Schedule Entrypoint Definiton	84
Figure 9-6: Schedule Entrypoint Declaration.....	84
Figure 9-7: Schedule Entrypoint Implementation	84
Figure 9-8: Catalogue View of a Model Field	85
Figure 9-9: Field Initialisation Code.....	85
Figure 9-10: Field Publication Code.....	85
Figure 9-11: Definition of ForceVector Property.....	86
Figure 9-12: ForceVector Property Code	87
Figure 9-13: Catalogue View of Propulsion Sub-system Container For Thrusters.....	88
Figure 9-14: Model Container Generated Code Example	88
Figure 9-15: Model Container Developer Code.....	89
Figure 9-16: Model References Generated Code	90
Figure 9-17: Model References Developer Code.....	90
Figure 9-18: THR Model Header File IThruster Control Interface	91
Figure 9-19: IThrusterControl Fire() Implementation.....	91
Figure 9-20: EventSource Wrapper Code for PSS Main Power Bus	92
Figure 9-21: EventSink SMP Wrapper for S/C Equipment.....	92
Figure 9-22: Implementation of PSS MainBus EventSource.....	93
Figure 9-23: Implementing an a EventSink For MainBus Events.....	93

Figure 9-24: GYRO Inherits From InertialSensor	96
Figure 9-25: InertialSensor Functionality.....	97
Figure 9-26: Inheritance of the InertialSensor Model Code.....	97
Figure 9-27: Implementing Inherited Functionality	98
Figure 9-28: Model Wrapping Concept	99
Figure 9-29: Initialising External Models Example	99
Figure 9-30: Calling a Matlab Function from An SMP Model	100
Figure 9-31: Finalising External Models Example	100
Figure 9-32: Packaging the GYRO Model.....	101
Figure 10-1: Simulation Environment State Diagram	103
Figure 11-1: IPowerGenerator Interface.....	112
Figure 11-2: PowerGeneration Container	112
Figure 11-3 Power Sub-system Assembly	113
Figure 11-4: IThrusterControl Interface	114
Figure 11-5: AOCSCControl “Thrusters” Reference	114
Figure 11-6: Integrating AOCSCControl With Propulsion Sub-system Thrusters.....	115
Figure 11-7: Fields of the HouseKeeping PacketGenerator.....	116
Figure 11-8: Housekeeping Packet Fields links	116
Figure 11-9: Power Sub-system MainBusPower EventSource	117
Figure 11-10: Creating Links for the Power Sub-system.....	118

Tables

Table 4-1:SMP Artefacts in the Lifecycle	28
Table 4-2: Simulator Artefacts.....	29
Table 8-1 Simulation Time Kinds	76
Table 11-1: Bundle SMP Manifest.....	109
Table 11-2: Example Simulator Integration Elements	119

Introduction

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

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

This Technical Memorandum comprises of a number of volumes.

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

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

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

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

- **Volume 1 – Principles and requirements**

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

- **Volume 2 - Metamodel**

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

- **Volume 3 - Component Model**

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

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

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

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

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

- Infrastructure Supplier - is responsible for the development of generic infrastructure or for the adaptation of an infrastructure to the specific needs of a project. In the context of a space programme, the involvement of Infrastructure Supplier team(s) may not be required if all required simulators are based on full re-use of exiting infrastructure(s), or where the infrastructure has open interfaces allowing adaptations to be made by the Simulator Integrator.
- Model Supplier - is responsible for the development of project specific models or for the adaptation of generic models to the specific needs of a project or project phase.
- Simulator Integrator – has the function of integrating the models into a simulation infrastructure in order to provide a full system simulation with the appropriate services for the user (e.g. system engineer) and interfaces to other systems.

- **Volume 5 – SMP usage**

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

1 Scope

This document is intended as an introduction into the SMP technical memoranda. It is not a reference manual, i.e. it does not claim to be complete. The focus is on explaining the **usage** of SMP. Note that the content of this document is not normative as it mainly describes the normative elements associated to SMP from different perspectives, targeting the following types of SMP users:

Simulator Architects: They have overall responsibility for the design of a simulator and its variants. We assume they have already an SMP-compliant simulation environment, and have decided to use SMP models. They may choose different options for the design approach and supporting tools, as well as for model integration. Typically, simulator architects would need design tool support to effectively apply SMP (see Section 4 on process and tool support). Section 8 covers the relevant tasks for a simulator architect

Model Developers: Work in conjunction with the simulator architects or they may be building a library of models. They need to understand the mechanisms of interfacing with the simulation environment as well as those for inter-model communication. Model Developers with a background in SMP1/2 or C++ may start reading section 5 (Getting Started). The model developers specific tasks are covered in section 9. Model developers would typically make use of supporting tools (see section 4 on tool support) for their day-to-day work.

Integrators and Testers: Have responsibility for ensuring that the simulator can be assembled, deployed and tested on an SMP compliant infrastructure to meet the requirements as defined by the customer. Those requirements address the functional and non-functional elements of the simulator including end user services and interfaces to other systems. SMP pays particular regard to improving and facilitating the integration of components shared between different parties, projects and phases.

Additional users of the standard not addressed by this user guide are:

Tool Developers: They develop tools supporting the simulator designers or model developers. These tools may be stand-alone tools, e.g. an editor for designing a simulator or a validator for model validation, or tools interfacing SMDL to other tools, e.g. document generation, code generation, or import of

models designed in other tools (for example based on the Unified Modelling Language (**UML**)).

Environment Furnishers: They want to adapt an existing simulation environment to be SMP-compliant. They need to know which services they have to implement in their environment to make it SMP-compliant, and how loading, initialisation, scheduling and persistence work. Environment Furnishers will primarily be concerned with definition of the SMP simulation environment.

Readers will benefit from the following pre-requisites:

- The concept of Object Oriented Design (OOD), which is mandatory to understand SMP.
- The Unified Modelling Language (UML) that has been used for most of the diagrams.
- The C++ programming language that has been used for the example code.
- The eXtensible Markup Language (XML) that has been used for the SMDL example files.

It should be noted that this document serves only as an introduction to SMP and is not a comprehensive and exhaustive guide to its use. It is foreseen that for any SMP tool a user manual would provide the necessary details of the SMP techniques and mechanisms appropriate to the specific tool.

2

Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this ECSS Technical Memorandum. For dated references, subsequent amendments to, or revision of any of these publications do not apply. However, parties to agreements based on this ECSS Technical Memorandum 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-ST-S-00-01	ECSS system – Glossary of terms
ECSS-E-TM-10-21	Space engineering - System modelling and simulation
ECSS-E-ST-40C	Space engineering – Software

Terms, definitions and abbreviated terms

3.1 Terms defined in other standards

For the purposes of this document, the terms and definitions given in ECSS-S-ST-00-01 and ECSS-E-TM-10-21 apply.

3.2 Terms specific to the present document

3.2.1 component

An implemented model with well defined interfaces that can be delivered in source or object code form. One or more instances can be instantiated within a single simulation. (See also portability)

3.2.2 domain

Context of use in which a simulation and its models are intended. Common domains are e.g.: Operations, training, engineering (performance, testing), visualisation, ... (see also model)

3.2.3 initialisation

The setting of the initial state of a model or simulation before a simulation run is started.

3.2.4 integration

[1] In the domain of software engineering the joining of modules to form a complete system

[2] In the domain of simulation the mathematical integration of state variables, usually over time

3.2.5 model

By simulation models it is meant here both data models, e.g. geometrical model of a system, and behavioural models, e.g. the algorithms representing the behaviour of a component or environment expressed in a high level programming language. A model normally (but not always) has inputs, outputs and internal state variables and constants.

A generic model represents an entity (e.g. a power distribution network) that can be configured to represent any instantiation of that entity.

NOTE Although generic models are a powerful concept, they can become over complex resulting in models that may require more effort to use than to develop a specific model from scratch. A tradeoff between fully generic models, fully bespoke models and code reuse/code sharing between models should always to be made.

Depending on the context, models can be classified according to their fidelity, their domain or their modelling technique.

3.2.6 modelling technique

Method used to analyse and describe the behaviour of a model. Common techniques are e.g.: Physical (electrical, mechanical...), behavioural, functional (with respect to external interfaces), geometric, ... (see also model)

3.2.7 portability

The ability to use a model in different simulation environments, different hardware platforms and different operating systems, usually just by recompiling (compare interoperability).

3.2.8 real-time

A simulation in which the simulated time progresses at the same speed as the wall-clock time (but is not usually the same as wall clock time).

3.2.9 restore

The ability to load a saved simulation state and start a simulation run from the state where the simulation was saved. In most simulation environments, restore does not work if model variables have been added, removed, or have different types.

3.2.10 save

The ability to save the state of a simulation at a given instant in time. This is used to allow users to start a simulation from various predefined states e.g. Eclipse Entry (see also Restore).

3.2.11 savepoint

The complete state of a simulation (the value of all its parameters and variables) which can be saved in such a way that a simulation can be restored to the same state and resumed at a later time. This is as well being referred to as: *Stateset, Breakpoint, Snapshot*

Any alternative term can be proposed for "Savepoint" – legacy should be reflected in definition

3.2.12 scenario

A particular initial configuration of a simulator and sequence of events to represent a particular part of a mission e.g. launcher deployment, eclipse operations, cruise phase.

3.2.13 scheduler

A component of the simulation environment responsible for scheduling the execution of the models within a simulator. The scheduler may schedule models cyclically (often, though not always, at multiples of a base frequency) or intermittently (e.g. as a result of a telecommand arriving). (See as well continuous simulation and discrete simulation)

3.2.14 simulation

A run of scenario in a simulator with a simulated start- and end-time. During the simulation events may be injected into the simulation by the user, a script, external hardware or another simulation.

3.2.15 simulation environment

The software infrastructure that is used to run models. It usually has a scheduler, supports the control of the models (via scripts and/or a GUI), visualisation of their public state variables and provides the simulation time. It may also provide other services such as save/restore, logging of model events and other events. Examples are EUROSIM, SIMSAT, SIMWARE, BASILES and SimVis.

3.2.16 simulator

An ensemble of one or more models that are executed together to represent the behaviour of phenomena and/or an artificial system (e.g. spacecraft). It also includes the simulator kernel with the model scheduling.

3.2.17 state variables

Variables that represent a model's state at a moment in time. These may be public (visible to the simulation environment or other models) or private to the model itself.

NOTE These variables represent the (minimal) set of elements to be taken into account when proceeding Save and Restore actions.

3.2.18 state vector

The ensemble of the state variables, relevant to be kept for a *Savepoint*.

3.3 Abbreviated terms

The following abbreviations are defined and used within document:

Abbreviation	Meaning
ANSI/ISO	American National Standards Institute/International Organization for Standardization
AOCS	Attitude & Orbit Control System
API	Application Program Interface
AR	Acceptance Review
CBD	Component Based Design

CDR	Critical Design Review
CSS	Coarse Sun Sensor
DDF	Design Definition File
DJF	Design Justification File
ECSS	European Cooperation for Space Standardisation
EGOs	ESA Ground Operation System
EGOS-MF	EGOS Modelling Framework
EIDP	End Item Data Pack
GCC	GNU Compiler Collection
GYR	Gyro
HKTM	House-keeping telemetry
ICD	Interface Control Document
IDE	Integrated Development Environment
IPR	Intellectual Property Rights
LRM	Language Reference Manual
MDK	Model Development Kit
MIE	Model Integration Environment
NR	Normative Reference document
PDR	Preliminary Design Review
PSS	Power Supply Subsystem
QR	Qualification Review
RB	Requirements Baseline
RD	Reference Document (Bibliography)
SAD	Software Architectural Design
SDD	Software Detailed Design
SMDL	Simulation Model Definition Language
SMP	Simulation Modelling Platform
SPOT5	Satellite Pour l'Observation de la Terre-5
SRR	Software Requirements Review
SVTS	System Validation Testing Specification
TC	Telecommand
THR	Thruster
TM	Telemetry
TRR	Test Readiness Review
UML	Unified Modelling Language
VISMP2	Validation Industrielle SMP2
XML	Extensible Markup Language

4

Processes and Supporting Tools

4.1 SMP and E-40

This section explains how ECSS-E-TM-40-07 fits together with ECSS-E-ST-40C, what the outputs are what tools can be used to produce them. It should be noted that ECSS-E-TM-40-07, while being in-line with the ECSS-E-ST-40 process, is not dependent on it and as such could be used in other domains for which SMP may be beneficial.

ECSS-E-TM-40-07 provides the SW lifecycle process specific to simulator development. It is natural for SMP have a significant bearing on **HOW** this lifecycle is implemented for a simulations project.

The main process steps for ECSS-E-TM-40-07 is shown in Figure 4-1 with indications where in the process SMP activities are realised:

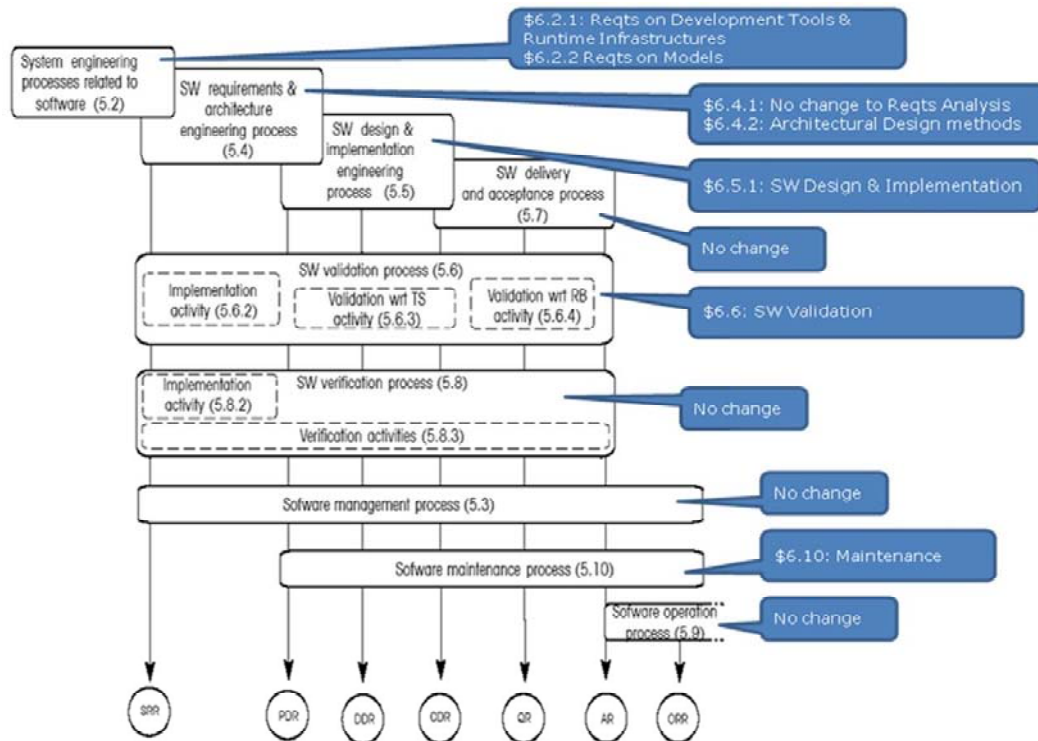


Figure 4-1: Areas of ECSS-E-TM-40-07 – Volume 1A Impacted by SMP

This section covers the identification how SMP supports the activities in ECSS-E-ST-40C and goes on to show where the technical aspects of SMP are able to provide the artefacts required throughout the E-40 lifecycle.

The aim here is to use, as far as practical, the language of the simulator developer and user when describing how SMP support the ECSS-E-TM-40-07 lifecycle.

4.2 ECSS-E-TM-40-07 Phases

This section contains references to SMP concepts and artefacts. For their definition the reader is directed to ECSS-E-TM-40-07 Volume 1A Annex A. To assist developers from traditional backgrounds a comparison is provided in Table 4-1.

4.2.1 System engineering processes related to software

This activity includes the selection of tools and methods to be used for the project. Decisions are made based on a variety of requirements and constraints.

In order for the project to benefit from SMP a compatible set of development tools together with a selection of suitable Runtime Infrastructure should be selected.

Candidates for this process must meet the requirements in section 6.2 of ECSS-E-TM-40-07 Volume 1A. This defines requirements on Development Tools that ensure tool outputs are compliant with Runtime Infrastructures that support SMP.

Particular sections of ECSS-E-TM-40-07 Volume 1A relevant to SMP are:

- Clause 6.2.1.1 covers the creation and modification of catalogues, configurations, packages, assemblies, schedules and generated code.
- Clause 6.2.1.2 states the requirements on Runtime Infrastructures in terms of loading a simulation, performing model initialization, updating (scheduling and execution) and provision of services.

ECSS-E-TM-40-07 Volume 1A also, in section 6.2.2, identifies simulator model specific requirement in areas such as fidelity, performance and portability.

Many of these types of requirements can be derived through prototyping involving the use of Tools and Infrastructure to generate refinements to the requirements.

The results of these activities are compiled into the project Requirements Baseline and subject to Software Requirements Review. Decisions made based on the prototyping are documented in the project DJF. This activity should be documented in the project Management Plan.

4.2.2 Software management process

There are no specific requirements imposed by the use of SMP on the project Software Management process. However SMP refers to the products of simulator development in specific terms such as catalogues, assemblies, configurations, packages and schedules, and therefore the management process needs to adopt these terms.

For example, workpackage descriptions should include the SMP artefacts by type and, importantly, all artefacts need to be assigned Configuration IDs.

4.2.3 SW requirements and architectural engineering

SMP is used extensively in this activity:

- definition of models and interfaces (internal/external) to be stored in the catalogue
- definition of types in use (define the simulation “State Vector”) to be stored in the catalogue
- groupings of type implementations (from the catalogue) into packages for use in the assembly
- instances and specialisations of models used to build a simulation and defined in the assembly
- model dynamics giving invocation timings, entry points, reference to instances in the assembly and grouping of tasks and stored in the schedule
- error handling etc as supported by the run-time infrastructure and supplied to developers through SMP –compliant tools
- definition of parameter values for model instances stored as configurations

In particular SMP-compliant tool outputs will be used in the Architectural Design and the Design Justification:

- catalogues to define models and types
- assemblies and schedules to show implementation

Runtime Infrastructure features are used to meet requirements and to impose design constraints.

The key PDR review item is the architectural design (SAD/ADD plus supporting design justification in DJF). Output from the Tools will figure prominently in the presented design hence SMP also impacts on the reviewers.

4.2.4 SW design and implementation

Details of the design and implementation are provided here, they are largely a detailed elaboration of the Architectural Design artefacts with supporting generated code and details of test procedures and expected results.

- catalogues and packages need to be fully populated with details and default values
- model states checked for coherence and consistency (e.g. radians where degrees are expected)
- type matching from Code Generator to the production of binary
- inter-model interfaces and data dependencies
- correct assembly (model instantiations)
- scheduling details – frequency and order of updates
- non-functional requirements – performance, resource usage, reliability ...

The DDF contains all these artefacts and are subject to formal review, The consequence of using SMP compliant tools in the Design phase is that all technical reviewers will require a level of understanding of SMP in order to perform a suitable review on the generated catalogues and assemblies.

For example, at CDR the design needs to be reviewed for consistency, completeness and correctness. If the design tool is used by reviewers is it sufficient to accept the tool's own consistency checks or is assessment via an independent tool required. Alternatively listings of the artefacts could be reviewed in which case a strong knowledge of SMP is required of the reviewer.

4.2.5 SW validation

During early project phases prototyping is used to refine requirements and for design proving exercises. SMP is designed to support the full lifecycle hence it is expected that the SMP process and creation of the relevant artefacts is also done for prototypes. References to these artefacts will be used throughout readiness reviews, tests specifications and verification activities.

Integration activities involve the configuration of tested models to run with stub models which are then replaced later when other tested components are available. SMP catalogues and assemblies can be used with stub models without any impact since the models implementation details are abstracted.

Configuration control is key during validation, SMP supports this by providing a version attribute to all its documents.

Testing makes extensive use of scripts which reference objects and methods. Use of SMP will be evident at this level. Test Specifications are produced (SVTS) and reviewed at CDR. Test procedures are reviewed at TRR (Test Readiness Review) which is the precursor to factory Testing.

4.2.6 SW delivery and acceptance

SMP has no impact in this area as the delivery process is the same for whatever artefacts (files, documents etc) constitute the deliverable items. Acceptance is performed on the system as a whole in its operational environment and this at too high a level for SMP to be visible.

It is worth noting that EIDPs (End Item Data Packs) are likely to refer to files and filetypes that are specific to SMP (e.g. Packages & Assemblies). SMP provides for *Bundles* that support the collection of artefacts required for a system delivery together with data defining the installation dependencies.

4.2.7 SW verification

SMP has no direct impact in this area.

4.2.8 SW operations

SMP has no impact in this area however it worth noting that simulation operators will be exposed to certain impacts of SMP in the form of log messages and other cases where reports are produced using terminology such as “catalog” and “assembly”.

4.2.9 SW maintenance

ECSS-E-TM-40-07 Volume 1A Clause 6.10.1 requires consistency between Catalogues and implemented code. It is also an issue for maintenance that Tools and Infrastructures are likely to undergo evolution and updates, it is important for these to be controlled for consistency and backwards compatibility.

Maintenance documentation needs to be kept in step with the tools, and even the version of SMP, in use as well as the project specific developments. It is a requirement on all projects that maintenance must be possible by staff other than the developers.

Hence engineers tasked with post-delivery maintenance need to be instructed how to implement changes and updates through the use of the SMP toolset and the subsequent generation and re-deployment process. It is not acceptable to apply quick patches to generated code.

4.3 ECSS-E-TM-40-07 artefacts provided by SMP

4.3.1 SMP as a tool chain

The main SMP features are briefly described as:

- Catalogues – definitions of the simulator structure/architecture in terms of models, types, classes, etc – note this is independent of model behaviour

- Assemblies – definitions of architecture in terms of model objects/instances and their interfaces through connections between inputs and outputs. This aids understanding for space engineers
- Schedules – dynamic properties of the assembly in terms of model invocations
- Configurations – default conditions for a simulation, such as model field values and the collection of implementation elements (types etc) into packages

These terms are further defined in ECSS-E-TM-40-07 Volume 1A section 3.2 and summarized in Annex A of the same document.

Suitable tools are utilised to produce the SMP products (catalogue, assembly etc) thereby allowing designers and developers to work in the most effective manner without being encumbered with SMP implementation details. In order for tools to be SMP compliant they should also support the various techniques also defined in ECSS-E-TM-40-07 Volume 1A such as containers, aggregation etc.

The selection of an SMP compliant toolset ensures that automated functions such as generation, validation and compilation are SMP compliant and therefore compatible with each other.

The collection of artefacts, created by the user and the toolset, combine to form the Design and Implementation Database, which can be aggregated into a bundle to form a coherent delivery.

The following diagram shows the SMP process in terms of SW development, from design to execution, the user (designer, developer, integrator or tester) does not need to be concerned with certain lower levels of this process, those activities performed by the user are identified by a user icon:

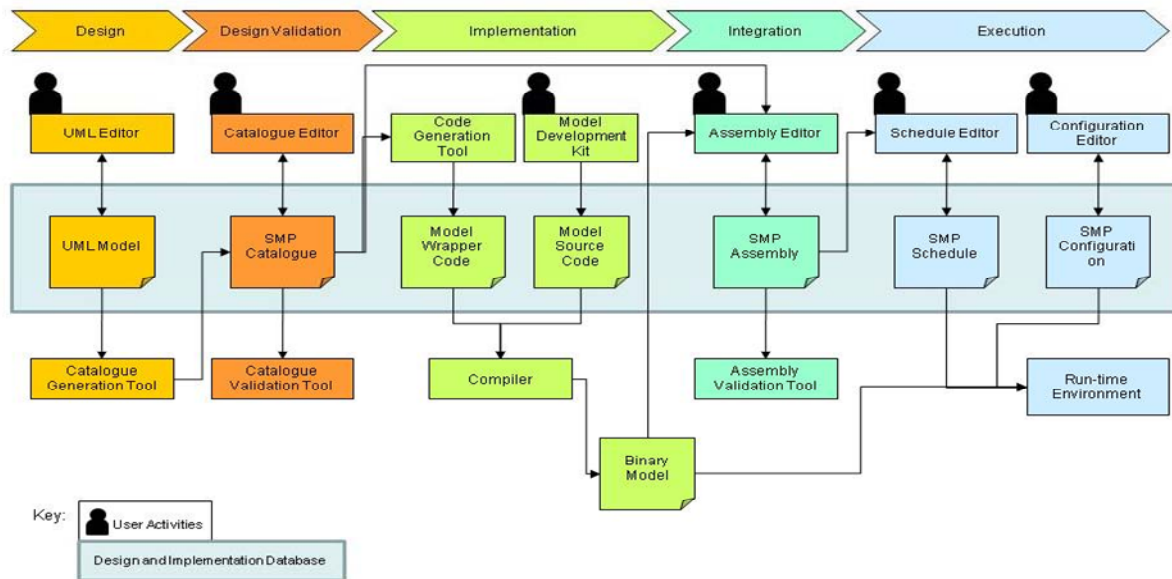


Figure 4-2: SMP Tool Chain Overview

Design activities involve the user creating UML model designs. For SMP the selected UML modelling tool will produce UML2 compliant models from which the Catalogue Generator can create SMP catalogues. EGOS-MF is an example of such a tool.

The user modifies catalogues using a suitable tool that also creates catalogues to be validated for compliance. Compliant catalogues are used with a Code Generator to produce SMP wrapper code ready for compilation

Implementation is performed by using a suitable IDE (such as Eclipse) the user develops model source code that, together with the generated wrapper code, is submitted to the compiler for the production of binaries objects.

The user employs an assembly editor to create SMP assemblies from the SMP catalogues created during the design phase.

To create the necessary execution files, the user makes use of suitable tools to create SMP schedules and SMP configurations.

Also, not shown on the diagram, to prepare for delivery the user pulls together the necessary SMP artefacts into bundles.

Clearly expertise of the SMP details is an essential part of the simulator development team, however it is important that such expertise is not required of every designer, developer, integrator and tester.

4.3.2 SMP artefacts and the lifecycle

This section identifies the simulator software artefacts which are delivered by the simulator software supplier for the different reviews defined within the ECSS-E-ST-40 software life-cycle process.

An example of a toolset supporting the SMP process is:

- EGOS-MF combined with a suitable UML modelling tool (such as MagicDraw):
 - Definition of simulation models
 - Creation of SMP catalogues
 - Creation of UML models
 - Document generation for SDD and ICD
- Simsat4 MIE combined with a suitable IDE (such as Eclipse and the GCC compiler):
 - Catalogue creation and validation
 - Code generation
 - Generation of binaries
 - Production of Assemblies
 - Production of Schedules

Execution of the simulation by Simsat4 used the binaries, schedules and assemblies created by the above.

Table 4-1:SMP Artefacts in the Lifecycle

SMP Artefact:		Catalogue	Assembly	Schedule	Configuration	Package
Phase	Doc					
SRR	RB	SMP is not design to provide significant input to the RB, other than supporting rapid prototyping			Identification of acceptance test scenario data sets	
PDR	DDF	Definitions of: Model classes, interface types, inheritance, data-model/ persistence, model metadata Model management: composites. containment Services to be used at run-time	Simulator composition: Definition of build for various scenarios Model instantiations Model dependencies	Performance: Frequency and order of invocations	Identification of system test scenario data sets	Hierarchal management of catalogue entities. Use in conjunction with Assemblies, schedules and configurations.
	SAD	Logical Model: Definition of models, interfaces and entry point	Architectural design: Models, composites, interface implementations			
CDR	DDF/ SAD	Details of model properties Model dependencies Platform constraints (binary generation) Model (compilable) templates Binary packaging	Instantiation properties – initial/default values Model integration details	Performance: Frequency and order of model invocations	Identification of integration test scenario data sets	Hierarchal management of catalogue entities. Use in conjunction with catalogues and assemblies.
QR	DDF/SAD	Tested source code, binaries, packages	Tuned/tested assemblies	Tuned schedules	Tested configurations (referred to by test results)	
AR	DDF/SAD	Final version of all artefacts, source, binaries & packages delivered in Bundles			Tested configurations (referred to by test results)	

4.4 Comparison of SMP artefacts to traditional methods

The list of SMP artefacts given in ECSS-E-TM-40-07 Volume 1A Annex A is given below together with a description of the corresponding development artefacts that are familiar to developers of traditional and legacy simulators:

Table 4-2: Simulator Artefacts

SMP Artefact	Brief Description	Traditional equivalent
Catalogue	A catalogue contains namespaces as a primary ordering mechanism, where each namespace may contain types. These types can be language types, which include model types as well as other types like structures, classes, and interfaces. Additional types that can be defined are event types for inter-model events, and attribute types for typed metadata. The language for describing entities in the catalogue is the SMP Metamodel, or SMDL.	Class/model header files
Configuration	A configuration supports the specification of arbitrary field values on component instances in the simulation hierarchy. This can be used to initialise or reinitialise the simulation.	Specific to run-time infrastructure, examples include Data Dictionary, Initial Conditions File, default database, start-up breakpoint etc..
Assembly	An assembly describes the configuration of a set of model instances, specifying initial values for all fields, actual child instances, and type-based bindings of interface, events, and data fields. An assembly may be independent of any model implementations that fulfil the specified behaviour of the involved model types, i.e. an assembly may specify a scenario of model types without specifying their implementation. On the other hand, each model instance in an assembly is bound to a specific model in the catalogue for its specification.	Make files
Schedule	A schedule defines how entry points of model instances in an assembly are scheduled. Tasks may be used to group entry points. Typically, a schedule is used together with an assembly in a dynamically configured simulation	Schedule file(s) specific to the run-time infrastructure that define model entry points, calling frequency, time offsets etc.
Package	A package holds an arbitrary number of implementation elements. Each of these implementations references a type in a catalogue that shall be implemented in the package. In addition, a package may reference other packages as a dependency.	A level of aggregation not previously used in a formal way
Model Implementation	A model implementation implements the behaviour of the model. The model implementation may be delivered by the simulation model supplier at either source code or binary code level.	The same – source code, binaries and data files that fully define the behaviour of a model

Getting started with SMP

5.1 Introduction

This section provides an “SMP in a Nutshell” overview of the basic steps involved in creating an SMP simulator that will execute.

This example provides the user with the minimum steps/amount of code in order to get an SMP simulation running. The example has a single model with an entry point that logs the message “Hello World”.

5.2 Define a catalogue

An SMP Catalogue is a document that describes the various components in an SMP simulation. The most important of these components and the most basic element required for a simulator to function are Models. Models implement the behavioural aspects of the simulation in the form of C++ code. Models are also associated with other SMP elements such as interfaces, fields, operations, namespaces, entrypoints etc which provide the mechanisms required to relate models to each other and execute their behaviour. Models and their associated elements are described in Section 8.2.

The catalogue is one of the key artefacts of SMP as it describes all of the elements that make up a simulator.

The first step in creating an SMP simulation is to create a catalogue containing the specification for the models you wish to use in your simulator. Figure 5-1 shows the HelloWorld catalogue being used in our simple example. This catalogue contains the following elements:

- A namespace – provides a way to logically organise models and other elements(e.g. by sub-system) . The namespace “SMP_Handbook” is used in the example.
- A model – models are the elements that contain the different behaviours of a simulator. The model name “HelloWorldModel” is used in the example.
- An entry point – a simple function with no return value and no arguments that is used to trigger model behaviour from the SMP infrastructure. An entrypoint called “SayHello” is used in the example.

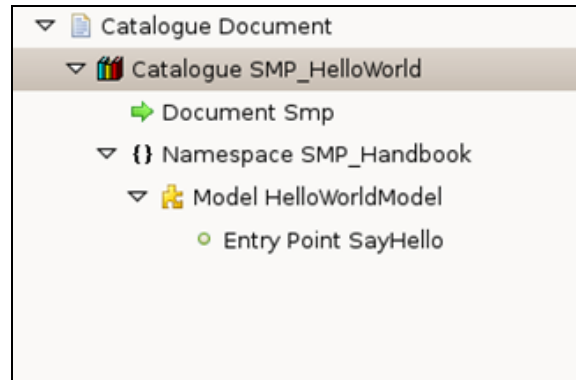


Figure 5-1: HelloWorld Catalogue

5.3 Generate the SMP wrapper code

By “generate SMP wrapper code” we mean that an SMP code generator uses the information in the catalogue to generate the SMP “boiler-plate” or “plumbing” code. It is expected that developers will usually have this kind of code generated for them allowing them to concentrate on implementing the model behaviour code.

```

// Initialise
void HelloWorldModel::_Initialise(void)
{
    // . . . . . Initialise private fields . . . . .

    // . . . . . Setup EntryPoints . . . . .
    //
    // EntryPoint: SayHello
    SayHello
    = new ::Smp::Mdk::EntryPoint( "SayHello",
                                "An endpoint - a function with no parameters and no return value.!",
                                this,
                                &HelloWorldModel::_SayHello );
    // Use existing implementation to manage Entry Points
    this->AddEntryPoint( SayHello );
}
    
```

Figure 5-2 Generated EntryPoint Set Up Code

Figure 5-2 shows code that has been generated using an SMP tool from the details entered in *SMP_HelloWorld* catalogue. The form of the generated code is not specified in the SMP standard and therefore generated code will be different between one tool and another. For example, in Figure 5-2 the code generated uses the MDK. The MDK provides a simplified API for SMP but it is not covered by the standard so it is optional whether a code generator tool uses or not.

The standard does, however, say that the tool must respect the specification provided in the catalogue. This ensures that models generated from different tools will work together. In this example we see code that creates an SMP Entrypoint object for the *SayHello()* endpoint defined in our catalogue. Later we will see how an SMP infrastructure is able to use this endpoint to execute something.

5.4 Add custom model code

In addition to generating SMP wrapper code, using the catalogue an SMP code generator has enough information to provide the “shell” of the implementation code. Typically the wrapper and shell code would be contained in two different files to keep this separation clean. Therefore the in most cases the model developer only has to be concerned with one simple implementation file where they code the behaviour of a model.

```

namespace SMP_Handbook
{
    Smp::Services::ILogger* logger;

    // ..... Entry Points .....
    // .....

    // --OPENING ELEMENT--HelloWorldModel::_SayHello--
    // Handler for Entry Point: SayHello
    void HelloWorldModel::_SayHello()
    {
        // MARKER: OPERATION BODY: START
        logger->Log(this, "Hello World!");
        // MARKER: OPERATION BODY: END
    }
    // --CLOSING ELEMENT--HelloWorldModel::_SayHello--|

```

Figure 5-3 Custom HelloWorld Entrypoint Code

Figure 5-3 shows an example of a HelloWorld Model implementation, generated using a tool, of the SayHello() entypoint. This is the only code the developer writes.

5.5 Define a package

A Package is the SMP mechanism for defining a simulator component. An SMP Package contains the implementation elements (either model types or instances) required in order to use an SMP component. This component would typically be a sub-system or model component.

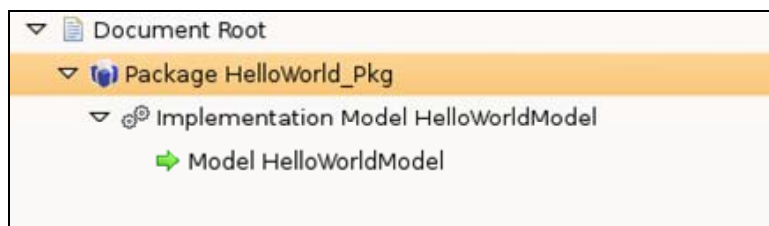


Figure 5-4 HelloWorld Package

5.6 Generate a simulator component

Having defined a package for the HelloWorld component we can use an SMP tool to generate the SMP component code and a build system for the component automatically.

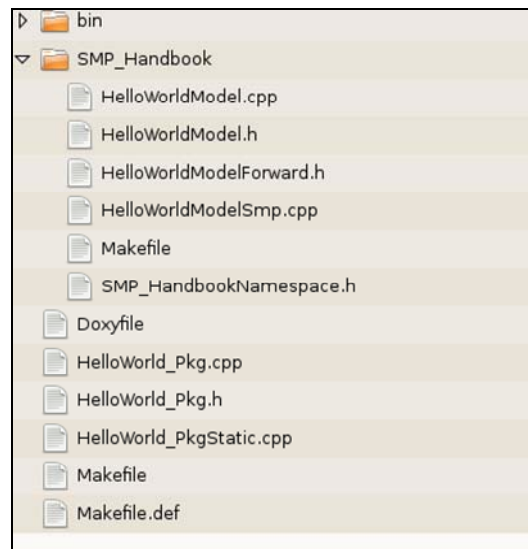


Figure 5-5 Generated HelloWorld Component

Figure 5-5 shows examples of all of the elements generated so far. In this example our SMP build tool will create an SMP HelloWorld_Pkg.so (i.e. shared object or dynamically loaded component library) file that can be loaded by an SMP simulator infrastructure. The organization of this code into various files and the make system used in order to compile it is not defined by the standard and is therefore dependent on how the SMP tool used chooses to do this.

5.7 Define an assembly

Although we have our model component ready there are some other elements to create to get the simulator to run in an SMP infrastructure.

Typically an SMP Assembly will be created. Whereas the catalogue formed the specification for our model from which we were able to generate code, the assembly describes a configuration or a deployment of the models in a simulator.

An SMP assembly describes the number of instances of the model we need. For example we might have defined a gyro model and we want two instances of it in our simulator variant, in another simulator variant we made need three. This can be achieved by changing the assembly only. Where we have more than one type of model instance the assembly also describes how these instances are connected together for a particular variant of the simulator.

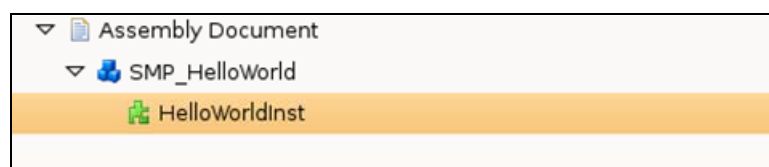


Figure 5-6 HelloWorld Assembly

Figure 5-6 shows a single HelloWorld model instance being declared in an assembly.

5.8 Define a schedule

Additionally instead of hard coding a schedule for executing the model instances in our simulator we can use an SMP Schedule. In our simple example we have defined:

- A *Task* - describes one or more activities to be carried out while the simulator schedule is executing.
- A *Trigger* - used in tasks to define what actually needs to be executed. In our example a trigger is defined that will call the SayHello() entrypoint on the HelloWorld model instance declared in the assembly.
- An *SMP Simulation Event* - defines the element that is scheduled so for instance in our example "Hello_SimEvent" is defined to run our "SayHello" task every second (and therefore call the SayHello() entrypoint).

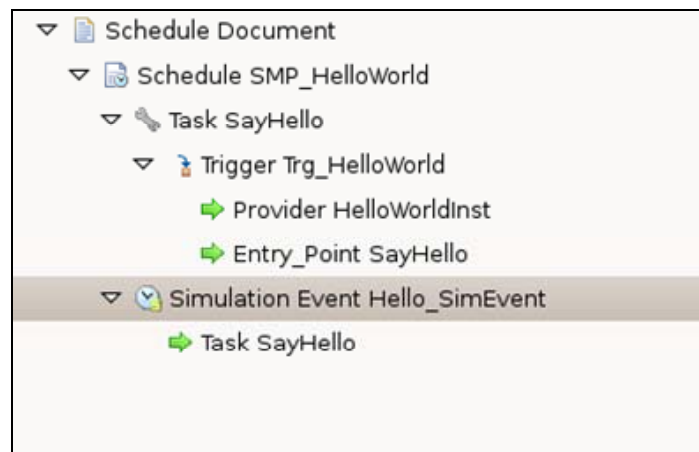


Figure 5-7 HelloWorld Schedule

Schedules are discussed in more detail in section 8.5.

5.9 Launch the simulation.

This step is dependent on the SMP infrastructure you choose to use. It usually involves creating a configuration for your SMP simulator that references the Catalogues, Assemblies, Schedules and binary files associated with the simulator so that the SMP infrastructure can load and execute the model code appropriately. How this deployment and loading happens is not governed by the E-40-07 standard and is therefore specific to each SMP infrastructure. What happens when the simulator executes is specified by the standard. Models and infrastructures then only interact through the interfaces specified in the standard and execute according to a defined simulator and model life cycle.

6

The Modelling Tasks

6.1 Introduction

This user guide presents SMP through a set of modelling tasks that are familiar to anyone who implements simulators and models in the space industry. The tasks are intended to describe a set of activities that need to be performed and problems that need to be solved during the process of creating or modifying a simulator.

The content of the tasks is free from any discussion or description of a particular technology, programming language or simulator infrastructure.

The main tasks covered here relate to the simulator processes of design by Architects, implementation by Developers and testing by Integrators.

The SMP tutorials that follow later in this user guide are presented in the context of these tasks so that the reader has familiar reference for the problem being addressed in any particular tutorial using SMP techniques.

6.2 Simulator architect tasks

6.2.1 Identify major sub-systems and their components

6.2.1.1 Aim

To breakdown the simulator in to its major parts based on the design presented by the **spacecraft** architecture and the requirements of the simulations that need to be performed. The main simulator components, (which will mainly, but not exclusively, be models), are identified together with the major interfaces between them. Implementation details are not relevant at this stage.

6.2.1.2 Description

Definition of the architecture in terms of the major systems and subsystems. Very often the top-level comprises of "Spacecraft", "Environment" and "Ground". The Spacecraft often comprises the conventional spacecraft sub-system such as Power, AOCS, Thermal, Payload etc. Additional simulator specific subsystems may be added dedicated to, for example, TC handling or TM generation.

Decomposition to the level of models, such as a sensor, without defining the particular instances of the models, i.e. how many sensors.

Interfaces between the components are identified in terms of control and/or data.

Mapping from actual sub-systems and components to simulation sub-systems and components will not, in general, be a simple one to one mapping but will be driven by the needs of the simulation, e.g.:

- if the simulation is to run real on-board software (either in an emulator or by re-compiling it for the simulation host) then a function such as TC handling will be partially mapped to the simulated on board software and partially to a model of the TC decoder hardware which is accessed by the on board software model as a model of an I/O device. Conversely if the on board software is not to be run in the simulator then the whole of the telecommand handling may be modelled in a single model;
- some physical units may perform multiple logically independent functions (e.g. an on board computer may implement the attitude control laws, handle TCs, generate TM packets and perform thermal control). Some functions may be spread over several physical units (e.g. attitude control, telecommand handling). The simulator architect will separately identify both the physical units and the functions that they perform.

Additionally the architectural breakdown of the spacecraft into sub-systems and models will consider the required componentisation of the simulator to meet the requirements of sharing and re-using simulator components.

Similarly, if the architecture is being based on a prescriptive reference architecture this will constrain or dictate the interfaces and granularity of the components of the simulator and therefore what elements can be exchanged.

The result is a coherent high-level of components and interfaces described, typically, in UML.

6.2.1.3 SMP aspects

- A Catalogue Generation Tool is used to generate the Catalogue(s) that are XML documents containing SMDL;
- The top level components are created as *namespaces* which contain types such as *model*, *class*, *event*, *interface* etc;
- Dependencies are generated as *containers* (composition of types) and *references* to other models;
- Methods are described as *operations*
- Data is described as *fields*, note that internal data is not breakpointed;

A tutorial for this activity is provided in section 7.1.

6.2.2 Identify data and control flows

6.2.2.1 Aim

Elaboration of components and interfaces identified above with focus on the type of interfaces. Focus is placed on the relationships between the components (systems/subsystems/models) and dependencies between them.

Decide how data is exchanged and the event notification mechanisms to be used.

6.2.2.2 Description

Identify the control of one model by another, data transferred between models, sequencing of model invocations and functionality in the spacecraft design that has impact on the modelling in the simulator.

A typical spacecraft behaviour calls for a variety on interfaces, such as:

- Key on-board events such as command pulses and clock ticks
- State change events such as power switching and thrusters firings
- Data driven events such as on-board data bus transfers
- Continuously changing properties such as attitude and orbit parameters, environment impacts and thermal values

Interfaces with the run-time infrastructure are also analysed, this covers:

- Handling of simulation state changes
- Calls from the scheduler to model update services
- User input such as failures and value changes
- Model data for display and messages for the simulation log
- Data to be saved when breakpointing and reset during a restore

Each interface is analysed and the design produced to reflect the nature of the interfaces between the models:

- Interface Based design – where components interact with other just through operations (such as update, get/set), no data is passed and all implementation details and data of a model are hidden.
- Typical example could be closed-loop AOCS acquisition of sensor data and setting torque demands on thrusters models.
- Dataflow-based design – where data is updated by one model and passed to another model.
- Collection and formatting of TM is a typical example of this.
- Event-based design – where events are non-deterministic or not cyclical, typically the knock-on effects of an event in one model has impact elsewhere.

- Event-based design feature events with arguments and “publish/subscribe” type relationships.
- A typical example is power distribution where changes in equipment state anywhere on the spacecraft are sent to the power model to re-balance the power state.

The outcome is that interfaces between the sub-systems and models have been identified and documented including their data, protocols and frequencies.

Scheduled events are specific control flows and are covered below in section 8.5.5.

6.2.2.3 SMP aspects

Details are added to catalogues and assembly files;

- Interface based design features interfaces and, for external interfaces, references. Interfaces can have properties for standard operations such get/set.
- Dataflow design features the definition of fields and their attributes such as input or output and visibility
- Event-based design involves an event source (raiser of event) and event sink (recipient) and event type which defines the data associated with the event

A tutorial for this activity is provided in section 8.3.

6.2.3 Incorporate existing SMP models

6.2.3.1 Aim

To obtain an existing SMP model (or sub-system) from another source such as a model library or another simulator, and incorporate it into a new SMP simulator.

6.2.3.2 Description

Re-use such as this can be performed in 2 ways – re-use of the source code and/or re-use of the binary objects

- Re-use of source covers the case where the implementation being re-used was not originally design for integration with the current simulator under design but where its implementation is sufficiently compatible to be directly re-used as is.
- The source package needs to contain the original catalogue, assembly and schedule files as well as the actual source files.
- Re-use of binaries is possible when the source code is not available due to various reasons such as IPR

Organisations can re-use an existing simulator architectural design (through an SMP catalogue) but have the added flexibility to re-

implement models as required for a new simulator with a much reduced effort.

Together with the binary files, the catalogue, assembly and schedule files from the original implementation are needed.

6.2.3.3 SMP aspects

Linking/merging and modification of SMP files:-

- catalogues
- assemblies
- schedules

These are text files containing XML/SMDL and can be edited with SMP compliant tools or they may be edited with simple text editors.

SMP compliant tools are used to validate the catalogue, assemble and schedule files.

6.2.4 Incorporate legacy models

6.2.4.1 Aim

To re-use a non-SMP model or sub-system implementation and incorporate it into a fully compliant SMP simulator.

It is assumed that:

- The scope, functionality and dependencies of the legacy models match the requirements of the new simulator.
- Any non-SMP dependencies are known not to restrict the re-use of the model(s).
- There are no technology or platform incompatibilities that restrict re-use of the legacy models.

6.2.4.2 Description

Create a simple model on the SMP catalogue to represent the legacy model

Create operations to wrap the equivalent operations in the legacy model

Declare input and output fields for each entry point representing all input and output data for the legacy model

In particular create an Update event for the main legacy model update function, ensuring that data requirement of the model update are coded in the Update function (which has no arguments)

Typically, when integrating a legacy model into an SMP environment, the following guidelines are useful:

- Initialise the model in the SMP constructor code segment

- Perform all model update functions within a single SMP Update() if possible, otherwise multiple model update entry points will need to have separate SMP Update events and schedule file entries.
- For method on the legacy model, to be used from other models, create an equivalent SMP interface
- For each of these SMP interfaces code the following steps into the SMP generated skeleton code:
- Copy the data from SMP fields to the equivalent legacy model input variables
- Execute the legacy model function
- Copy the data from the legacy model output variables to the equivalent SMP fields
- Any termination functions are added to the SMP destructor code segment.

6.2.4.3 SMP aspects

Care needs to be taken when the legacy model has its own state that is not mapped to SMP state variables. This has an impact when breakpoints are made and reset.

There is no single method within SMP for handling this case, it may that the best solution is to create handlers for breakpoint and restore events and save/load model state data separately from the main breakpoint files.

This commonly happens in the case of software emulation of on-board software.

6.2.5 Decide how to connect simulation components

6.2.5.1 Aim

Determine how the connections between the various simulation sub-systems and components are to be established.

6.2.5.2 Description

Each data and control flow is analysed to determine how it shall be implemented. With regard to SMP this means deciding whether to use;

- Models interoperate in an adhoc manner based on when things happen during simulator execution
- Models share data based on computations performed at specific points during the simulator execution
- Models collaborate using a set of logic partitioned throughout different sub-systems and models performed according to a schedule.

The outcome is the mechanisms to be used to establish the connections between simulation sub-systems and for the internal connections between sub-system components in the simulation will have been decided upon.

6.2.5.3 SMP aspects

Details are added to catalogues and assembly files;

- Model interfaces (see Interface-based Design)
- Dataflow – exchange of data between models using input and output fields(see Dataflow-based Design)
- Events - to provide a publish and subscribe mechanism to exchange data between models based on what happens as the simulator executes rather than via schedule (see Event-based Design)

A tutorial for this activity is provided in section 8.3.

6.2.6 Determine model scheduling

6.2.6.1 Aim

This is a more specialised case of the previous section. Here we concentrate on model control that is based on the event that a particular time has been reached, whether it is simulation time, simulated epoch, Zulu time or mission elapsed time.

6.2.6.2 Description

Each simulation framework has a scheduling function at its core. Models need to be updated at regular frequencies, at specific times or at “time now plus an offset”.

Identify which events need to be scheduled, and how should they be scheduled (e.g. by the infrastructure scheduler, i.e. SMP scheduler, or by the some other scheduler such as an emulator's internal scheduler which may be more appropriate for events that need to be executed after very short delays)

For scheduled events, no matter how they are scheduled, determine what, if any, data has to be associated with the event and how this data should be handled. For those events for which it is decided to use the SMP scheduler then the abstract events need to be mapped onto SMP events and their associated model entry points.

If the handler of a scheduled event is in a different component to the originator then consider how this is to be achieved. Does the originator receive the call from the scheduler and then call the receiver or does the originator request the receiver to schedule an event.

The dependency of model updates on each other must be determined. For example AOCS closed loop control at a minimum consists of models of sensors, the control laws and actuators. An environment model would be used to process actuator outputs and provide updated inputs for the sensors. Hence a cyclic scheduled task can be used to ensure that the model updates for sensors,

control laws and actuators are performed in that order, and that the environment is then updated accordingly.

Other specific timed events are defined, such as thruster duty cycling, where the thruster valve is closed at a fixed time after opening.

Specialised signals also need identifying. For detailed bus modelling the broadcast pulse might be seen as a regular schedule events but its timing is significant in the context of other bus traffic that is likely to be asynchronous.

For pulse trains (e.g. clocks) should there be a notification for each pulse (or pulse edge) should there just be a indication of whether or not the pulse train is active. Low frequency pulse trains can best be handled as individual pulses, but this is not practical for high frequencies.

6.2.6.3 SMP aspects

Scheduling details are stored in schedule files

- When using interface-based or event-based design the schedule events are defined as triggers that have associated model entry points.
- When using dataflow-based design simulation events are defined as transfers that have field links
- Simulation events can be defined as tasks which group together triggers and transfers that are related to other and need to be performed in a particular order

A tutorial for this activity is included in section 8.5.

6.2.7 Decide how to configure simulation components

6.2.7.1 Aim

Determine how the various simulation sub-systems and components previously identified will be configured.

6.2.7.2 Description

Each component needs to have defined for it its instantiations, set-up data and how that impacts on other components

Configuration data can be set-up in the source code (hard-coded), provided by another component, through the use of data files or by using run-time infrastructure facilities such as initial conditions files, breakpoint files or scripts.

6.2.7.3 SMP aspects

- Field default values can be defined in the catalogue
- Model instances defined in the assembly file may define initial values for fields. (see 8.4.2)

- An SMP simulator may use an SMP Configuration File to load initial values into the fields of model instances.

6.3 Model developer tasks

6.3.1 Implement new sub-systems and models

6.3.1.1 Aim:

To create simulation models from the specifications provided by the architect, using the SMP standard and ensuring full SMP compliance.

To specify the runtime elements of a simulator including their connectivity, attributes, methods and any utilities that are required for them to function.

To determine the internal state that has to be maintained for each simulation component.

Note that, in effect, SMP components contain models and sub-models that refer to the breakdown and dependencies of software units. It is desirable in the Space Simulation domain to retain the engineering concepts of systems and subsystems in order to relate the software tasks to the traditional spacecraft delineation. For example the simulator is likely to contain a model of the Power subsystem, and sub-models of this are defined for the various equipment types within the Power subsystem such as battery, solar array etc.

6.3.1.2 Description

Detailed iteration of the Architect's specification:

- Catalogue – ensure all data declarations are complete with required attributes and values assigned; check that all links are consistent, this applied to fields, operation and events
- Assembly – check instances are correct in cardinality and in their model types.
- Schedule – ensure schedule dependencies, frequencies and relative timings are correct

Several of these tasks can be performed automatically by the use of validation tools.

Use SMP compliant toolkit for code generation. This produces auto-coded declarations for the objects described in the catalogue files.

Perform code development using the generated skeleton C++ files to add project specific functionality. All model entry points need to be populated with project code:

- Behavioural code in model functions and sub-functions
- Code for actions in events

- Handling code in all declared interfaces

Compilation and linking with the assembly and schedule files together with the run-time environment

Unit testing against the Architects specifications.

6.3.1.3 SMP aspects

Generally speaking, code generation creates folders for each namespace. For each model/submodel contained therein it produces header files that contain SMP wrapper code, and C++ files containing boilerplate code for all declared types with sections for project specific code to be included.

The developer uses the toolchain described in section 4.3.1.

6.3.2 Prepare models for re-use

6.3.2.1 Aim

To modify and package an SMP model of a system, sub-system or sub-assembly that has not previously been intended for re-use so that it can be delivered as an independent item for other parties to re-use as is.

6.3.2.2 Description

The developer ensures that a fully specified model has clearly identified interfaces and dependencies that allow other to identify its capabilities so that it may be re-used.

The developer ensures that models planned for re-use are:

- Provided with a fully documented catalogue
- Specifically checked that the model dependencies are clearly identified
- Checked that the catalogue has been partitioned correctly for the scope of the components/models described
- Ensured that the catalogue, assembly and schedule files have been successfully validated by SMP compliant toolkits

The model components can then be packaged as a re-useable unit

6.3.2.3 SMP aspects

The complete set of model artefacts are packaged as an SMP *bundle*

6.4 Simulator integrator/tester tasks

The SW Validation Process (ECSS E-ST-40C section 5.6) identifies three main tasks:

- Validation process implementation - planning the validation process

- Validation with respect to the Technical Specification (TS)
- Validation with respect to the Requirements Baseline (RB)

It is important to realise that these three tasks overlap and that they can all be started to varying degrees once the Requirements Baseline is stabilised.

The SW Verification Process (ECSS E-ST-40C Section 5.8) identifies two main tasks:

- Verification process implementation
- Verification activities

As identified above, validation is the demonstration that the software meets its design specification; but note that this can be achieved by testing or analysis.

The main focus of this section of the User guide is to explain the tasks used in 'Verification activities'. These tasks comprise:

- Verify models – which involves
- setting-up the integration environment as separate from the development environment,
- building a model delivery and repeating component-level tests
- Integrate models – involving
- the systematic integration of models into the integration simulator and replacing stub models until all models are integrated with no stub or test software remaining,
- end-to-end system testing of the complete simulation.
- Verify Simulator – this is formally planned and conducted against the requirements baseline

It is acknowledged that RD-1 summarises that SMP provides little support for integration, its primary use during design and implementation. However there are aspects of SMP that provide a beneficial approach to integration and these are identified in below sections.

6.4.1 Set-up integration environment

6.4.1.1 Aim

Set up the integration environment into which model deliveries can then be made and tested.

6.4.1.2 Description

The integration environment should preferably be isolated from the development environment(s) to ensure that model deliveries are self-contained and complete. If this is not done then there is a danger that models have dependencies on the features of development environment that will not be

present in the environment in which the final system is to be run. These dependencies may be as simple as the presence or values of “environment variables” or the particular version of the compiler tool chain or libraries that are used. The use of a separate integration environment rather than re-using a development environment ensures that a consistent environment is used for integration.

6.4.1.3 SMP aspects

An SMP compliant tool-chain needs to be established as shown in section 4.3.1 above. This comprises editors, validation software, code generators and the run-time environment as provided by the selected SMP compliant simulation infrastructure.

6.4.2 Verify models

6.4.2.1 Aim

Verify the behaviour of a model in isolation.

The model’s behaviour is verified to be compliant with its Design (DDF) and with the requirements of the Technical Specification.

Produce Unit Test Procedures, Test Cases and Test Results.

6.4.2.2 Description

It is unusual for a simulator model to comprise more than one code module, but in the event of this being the case, each code module should be unit tested in a bottom-up manner.

For the purposes of model testing, the Tester should regard the SMP infrastructure as a pre-validated test tool. This means that the model can be integrated with its infrastructure environment as soon as it is useful to do so.

Each development participant delivers to the integrator (ref RD-1):

- Models source code.
- Models binary code.
- Model assemblie(s)/schedule(s).

The Integrator identifies Test Procedures that exercise all of the paths through the module. In addition to nominal activity, this must include all exception conditions.

The Integrator performs unit tests on each delivered model in the integration environment:

- Create drivers to invoke each part of the module’s interfaces;
- Configure test data to substitute any data inputs and mechanisms to take any data outputs;
- Implement stubs to respond to any external calls of the module.

6.4.2.3 SMP aspects

Models for integration should comprise of packages with :-

- Source code – generated headers, skeleton code with project specific additions
- Binaries - .so files
- Makefiles used by developers
- Sub-assembly - containing model instances, initialisation details and internal connections

6.4.3 Integrate models

6.4.3.1 Aim

To build together increasing numbers of coupled modules and in so doing verify that their interfaces are in accordance with the architecture described in the Technical Specification.

The interfaces between all models is verified to be compliant with its requirements of the Technical Specification.

Produce Integration Test Procedures, Test Cases and Test Results.

6.4.3.2 Description

Integration takes place once two or more models have been unit tested and can only complete after all models have been unit tested. Cases of integration with HWIL will normally involve a software adapter, which, once developed, would also require unit testing.

Integration in an SMP environment can be done by models delivering sub-assemblies that then have to be combined into a global assembly. The integrator then needs to add the inter-model links.

Model binary libraries are placed in the integration area.

The Integrator creates the integration version of the schedule in order to control the execution of the models under test, which is specific for that integration build, covering:

- Transfer all field links
- Trigger all model entry points

Input from Architect needed for this.

Specify run-time configuration in terms of assembly file, schedule file and binary library for the run-time infrastructure.

Due to the incompleteness of the system being integrated, the creation of stub models is necessary to represent non-integrated models and their external interfaces. Stub models are created as part of the integration and in due course replaced by the relevant model when delivered.

6.4.3.3 SMP aspects

To set-up and integration simulator, the Integrator can start by creating a Global Assembly file and a Global Schedule file.

The Global Assembly contains the top-most components defined as *namespaces*, often this will be Space Segment, Ground Segment and Environment.

In essence this will define an “empty” simulator.

As models are delivered their assembly files are incorporated into the Global Assembly, with the relevant levels of nesting obeying the system/sub-system breakdown of development.

Their transfers and entry point triggers are added to the schedule file.

At present SMP limitations mean that the activities of combining model assemblies and schedules into the Global files is a manual activity.

Assembly and Schedule validators can be used to ensure consistency and compatibility with the run-time infrastructure.

Integration of the model itself can be done by either compiling in the source code or by loading the binaries (typically shared object files)

For Integration tests to be performed the simulation must be built to a sufficient level of completeness for all paths to be tested through the model(s) being integrated. Stubs are needed for this, these are created and used as follows:

- Design the stub model to provide all the required external interfaces (Field Links and Interface Links) from the point of view of the models under test;
- Define an Entry Point in the stub model to implement the test scenario;
- Make an Assembly that contains an instance of the stub model and one instance per model under test and make the necessary connections between all model instances;
- Make a Schedule to trigger the stub model Entry Point as required;
- Create the simulation and execute the test.

6.4.4 Test complete simulator

6.4.4.1 Aim

To perform end-to-end test on the integrated simulation and, specifically, test for the first time system-level requirements.

6.4.4.2 Description

Testing concentrates on the interface between the major components (at model-delivery level) and the external system interfaces

All system inputs (telecommands, user-inputs) are tested and system outputs checked (telemetry, logs, displays).

6.4.4.3 SMP aspects

Full simulator versions of assemblies and schedules are produced. Catalogues are partitioned and final packages are defined.

A tutorial for this activity is provided in Section 11.

6.4.5 Verify simulator

6.4.5.1 Aim

Conduct end-to-end verification of the complete software simulator against the user requirements established early on in the project.

The behaviour of the complete system is verified to be compliant with the requirements of the Technical Specification. Attention is given to non-functional and system-level requirements for which testing has not been possible until integration has been completed.

Produce System Test Procedures, Test Cases and Test Results.

6.4.5.2 Description

Identify major areas of behaviour that need to be tested (Test Designs). The aim of each Test Design is to identify coherent, cohesive sets of tests that can be executed stand-alone (without requiring unrelated tests to be re-run) and repeated on demand.

Identify ways in which the simulator is to be verified (Test Procedures). In practice it is often possible to run some Test Procedures against the code before it has been fully integrated.

Identify specific scenarios or 'Test Cases' of each Test Procedure. Each test case comprises a set of inputs and a set of expected outputs, against which the actual outputs are compared.

The System Tester builds the simulator using (and therefore verifying) the build instructions.

The System Tester executes each Test Design on the simulator system, recording the results of each test.

Focus on system-level requirements such as:

- Performance – load and stress
- Scale/volume
- Resource usage
- Stability
- Durability
- Security
- Maintainability

6.4.5.3 SMP aspects

Simulator builds marking significant baselines of the integration process can be defined as SMP bundles that specify a complete self-contained simulator build. These bundles should be controlled Configuration Items which helps the configuration Control aspects of integration.

A tutorial for this activity is provided in Section 11.6.

7

The example simulator

7.1 Introduction

For the purposes of the tutorials this user guide is supplied with an example simulator. This is a fully working SMP simulator that contains all the examples used in this user guide.

SMP users are welcome to examine this simulator using their own SMP tools and are free to experiment with the models and the simulator architecture in order to better understand the examples and SMP.

The example simulator comes from the VISMP2 project that aimed to perform an industrial validation of SMP on multiple run-time infrastructures through a typical simulation project.

VISMP2 has been selected as it employs most of the SMP concepts, in particular catalogues, assemblies, schedules and packages, together with SMP services such as interfaces, properties, fields, triggers, events, transfer etc. Further details can be found in RD-1.

7.2 Functional Description of Example Simulator

The VISMP2 simulator is representative of the majority of spacecraft simulators in terms of its functionality. It simulates:

- The communications between the spacecraft and the ground station
- The on board attitude and orbit control
- The power generation and supply to on-board equipment

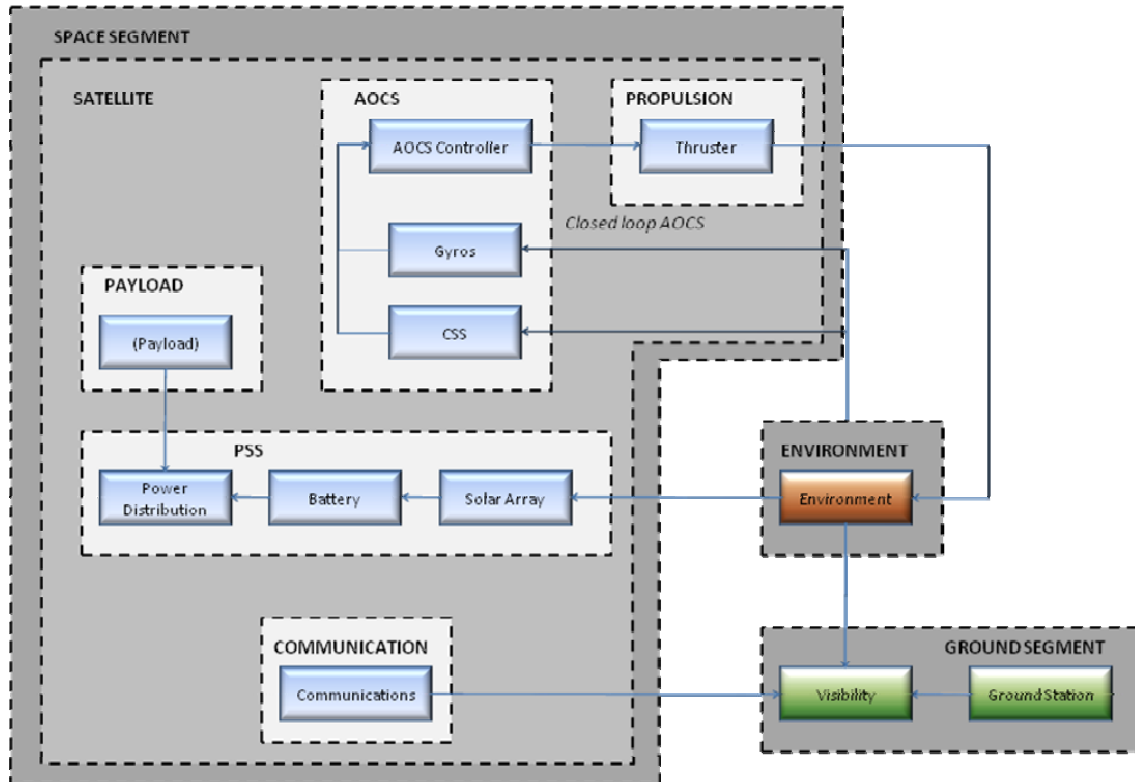


Figure 7-1: Example Simulator Overview

The space-to-ground communications are simulated by a ground station model and models of the on-board transmitters, receivers and antennae. A model of the spacecraft environment (orbit and attitude) is used to determine the visibility of the satellite by the ground station.

On-board attitude and orbit control is based around a simplified AOCs Controller model designed to perform a sun acquisition. The Controller uses inputs from models of Sun Sensors (CSS) and gyros. Its outputs are to models of thrusters. For the purposes of VISMP2 performance characteristics of the AOCs equipment are taken from SPOT5.

The AOCs equipment models, thrusters, CSS and gyro, use the environment model for spacecraft attitude data. A propulsion model exists to model the thrusters effects on the mass properties.

The Power subsystem contains solar array, storage (battery) and distribution to calculate the power availability and the power consumption by all the satellite equipment models. The solar array model uses the environment model for solar input.

A representative payload model exists as an equipment model but no payload functionality is defined or modelled.

7.3 Software architecture of example simulator

At the highest level the VISMP2 simulator consists of the 3 conventional spacecraft models:

- The space segment

- The environment
- The ground segment

Each of these is defined as a SMP model under a top-level root model of the whole simulator.

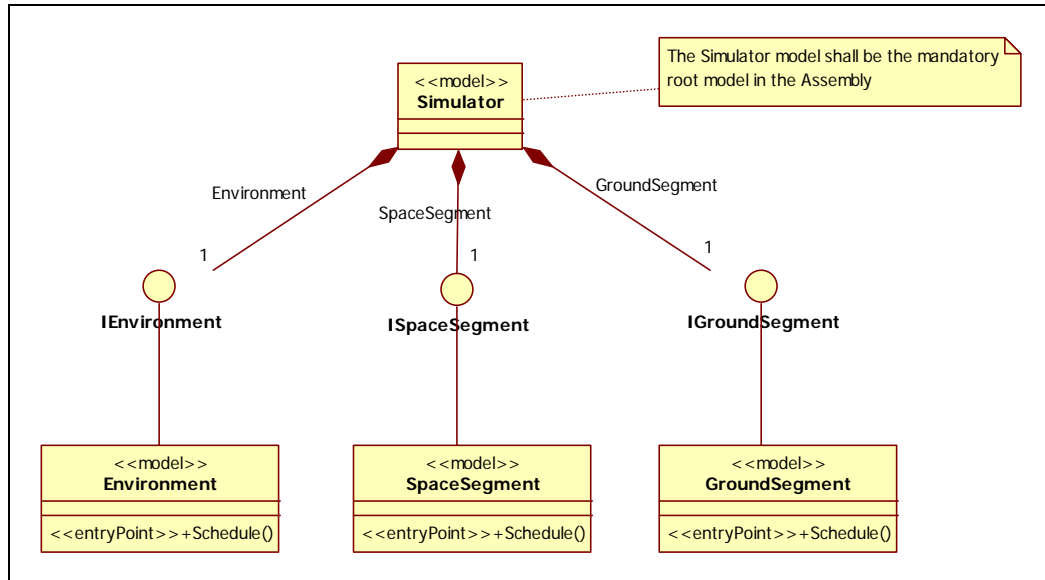


Figure 7-2: Example Simulator Main Components

The GroundSegment container contains one IGroundSegment interface, which is implemented by the model GroundSegment. It contains the ground Station and Visibility models.

The SpaceSegment container contains one ISpaceSegment interface, which is implemented by the model SpaceSegment. It contains the satellite model.

The Environment container contains one IEnvironment interface, which is implemented by the Environment model

The Space Segment contains the interface ISatellite, which is implemented by the Satellite model.

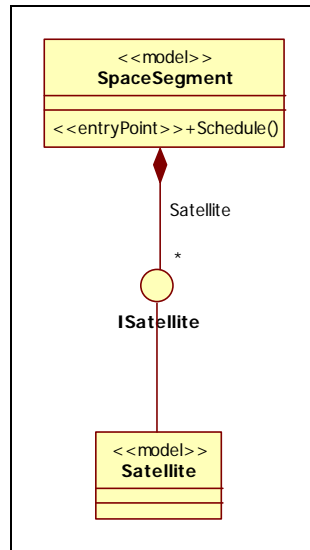


Figure 7-3: Space Segment Model

A satellite is composed of sub-systems. In the scope of this simulation, these are respectively the AOCS, the PSS, the Communication, the Propulsion and the Payload sub-system. ISubSystem has one method Update() which executes the update of the sub-system based on the current subsystem state.

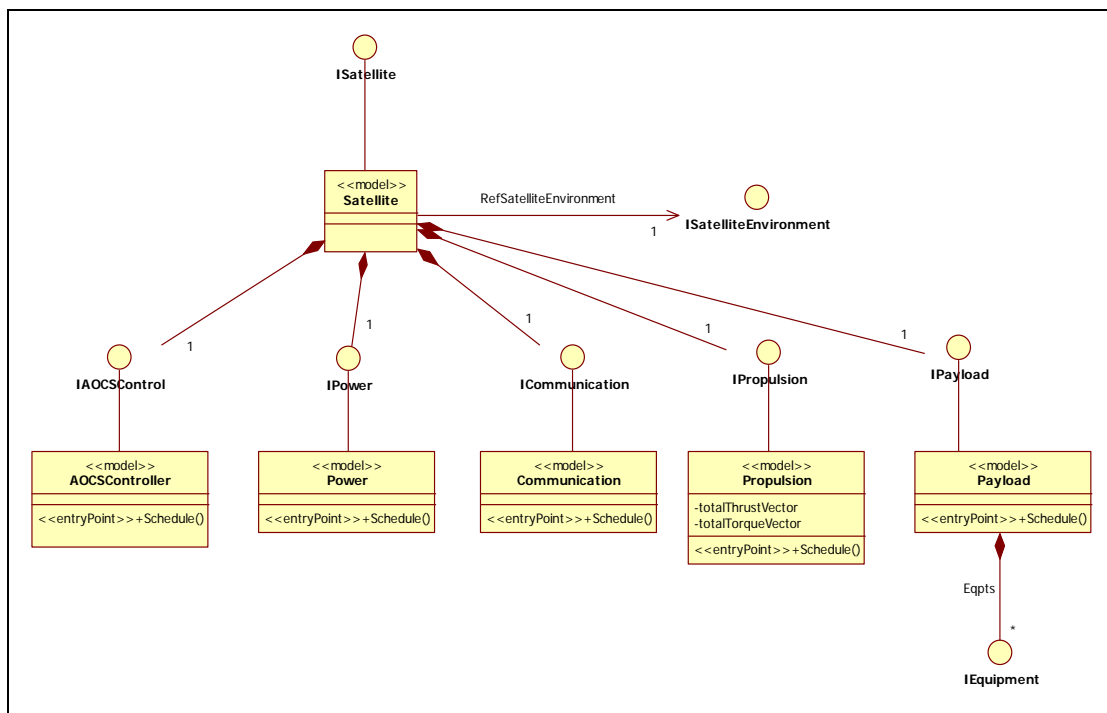


Figure 7-4: Example Simulator Satellite Model Overview

The following SMP Catalogues are contain all the elements that describe the example simulator:

- aocs_Sensors_Actuators: contains GYR, CSS models as well as the Propulsion sub-system (implementations provided from a “legacy” non-SMP environment).

- Com_Trans: contains the Transmitter model.
- pss_EPS: contains the PSS sub-system model.
- pss_SA: contains the SolarArray model.
- aocs_Control: contains the AOCS Controller model.
- com_Gen: contains the Communication sub-system.
- ground_Gen: contains the Ground Segment sub-system.
- common: common catalogue. This contains top-level models (e.g. Simulator, Satellite, ... Simulator being the root model of the simulation) and the Payload sub-system. Some of common models like Equipment, Sensor, etc are also specified here.
- Utilities: common catalogue. This contains utility types used by all catalogues.

These elements are then integrated using the SMP techniques explained in the tutorial sections of this user guide.

8

The simulator architect tutorial

8.1 Introduction

In this tutorial we illustrate the tools and techniques that a Simulator Architect uses to produce the SMP artefacts required for the Simulator Architectural Design (SAD or SDD) provided for the PDR in the E-TM-40-07 process.

These tutorials are based on the tasks introduced in section 5 , namely:

- Identify Major Sub-systems and Their Components
- Identify Data and Control Flows
- Decide How to Connect and Configure Components

8.2 Task: Identify major sub-systems and their components

8.2.1 Overview

When using SMP to describe the simulation architecture the Simulator Architect will produce an SMP catalogue to describe the elements of the simulator as SMP models and interfaces.

In this section we will introduce the following SMP topics:

- Catalogues (Volume 2: Metamodel section 5.1.1)
- Namespaces (Volume 2: Metamodel section 5.1.2)
- Component Based Design (Volume 2: Metamodel section 5.3.2, Volume 3: Component Model section 3.3)
- Interfaces (Volume 2: Metamodel section 5.2.3, Volume 3: Component Model section 3.3)
- Models (Volume 2: Metamodel section 5.2.4, Volume 3: Component Model section 3.4)
- Fields (Volume 2: Metamodel section 5.2.2)
- Operations (Volume 2: Metamodel section 5.2.1)
- Events (Volume 2: Metamodel section 5.3.3, Volume 3: Component Model section 3.3.3)

- Entrypoints (Volume 2: Metamodel section 5.2.2, Volume 3: Component Model section 3.3.4)

8.2.2 Capture the architecture in a catalogue

8.2.2.1 Catalogue tool

An SMP catalogue is expressed in terms of SMDL in an xml document. However, an architect would typically not create this document directly but instead would use a tool. Some examples of these tools are:

- A catalogue editor
- A UML software design tool mapped to SMP types
- An engineering design tool

An SMP catalogue is a technical document that describes a *specification* for the simulator under design. It describes the models, interfaces, operations, fields, properties and the dependencies between them.

Figure 7-1 shows an engineering view of the example simulator spacecraft architecture (simplified) as it might be delivered to the Simulator Architect.

8.2.2.2 Catalogue design

The Simulator Architect could create an interpretation of the design shown in Figure 7-1 in a number of ways. For example using a UML Model the Simulator Architect could describe the simulator high level architecture as follows:

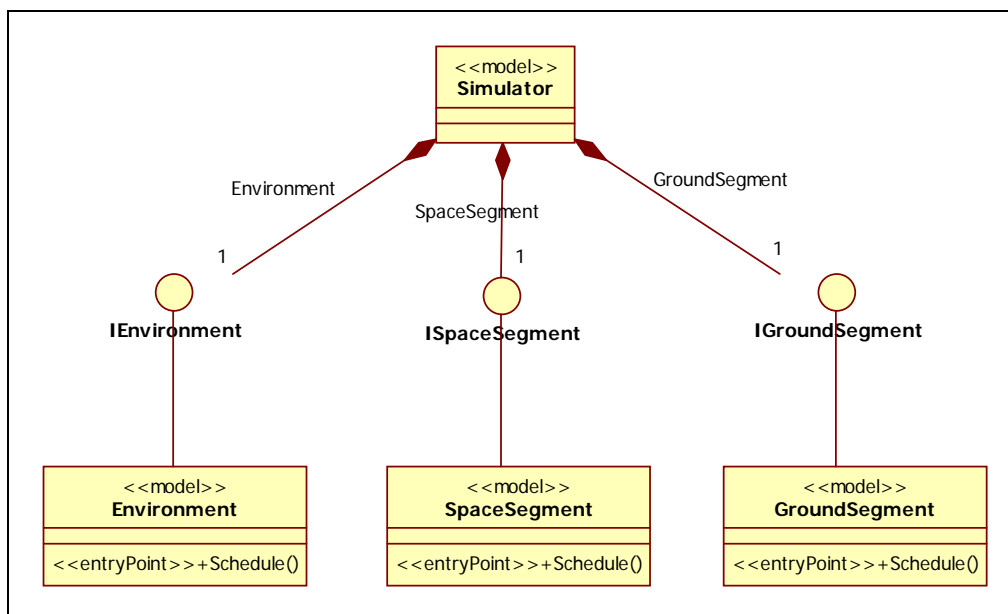


Figure 8-1: UML Model of Main Simulator Architectural Elements

A UML representation of a catalogue is often used but is by no means mandatory. Using a UML tool is a choice for the Simulator Architect. Use of UML is popular with software engineers because SMP contains many object-oriented software techniques that can be expressed accurately using UML. Such

a UML tool might also be able to automatically translate this design into an SMDL catalogue. However, UML encompasses software engineering not space engineering concepts and may not always be suitable. Alternatively an engineering software tool might be used to do the same translation from space engineering concepts into an SMP catalogue.

Often the simulator design is created using an SMP catalogue editor tool which offers a more direct view of the catalogue contents.

For example the high level simulator architecture is shown in Figure 8-2 below as a snapshot from a catalogue editor.

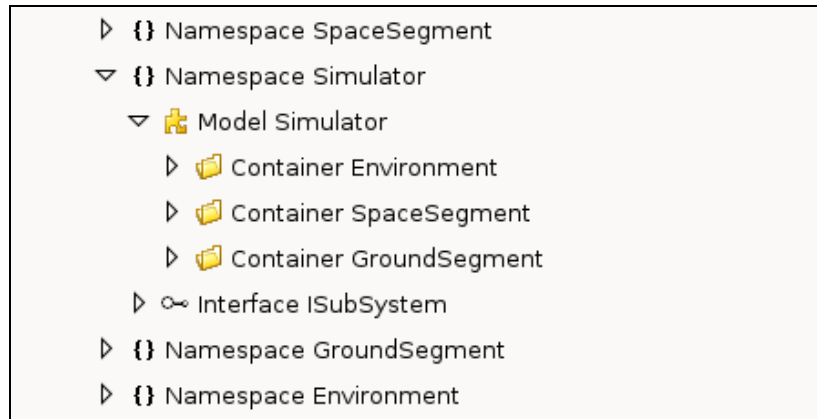


Figure 8-2: Snapshot SMP Catalogue Editor

8.2.3 Organise the catalogue through namespaces

(See also Section 5.1.2 of ECSS-E-TM-40-07 – Volume 2A: Metamodel)

Namespaces are a place to put the models in your simulator, a way to organise the catalogue information. They have special technical significance in the implementation code as they are used as mechanism for the software elements to refer to one another (the namespace forms part of the name used when referring to the component by name). Namespaces maybe nested to represent a hierarchical organisation of simulator components. Figure 8-3 shows the basic relationships between Namespaces and other elements as a UML model.

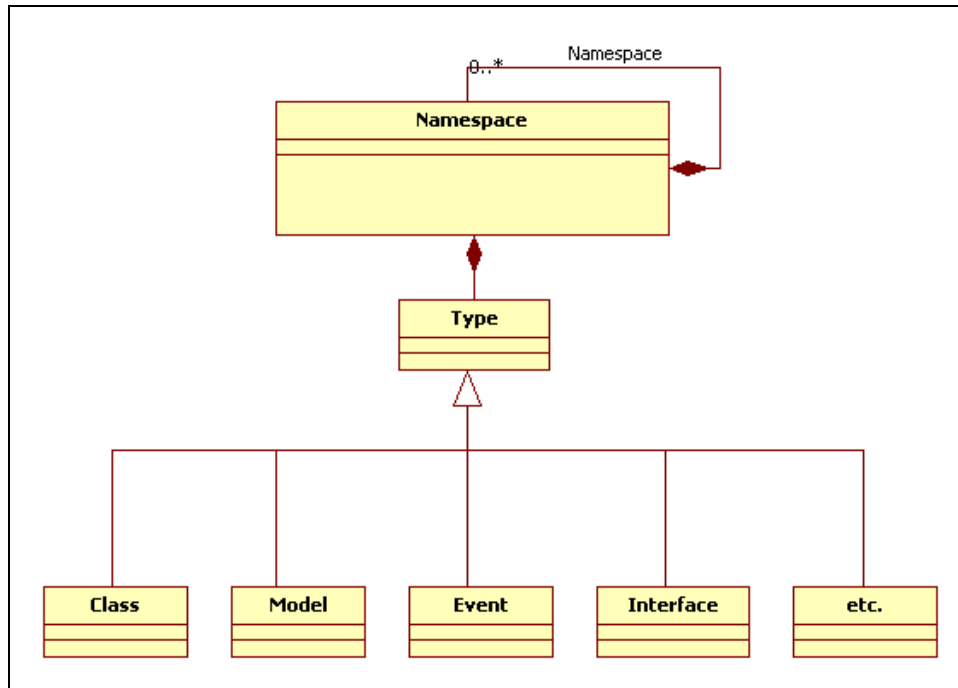


Figure 8-3 Namespaces in SMP

Figure 8-2 shows an explorer style view of an SMP catalogue. In this snapshot we can see that the SMP simulator has been broken down in to a number of *namespaces* that represent the main architectural breakdown of the simulator. Namespaces are a mechanism used to organise models within an SMP catalogue. The example shows that the main elements of the architectural design; “Simulator”, “SpaceSegment”, “Environment” and “GroundSegment” have been organised into namespaces.

8.2.4 Structure the simulator with component-based design

The UML representation in Figure 8-1 introduces the first of our SMP techniques, *component-based design*.

SMP component-based design is a fundamental technique for expressing the relationships between the model elements in a simulator. There are two ways to express these relationships, either as aggregation or as composition. These are concepts from object-oriented software design which can be simplified to “has a” (aggregation) and “is made of” (composition). In SMP this translates to a model having a “reference” (aggregation) to another model which it needs to use or being a “container” (composition) for expressing that another model is a part of the model being described in a hierarchical relationship.

The example given in Figure 8-1 shows that at the highest level in our simulator architecture there is a “Simulator” model which contains (represented by the UML composition symbol of a filled diamond) the other three simulator models (“SpaceSegment”, “GroundSegment” and “Environment”). This diagram represents a hierarchical organisation of the simulator components which will be decomposed further later. SMP containers and references are key elements to

use when modelling sub-systems and creating the main structure of a simulator.

8.2.5 Describe containment structure through interfaces

Note that the composition/containment relationship is expressed via the *interfaces* of the “SpaceSegment”, “GroundSegment” and “Environment” models. This meaning of “interface” is also a concept from object-oriented software. Interfaces used in this way separate the containing model from the actual implementing model being contained. The container only depends on this interface and not the actual implementation. This concept is discussed in more detail in a later tutorial when we consider how to introduce a different model implementation into a simulator.

8.2.6 Define the sub-system models

(See also Section 5.2.4 of ECSS-E-TM-40-07 – Volume 2A: Metamodel)

When we have decided on the main structural elements of a simulator (i.e. the sub-systems) the next important step is to define the elements that those sub-systems contain that will provide the behaviour for the simulator.

These behavioural elements are captured in SMP *models*.

In defining model in the simulator architecture SMP catalogue they will primarily include the definition of the following elements:

- Fields
- Operations
- Entrypoints
- Internal Data

Figure 8-4 shows these basic SMP model features and the relationship with the Container and Interface features mentioned in previous sections. The additional features are described below.

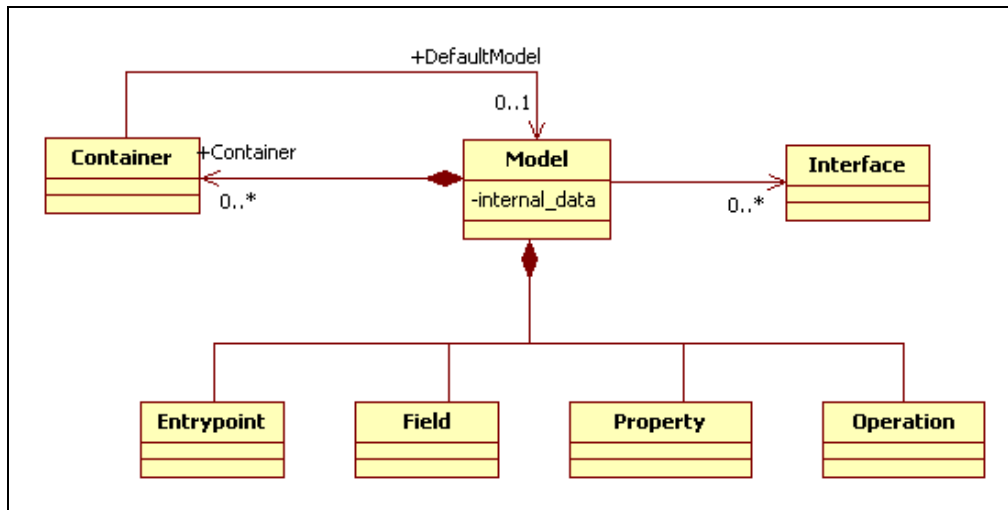


Figure 8-4 Basic Schema of an SMP Model

- **Fields**

Fields are part of the internal state of a model and have a value type which may in addition have a default value associated with their definition in a catalogue.

The architecture may also declare that a field is an *input* or an *output* field of a model. Inputs and outputs are a special case for fields and are discussed shortly with respect to dataflow-based design.
- **Operations**

Operations define the methods that will eventually be implemented for a model. Operations are called to perform the computations or behaviour of a model. They may be passed parameters and return a value. Operations therefore operate on the fields of a model and the parameters they are passed.
- **Entrypoints**

Entrypoints are a special type of model operation that have no return value and no parameters and that are always declared public so that they are accessible to other elements in the simulator. Entrypoints are of particular importance when considering scheduling.
- **Internal Data**

Internal data are data variables used by a model that are not known by the SMP environment and not visible to other models or to the SMP infrastructure that executes the models. As the name suggests they are variable used internally by model. For example they are often used for intermediate steps in a computation. Since internal data are not known by an SMP infrastructure they cannot be breakpointed for use in an SMP runtime tool for observation and cannot be part of the state of a model.

Further details and examples of the implementation of these features are given in Section 9.3.3.

8.3 Task: Identify data and control flows and their modelling

8.3.1 Overview

Once the architect has defined the structural elements of the simulator: sub-systems, containers, interfaces and models the next task is to decide how these elements work together to provide and use each others data and functions.

Now the architect has to decide which mechanisms will be used by the simulator components in order to interoperate with one another.

This section introduces:

- Interface-based design
- Dataflow-based design
- Event-based design
- Properties
- Inheritance
- Field links
- Interface links
- Event links

8.3.2 Interface-based design

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 can provide any number of interfaces.

An Interface defines a contract between models. Every model implementing the interface has to provide all the functionality of the interface, so that every model, which consumes this interface can rely on a complete 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 8.3.3) that read or write a single value, access to fields can be added.

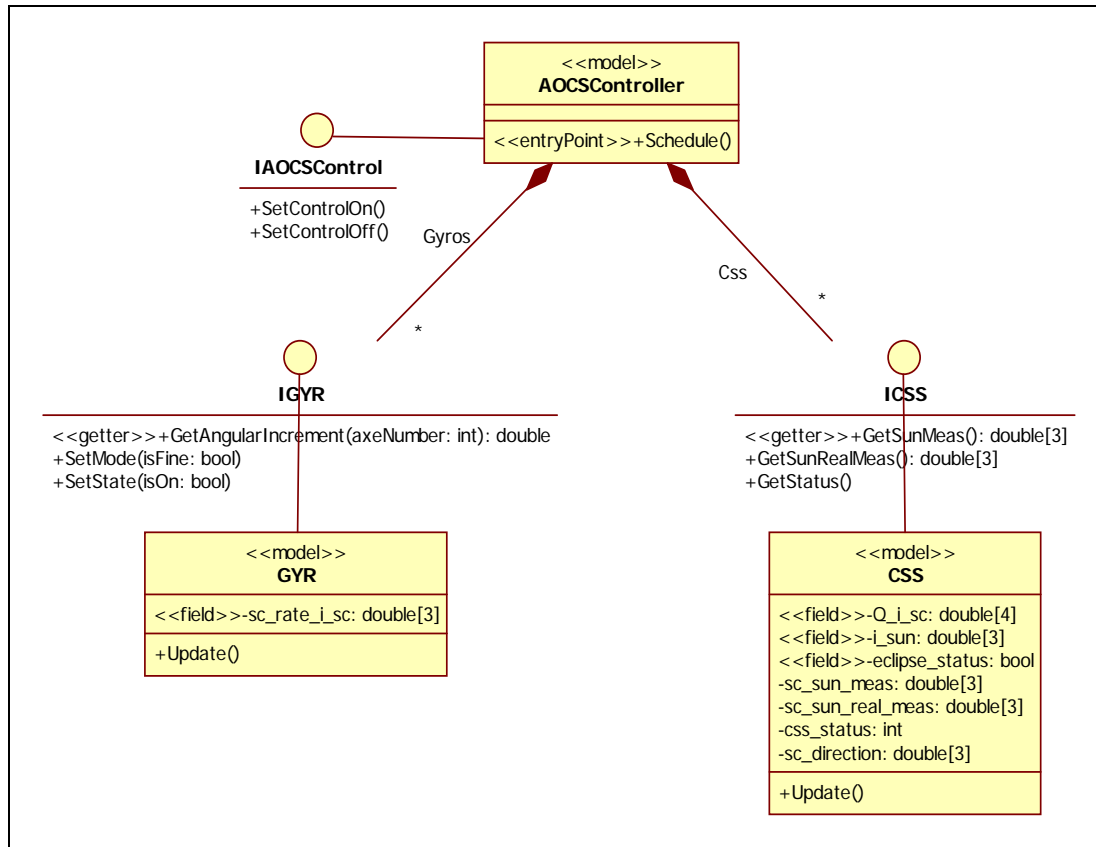


Figure 8-5: AOCSS Sub-system Interfaces

In Figure 8-5 above we can see several elements of an interface-based design for the AOCSS sub-system. These different elements illustrate some key facets of interfaces in SMP.

Offered Interfaces

The AOCSS sub-system itself offers the *IAOCSSControl* interface shown on the UML diagram extending beyond the scope of the AOCSS package, a convention used to indicate that the interface is available to other sub-systems to use.

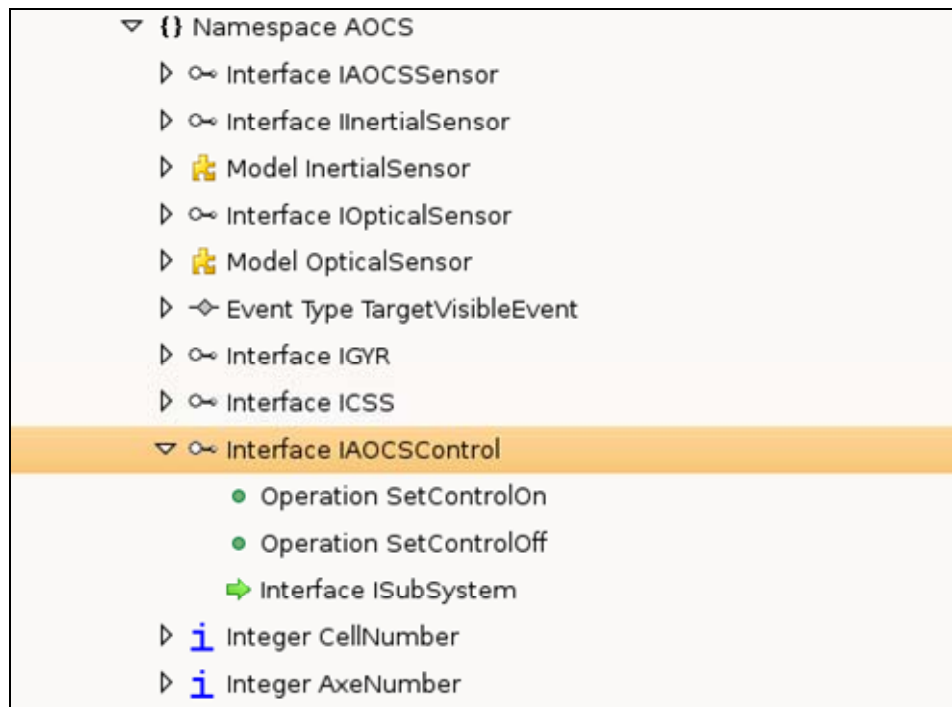


Figure 8-6: IAOCSSControl Interface Definition

The Figure 8-6 above shows the IAOCSSControl interface definition in a catalogue editor.

(Notice that all of the elements that form the AOCS sub-system have been grouped together under the AOCS namespace, including the integer types “CellNumber” and “AxeNumber”.)

Expressing Containment Via Interfaces

The *AOCSControl* model is also a container for the gyroscope (*GYR*) and sun sensor (*CSS*) models and contains them, as explained previously, through their interfaces (*IGYR* and *ICSS* respectively).

Using an External Interface Via a Reference

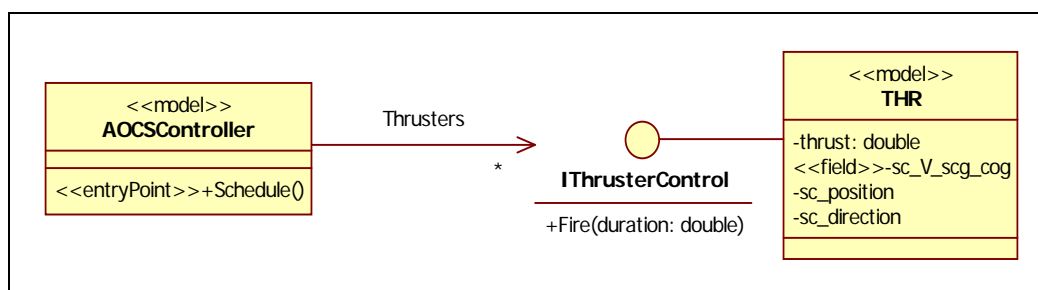


Figure 8-7: AOCS Controller References Thruster Model

The *AOCSControl* model is also a consumer of the *IThrusterControl* interface that is offered by the *THR* model (which is part of the *Propulsion* sub-system).

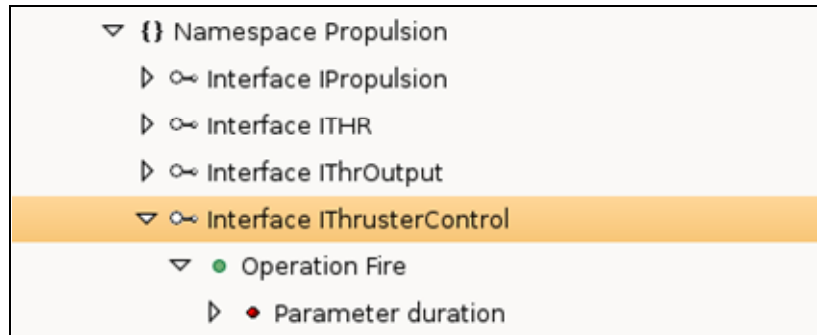


Figure 8-8: IThrusterControl Interface Definition

The Figure 8-8 above shows the IThrusterControl interface defined as a interface of the Propulsion Sub-system using a catalogue editor.

The relationship between the AOCSController Model and the IThrusterControl interface is expressed using an SMP *reference*. A reference is used in SMP when a component (e.g. sub-system or model) uses an interface of an external component (i.e. that it does not contain).

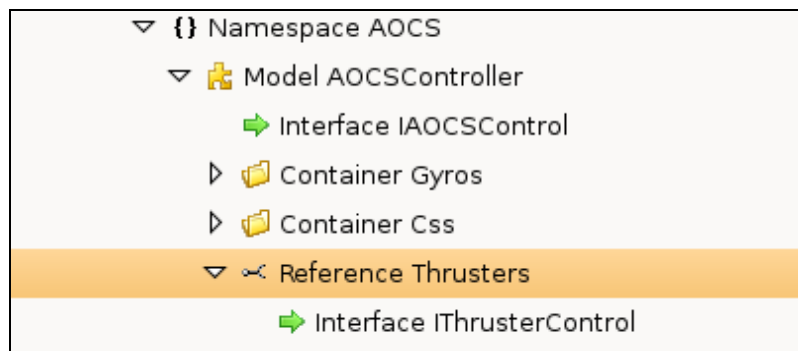


Figure 8-9: AOCSController Reference to IThrusterControl

The mechanism for defining interface dependencies is very similar for containers and references but their purpose, the way they are managed and the code that will be generated for them is different.

Interface-based Design Rationale

Now that we have considered the mechanisms of interface-based design it is worth considering the rationale for these mechanisms especially for readers that may be unfamiliar with object-oriented and component-based development (CBD) techniques.

Interfaced-based control flow has been chosen in the case of the AOCS example because we are modelling a control mechanism which is suited to interface-based design. This mechanism uses direct invocation under the control of the caller (interface consumer). In the case of the AOCS this will be the scheduler calling the AOCS update routines and the AOCS calling the update routines on the sensors and actuators.

The most important aspect of interfaces, particularly from an architectural point of view, is that they define the dependencies between elements in a simulator before we specifically consider how those elements will be implemented. For example in the AOCSController design what we have essentially defined is

shown in Figure 8-10. This is the information that can be entered in the catalogue as a first step in creating the architecture for the AOCS model.

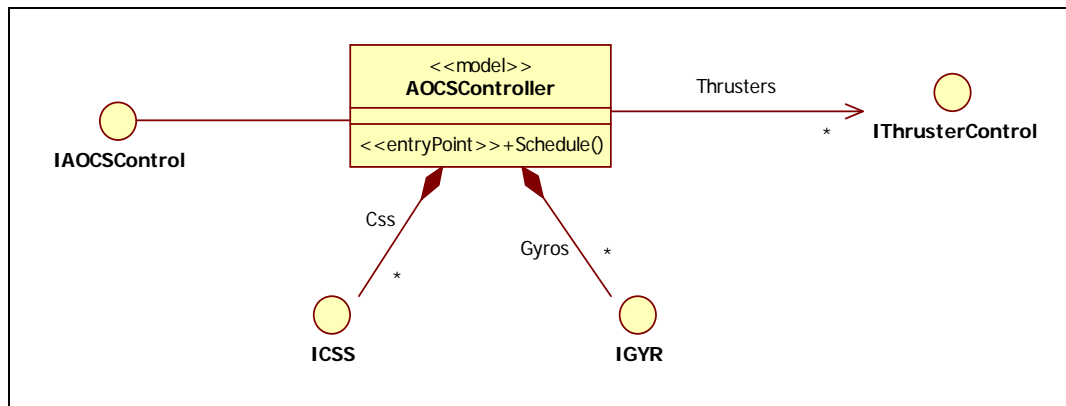


Figure 8-10: AOCS Interface Dependencies

In Figure 8-7 we showed that the *IThrusterControl* interface is implemented by a *THR* model. In fact the *AOCSController* model does not need to know this as it does not depend on this implementation. The Propulsion Sub-system could be implemented in any number of ways (e.g. as a monolithic model with no breakdown or into separate thruster models) but in order to start creating the simulator architecture we can defer the implementation decisions by focusing on the interfaces that are required first.

The other benefit of this approach is that design of the simulator components can be more flexible (e.g. one of many thrusters models) as long as the interfaces dependencies are respected. The essence of using a purely interface-based approach is to primarily focus on these interface dependencies. In the software world this is the most important aspect of CBD.

(As a separate consideration we will see later why breaking down a sub-system into separate models has a considerable benefit when configuring the simulator. If we have a separate thruster model in the Propulsion Sub-system then we can configure the number of thrusters we need. In a monolithic implementation this would not be possible through configuration and would have to be hard-coded in the Propulsion model. SMP does not restrict these choices but offers the facilities to have a more configurable and flexible design through its object-oriented/component-based approaches such as interface-based design.)

Other Features of Interfaces

An interface can inherit functionality from any number of other interfaces. This mechanism is called interface inheritance. As an interface is only a contract with no implementation, multiple inheritance of interfaces can be mapped to most languages not supporting multiple inheritance (e.g. Java or C#, which both natively support interfaces).

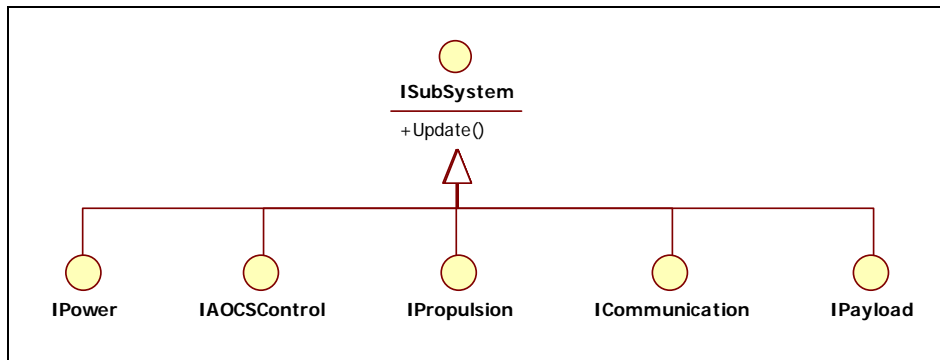


Figure 8-11: Interface Inheritance

Figure 8-11 shows how interface inheritance can be used to ensure that all the main simulator sub-systems implement an *Update* operation from the *ISubSystem* interface.

8.3.3 Properties

As access to fields via interfaces is not possible, it is common to provide a pair of operations giving read and write access to a field via an interface. To ease the specification of such a pair, the *property* element has been introduced. It typically maps to a parameter-less operation to read a value (called the getter), and a void operation with a single parameter to write the same value (called the setter). However, properties may restrict access to read-only or write-only simply by providing only one of these two methods.

Therefore properties are another aspect of interface-based design that controls or encapsulates access to the fields of a model. The getter and setter operations of a property may also apply rules or logic to how the fields are manipulated that is not apparent to the model that uses the property.

The example in Figure 8-12 below shows a read-only property in a catalogue editor. Note that the *MaskingAngle* property is linked to the field *maskingAngle*. The property *MaskingAngle* is public and accessible to external elements. However, the field *maskingAngle* is private to the model.

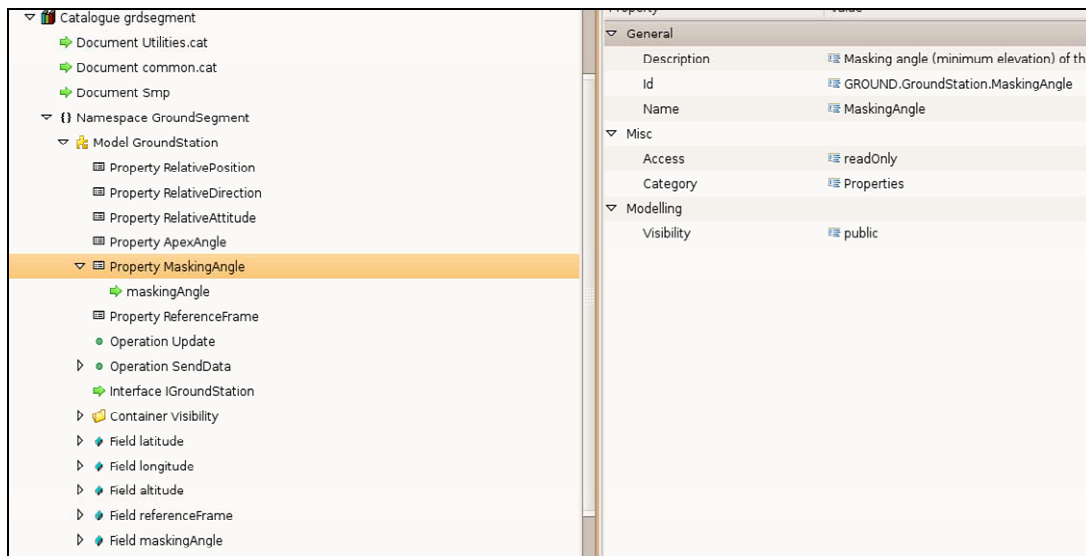


Figure 8-12: Catalogue View Of a read-Only Property

For this read-only property a code generator will only create a “getter” method, being read-only the property is not “settable”. This is shown in the following code snippet.

```

| // Get MaskingAngle.
| // Masking angle (minimum elevation) of the ground station for all directions.
| ::GeometryLibrary::Angle GroundStation::get_MaskingAngle()
| {
|     return maskingAngle;
| }

```

Figure 8-13: Read-only Property Code Snippet

8.3.4 Dataflow-based design

Whereas interface-based design is a technique particularly useful for control or direct invocation based interactions between models, dataflow-based design models a system based on the data that is shared amongst models and controls when that data is changed.

Dataflow-based design is a more traditional simulation technique that does not especially rely on object-oriented/component-based techniques. In a dataflow-based design the architect defines the data (i.e. *fields*) in the models, the relationship between those data as *inputs* and *outputs* and the *entrypoints* that trigger the exchanges of data. Once the simulator has been configured to support those relationships the SMP infrastructure controls this process based purely on the configuration (i.e. SMP assemblies and schedules), there is no programming code required to implement the dataflow in a dataflow-based design in SMP (of course programming is required for other aspects of the models).

In the Example Simulator we have used a dataflow-based design to model the collection of housekeeping telemetry data from spacecraft equipment. Figure 8-14 shows a graphical view of this design from a UML tool.

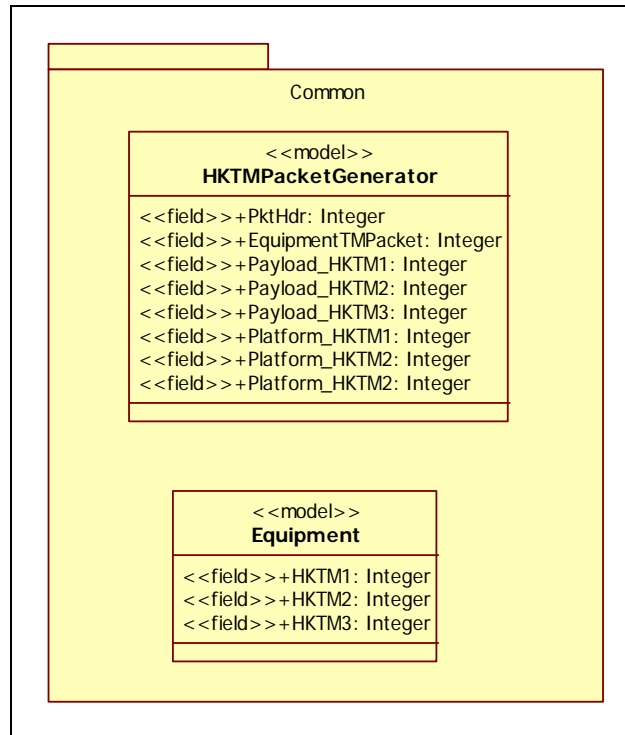


Figure 8-14: Dataflow Example UML Class Model

Figure 8-15 shows a view of the SMP Catalogue for this design from a catalogue editor. Where we can see that the housekeeping TM fields of an Equipment model have been marked as “output” fields. Similarly the fields of the HKTMPacketGenerator model (Payload_HKTM1 etc) have been marked as input fields.

<ul style="list-style-type: none"> Model HKTMPacketGenerator <ul style="list-style-type: none"> Entry Point TransmitHKTM Array EquipmentTMPacket Array PktHdr Field Payload_HKTM1 <ul style="list-style-type: none"> Default Field Payload_HKTM2 Field Payload_HKTM3 Field Platform_HKTM1 Field Platform_HKTM2 Field Platform_HKTM3 Field PktForTrans Field HKPktHdr Model Equipment <ul style="list-style-type: none"> Field HKTM1 Field HKTM2 	<table border="1"> <thead> <tr> <th>Property</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td colspan="2">General</td> </tr> <tr> <td>Description</td> <td>HK tm item 1 for Payload Equipment.</td> </tr> <tr> <td>Id</td> <td>ID_1dc8abfa-0bca-45ca-9d28-87a08222dd07</td> </tr> <tr> <td>Name</td> <td>Payload_HKTM1</td> </tr> <tr> <td colspan="2">Misc</td> </tr> <tr> <td>Input</td> <td>true</td> </tr> <tr> <td>Output</td> <td>false</td> </tr> <tr> <td>State</td> <td>true</td> </tr> <tr> <td>View</td> <td>true</td> </tr> <tr> <td colspan="2">Modelling</td> </tr> <tr> <td>Visibility</td> <td>private</td> </tr> <tr> <td colspan="2">Value</td> </tr> <tr> <td>Short Value</td> <td>0</td> </tr> </tbody> </table>	Property	Value	General		Description	HK tm item 1 for Payload Equipment.	Id	ID_1dc8abfa-0bca-45ca-9d28-87a08222dd07	Name	Payload_HKTM1	Misc		Input	true	Output	false	State	true	View	true	Modelling		Visibility	private	Value		Short Value	0
Property	Value																												
General																													
Description	HK tm item 1 for Payload Equipment.																												
Id	ID_1dc8abfa-0bca-45ca-9d28-87a08222dd07																												
Name	Payload_HKTM1																												
Misc																													
Input	true																												
Output	false																												
State	true																												
View	true																												
Modelling																													
Visibility	private																												
Value																													
Short Value	0																												

Figure 8-15: Dataflow-based Example HK TM Data collection

When describing a dataflow-based design in a catalogue there is no connection made between these models, we only describe their input and output fields.

Note that in this design the fields of the HKTMPacketGenerator model (i.e. Payload_HKTM1, Platform_HKTM1 etc) are defined against what will

eventually be instances of equipment in the running simulator. This design effectively hardcodes the equipment configuration in the catalogue.

One of the consequences of choosing dataflow-based design is that, unlike interface-based design, the dependencies that are defined between models are based on their implementations not their interfaces so that the architect will lose some degree of the flexibility in changing the implementations as they will have to change (i.e. regenerate) the model code not just change the assembly. However, this is compensated for by the fact that the changes are all usually generated by a SMP code generation tool.

However, Dataflow-based designs are often more scalable than Interface-based or Event-based designs as they provide a much simpler way to handle asynchronous processing that occurs, for example, through use of multiprocessor machines.

8.3.5 Event-based design

The simulator architect can choose to use event-based design where models tend to react to changes in the simulator on a sporadic basis depending on what is happening when the simulation is running.

Event-based design in SMP is based on a publish and subscribe mechanism. With this mechanism a model (the publisher) offers to inform other models when a change occurs via an event. The publishing model may also pass some simple data, via *event arguments*, with the event. Other models then subscribe to being notified when these changes occur. These subscribing models then provide an event handler to process the event based either on the fact that the change has occurred and additionally on the data that has been passed.

Event Sources, EventSinks and EventTypees

In SMP the publisher models are *eventsources* and the subscriber models are *eventsinks*. The two may share event data via an *eventtype*.

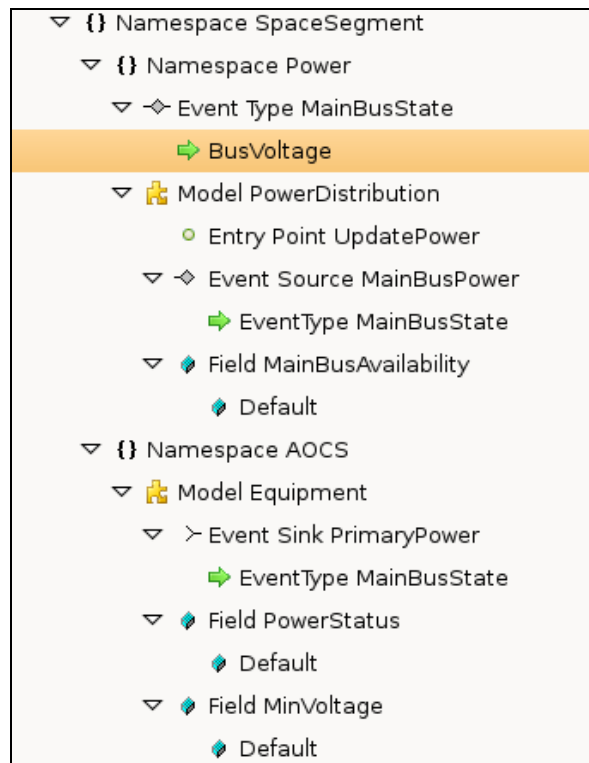


Figure 8-16: PSS Event-based Design CatalogueView

In the example simulator the *PowerDistribution* model provides a *MainBusPower* eventsource and emits this event (based on the state on the *MainBusState* field) during execution of the *UpdatePower()* entripoint. The *Equipment* model then handles the emission of the events through their *PrimaryPower* eventsink.

Note also that the that eventsource and eventsink share the same *MainBusState* eventtype which they use to pass data via an argument, in this case the *BusVoltage*. SMP eventtypes may only pass simple data types as in the case of the *BusVoltage* argument which is of type *Float32*. In the example simulator the *PowerDistribution* model will flip the mainbus voltage between a nominal value and zero via its *MainBusPower* eventsource. The *Equipment* Models that get notified of the Main Bus power changes via their *PrimaryPower* eventsinks can then change their power state based whether the supplied voltage is sufficient and switch off when the supplied voltage is zero.

8.4 Task: Decide how to connect components

8.4.1 Overview

This section will explain how to connect simulation components using an assembly.

8.4.2 Assemblies

In SMP connecting and configuring the simulator components is done by creating an *assembly*. In the catalogue we defined the model *types* that the

simulator would contain in the assembly we now define the *instances* of those types that we wish to run when the simulator is executing in an SMP compliant runtime infrastructure.

So for example the assembly will define that the simulator has only one AOCSController, two CSS and two gyros and so on.

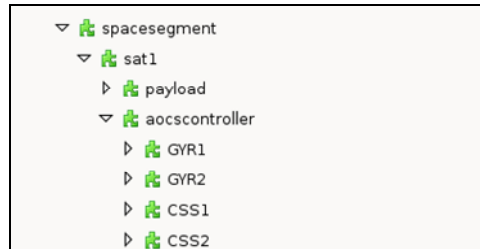


Figure 8-17: Model Instances in an Assembly

Additionally the connections between these specific instances now need to be defined in the simulator (see the discussion of the various techniques that follows in this section).

8.4.3 Interface-based design

8.4.3.1 Assembly

For an interface-based design we need create the *interface links* between the instances in the assembly. In the catalogue we defined dependency between a model *type* that requires an interface supplied by another model *type*. The AOCSController example previously given in Figure 8-10 shows this definition. When we wish to use this definition in a simulator we need to connect the *instances* of the models in our assembly which are using interface-based design, we do this using *Interface Links*.

So recap on the AOCSController example. In our catalogue we have an AOCSController model type that has a *reference* defined for an IThrusterControl interface. The catalogue also contains a definition of this interface (i.e. its operations and properties). We also have a THR model type in the catalogue that can supply this interface.

In our assembly we have an instance defined for the AOCSController called *aocscontroller* (see Figure 8-18). In the assembly we also have *PropulsionSubSystem* model instance that contains eight instances of the THR model.

Finally, in our interface-based design we connect the *aocscontroller* instance the eight *THR* instances by defining *interface-links* for each. The *interface-links* are, in effect, the instances of our IThrusterControl references.



Figure 8-18: Defining Interface-Links From AOCSController to the Thrusters

8.4.4 Dataflow-based design

For dataflow-based design we need to create the *field links* between input and output fields of the instances defined in the assembly.

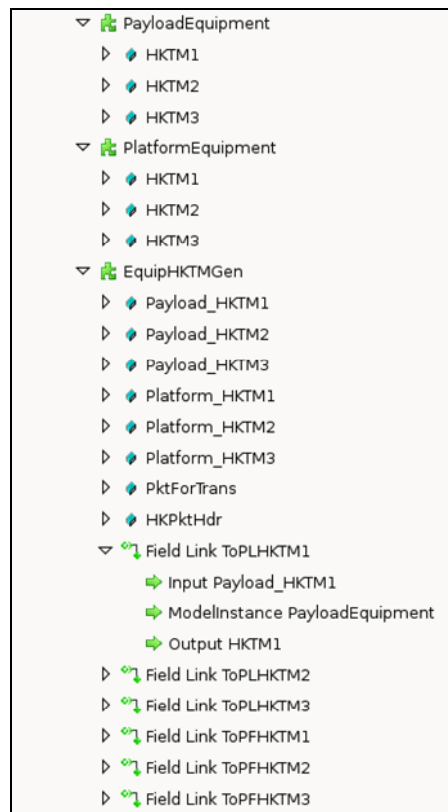


Figure 8-19: Diagram of Dataflow Field Links in Assembly Editor

In the diagram above we can see that two instances of the *Equipment* model have been created and an instance of the *HKTMPacketGenerator* model called *EquipHKTMGen*. We can also see the data fields of these models (HKTM1, Payload_HKTM1 etc). Additionally the *EquipHKTMGen* instance has a number of *FieldLinks* defined. In this dataflow example the fields of the *EquipHKTMGen* instance are declared as input fields and the fields of the two *Equipment* model instances are declared as *output fields* (i.e. the *EquipHKTMGen* instance receives the data from the equipment model instances).

For a dataflow-based design the assembly, therefore, declares the connections between the data fields of model instances using a *FieldLink* element.

8.4.5 Event-Based Design

For an event-based design the assembly will contain the *event links* that link the instances providing the *Eventsources* with the instances that contain the *Eventsinks*.

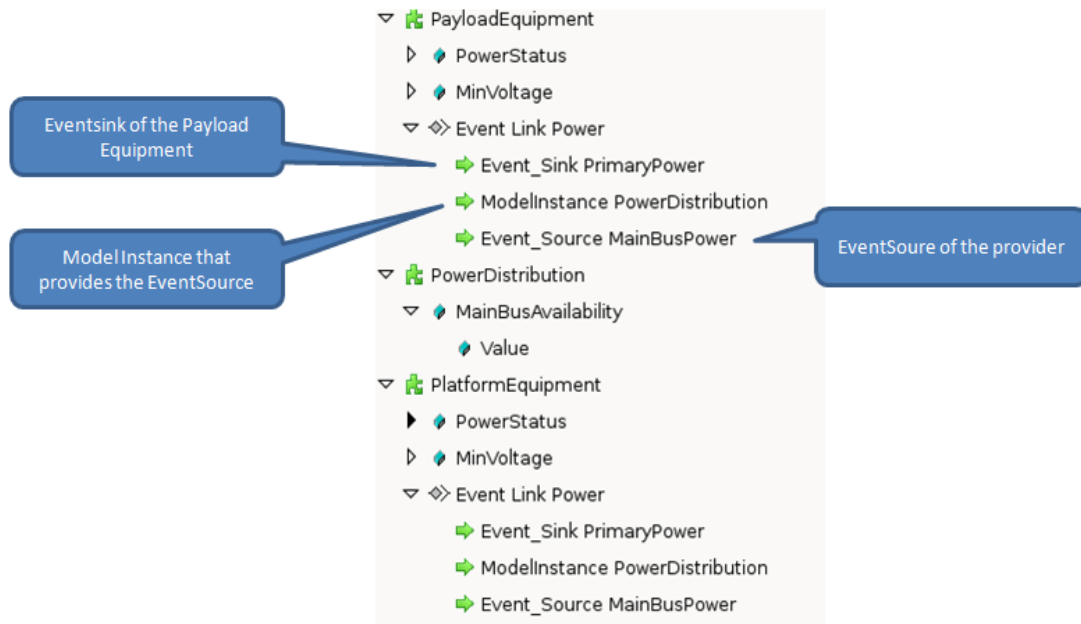


Figure 8-20: Event Links for MainBus Status

In the figure above we can how the *PrimaryPower* eventsinks for the model instances *PayloadEquipment* and *PlatformEquipment* (which are instances of *Equipment* Models) have been connected to the *MainBusPower* eventsource of the *PowerDistribution* instance.

8.5 Determine model scheduling

8.5.1 Overview

This section will explain how to schedule simulation components part of an integrated simulator using schedules.

8.5.2 Schedules

(See also Section 7 of ECSS-E-TM-40-07 – Volume 2A: Metamodel)

A schedule is the part of the simulation configuration that contains the definition what needs to be executed by the simulator as *schedule tasks* and when they need to be executed as *schedule events*. Schedule tasks may contain the following *activities*:

- Trigger - A Trigger allows execution of an EntryPoint for a specific model instance.
- Transfer - A Transfer initiates a data transfers for one or a number of Fieldlinks defined for a model instance.
- SubTask – allows the re-use of a set of activities (triggers and tranfers) defined as a task by allowing them to be executed just by referencing the task already defined.

Schedule events are based on the following characteristics:

- Simulation time event
- Epoch Time Event
- Zulu Time Event
- Mission Time Event

These events are described in Table 8-1 below :

Table 8-1 Simulation Time Kinds

Name	Description
SimulationTime	Simulation Time starts at 0 when the simulation is kicked off. It progresses while the simulation is in Executing state.
EpochTime	Epoch Time is an absolute time and typically progresses with simulation time. The offset between epoch time and simulation time may get changed during a simulation.
ZuluTime	Zulu Time is the computer clock time, also called wall clock time. It progresses whether the simulation is in Executing or Standby state, and is not necessarily related to simulation, epoch or mission time.
MissionTime	Mission Time is a relative time (duration from some start time) and progresses with epoch time.

Additionally a schedule can contain information about the definition of the timing events such as start of epoch time and mission time.

8.5.3 Scheduling – Using triggers

We can use the example of an interface-based design to illustrate the use of *triggers* in a schedule. However, the use of *triggers* is not restricted to interface-based design. Since a trigger is just a mechanism to schedule the execution of an *Entrypoint* they could equally be used in event-based (to invoke some *Eventsource* behaviour that raises an event) or dataflow-based design (to update some model data that is then provided to other models via *FieldLinks*).

Use of triggers is pertinent for interface-based design because fundamentally it involves a sequence of calls is invoked between a set of participating models, that have interface links between them. In the Figure 8-21 *TriggerAocs* will call the entrypoint called *schedule* of the *aocscontroller* model instance (the trigger provider).

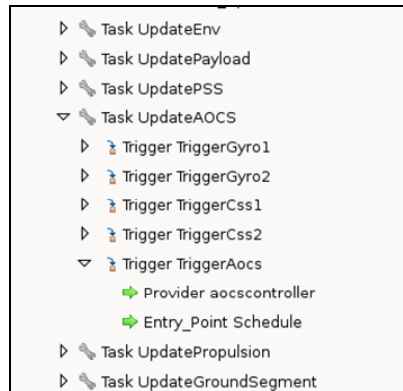


Figure 8-21: AOCSControl Trigger and Task

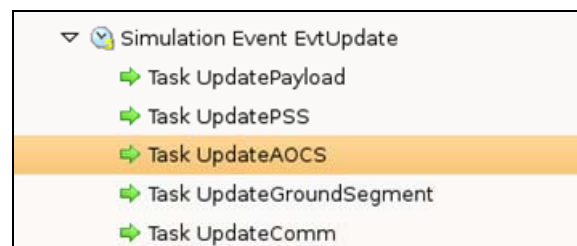


Figure 8-22: Simulation Update Event

In Figure 8-22 we can see the definition of the *Simulation Event* in an SMP Schedule. In which we have defined an update event that will call a number of tasks that will update the major sub-systems of the simulator.

8.5.4 Scheduling – Using transfers

For dataflow the schedule is required in order to define the task that will *transfer* data between the input and output fields of the instances defined in the assembly. A schedule transfer is defined for each field link that was defined in the assembly. The schedule tasks are arranged according to what data needs transferring and the schedule events determine when. Tasks, therefore, also help to organise the data transfers into groups so that they can be used in different combinations during different scheduling events.

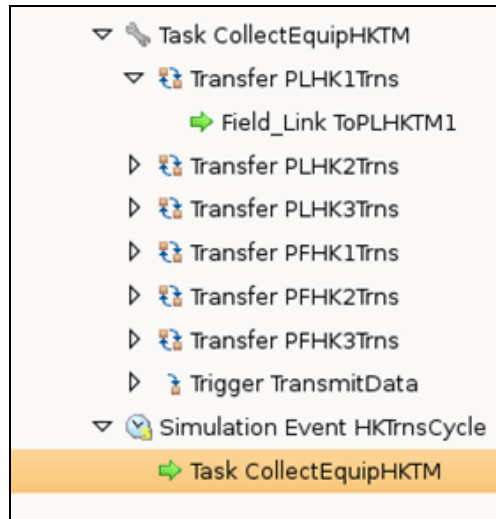


Figure 8-23: Schedule Task Containing Datalow Transfers

In Figure 8-23 above we can see a schedule *Task* called *CollectEquipHKTM* defined. This task contains a number of *Transfer* definitions. Transfers reference the *FieldLinks* defined previously in an assembly. In this case the transfer *PLHK1Trns* references the *FieldLink ToPLHKTM1* which links the fields of the *PlatformEquipment* instance to the *EquipHKTMGen* instance previously declared in the assembly. We then declare that the *CollectEquipHKTM* task is scheduled using a *Simulation Event*.

This example shows how using SMP dataflow-based design models can transfer data using a schedule purely by using the declarations provided in the catalogue, assembly and schedule with no other programming required by the model developer.

8.5.5 Scheduling - Triggering events

For event-based design the schedule may contain a trigger that calls an entrypoint that will in turn invoke the `emit()` method of an eventsource. The runtime infrastructure is then able, using the information in the assembly, to call the event-handling methods, declared in the catalogue, for each of the eventsinks that have been linked to the eventsource.

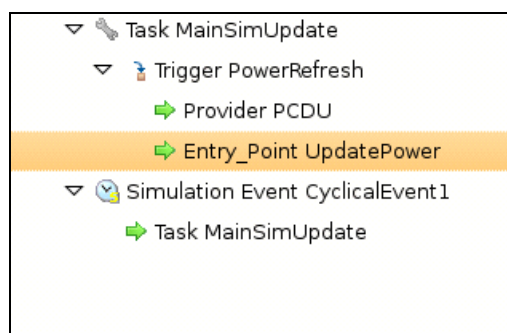


Figure 8-24: MainBus Events Entrypoint Scheduling

The model developer tutorial

9.1 Introduction

This section discusses the tasks that are usually performed by model developers in E-40-07. The primary objective of this set of tasks is to deliver a unit tested set of simulator components that are ready to be configured and integrated into a working simulator variant.

This section contains a set of guidelines for how model developers should approach the development of SMP models.

9.2 The SMP simulation environment and SMP models

A Simulation Environment has to implement the *ISimulator* interface to give access to the models and services. This interface is derived from *IComposite* to give access to at least two managed containers, namely the “Models” and “Services” containers. Finally, a simulation environment has to pass a publication server to all models in the *Publishing* state.

The Simulation Environment is always in one of the defined simulator states, with well-defined state transition methods between these states.

The available simulator states are enumerated by the *SimulatorStateKind* enumeration, while the *ISimulator* interface provides the corresponding state transition methods. Except for the *Abort()* state transition, which can be called from any other state, all other state transitions should be called only from the appropriate states, as shown in the Simulation Environment State Diagram in Figure 10-1, and explained in Section 10. However, when calling a state transition from another state, the simulation environment shall not raise an exception, but ignore the state transition. It may use the *Logger* service to log a warning message.

The simulator states correspond to dedicated Model states. Therefore a model is always in one of the defined model states too, with well-defined state transition methods between these states. The available model states are enumerated by the *ModelStateKind* enumeration, while the *IModel* interface provides the corresponding state transition methods.

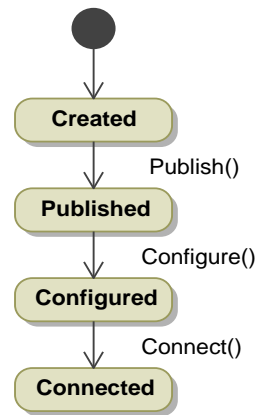


Figure 9-1: Simulation State Transitions

Section 10 discusses the state transition model for the SMP simulation environment in more detail.

9.3 Task: Implement new sub-systems and models

9.3.1 Generate code from catalogue

In the majority of cases a model developer will generate the skeleton code as a starting point for their model development using an SMP code generation tool.

Such a tool will generate code and organise it based on the contents of a catalogue and the SMP specification.

Points to mention here are (for the de facto SMP C++ mapping):

- Catalogue namespaces will be implemented as file system folders
- The code generator will generate a model header file containing the model interface code and will inherit from at least the ManagedModel interface.
- Typically the model implementation code will be organised into:
- an SMP wrapper file which implements the “boiler plate” code to use the SMP facilities required by a model (e.g. containers, references, eventsources and eventsinks, entrypoints etc) and hooks into the simulation services (e.g. logger) and lifecycle routines(publish, initialise, configure etc)
- a separate C++ file that is used for the modellers custom implementation code (i.e. the model behavioural code).

Section 10 gives more details of the simulation life cycle elements (state transition model) for an SMP simulator that are typically generated in the SMP wrapper. Developers are typically concerned with custom code for initialising, publishing and connecting there models.

9.3.2 SMP MDK

Many of the examples used in this document show SMP code that uses the SMP MDK (Model Development Kit). The MDK is not part of the SMP standard and can be thought of as other tool. The MDK shown in this document provides wrapper code to make writing SMP code even simpler for code generators as well as developers hence it is often used.

However, the MDK is only one solution for creating such boiler plate code and other mechanisms may evolve. The important point is that the model integration does rely on the MDK or any other non-standard mechanisms.

9.3.3 Implementing basic SMP model features

9.3.3.1 Introduction

(See also Features are described in Section 4.3 of ECSS-E-TM-40-07 Volume 2A: Metamodel)

Model features are the elements contained with a model and in SMP model features are defined in a catalogue using SMDL.

Usually a model developer will use a code generator tool to obtain a skeleton implementation of these features in C++ code. Of course a developer could also choose to code a model from scratch by hand, you are not forced to use a code generator. However, the facilities available in SMP to integrate a model into an SMP infrastructure require a lot “boiler plate” code and therefore it is much easier to use a tool to perform the task of creating this code consistently.

Using such tools the job of the developer is minimised to having only implement the required model behaviour. Usually a code generator will generate all the required code for the declaration and publication of features defined in a catalogue (e.g. for Fields, Properties, Operations, Entrypoints).

Some examples of often used features are given below.

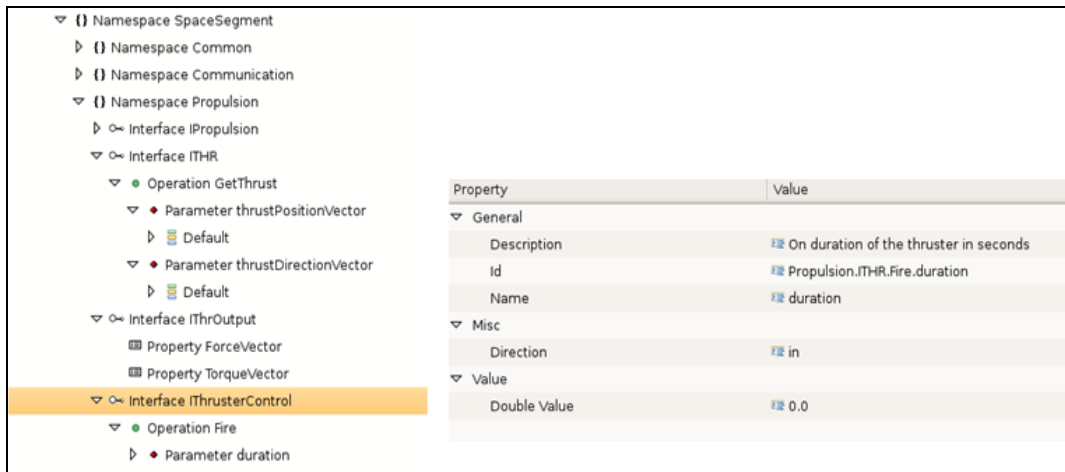
9.3.3.2 Operations

(See also Section 4.3.7 of ECSS-E-TM-40-07 Volume 2A: Metamodel)

The definition of an operation (in a catalogue) consists of the following:

- The name of the operation
- Its visibility (using the VisibilityKind enumeration which contains the following types of visibility “private”, “protected”, “package” and “public” . See Section 4.1.2 of ECSS-E-TM-40-07 Volume 2A: Metamodel)
- A list of parameters for the operation where each parameter is described by a type and a direction (from the DirectionKind enumeration containing “in”, “out”, “inout” and “return”. See Section 4.3.8 and 4.3.9 of ECSS-E-TM-40-07 Volume 2A: Metamodel for Parameters and DirectionKind respectively)
- A list of the exceptions that the operation can raise (see Section 4.2.10 of ECSS-E-TM-40-07 Volume 2A: Metamodel)

Figure 9-2 shows the catalogue definition of the *Fire()* operation in the *IThrusterControl* interface. This operation has *duration* parameter of type *Smp::Float64* (shown as *Double*), the *DirectionKind* of the duration parameter is *in* and the default value of 0.0.



Property	Value
General	
Description	On duration of the thruster in seconds
Id	Propulsion.ITHR.Fire.duration
Name	duration
Misc	
Direction	in
Value	
Double Value	0.0

Figure 9-2: Definition of Fire() Operation

Figure 9-3 shows this operation as defined in a C++ header file as void operation of the class *IThrusterControl* with the duration parameter implemented as a constant so that it is immutable in this operation (i.e. mapping to the “*in*” *DirectionKind*).

```

namespace SpaceSegment
{
    namespace Propulsion
    {
        // ----- UUID -----
        static const ::Smp::Mdk::Uuid Uuid_IThrusterControl("22f18dd4-7f1a-44d8-883b-d51772776aa9");
        class IThrusterControl
        {
            // ----- Operations -----
            public:
                // Fire a thruster
                virtual void Fire( const ::Smp::Float64 duration ) = 0;
        };
    }
}

```

Figure 9-3: Fire Operation SMP C++ Mapping Definition

Finally, Figure 9-4 shows the developer code to implement this operation. This is typically the only code the model developer needs to write.

```

// -----
// ----- Implementation of Interfaces -----
// -----
// --- Implementation of IThrusterControl interface
// Fire a thruster
void THR::Fire( const ::Smp::Float64 duration )
{
    // TODO: add your code here
    //***** START HAND CODE *****//
    // Set on_activation: ON thruster or not.
    // Compute the Torque and Force vectors.
    if (duration>0.0)
        on_activation=true ;
    else
        on_activation=false ;
    Update() ;
    //***** END HAND CODE *****//
}
    
```

Figure 9-4: Fire() Operation C++ implementation

9.3.3.3 Entrypoints

(See also Section 3.3.4 of ECSS-E-TM-40-07 Volume 3A: Component Model and 5.3.1 of ECSS-E-TM-40-07 Volume 2A: Metamodel)

Entrypoints provide the mechanism by which the simulation environment scheduler or event manager can call a model. In fact any other model or component can call a model's endpoint as they are part of its public interface.

Entrypoints are simple operations with no return value and no parameters.

A typical example of an endpoint is given below.

Note also that endpoints can be associated to input and output fields. If these links are defined the simulation environment will ensure that associated inputs will be updated before the endpoint is called and that output fields are updated after the endpoint has been called.

Figure 9-5 shows an endpoint being defined in a catalogue. As a typical usage of endpoints this endpoint is used by the scheduler to invoke an update of the THR (Thruster) model.

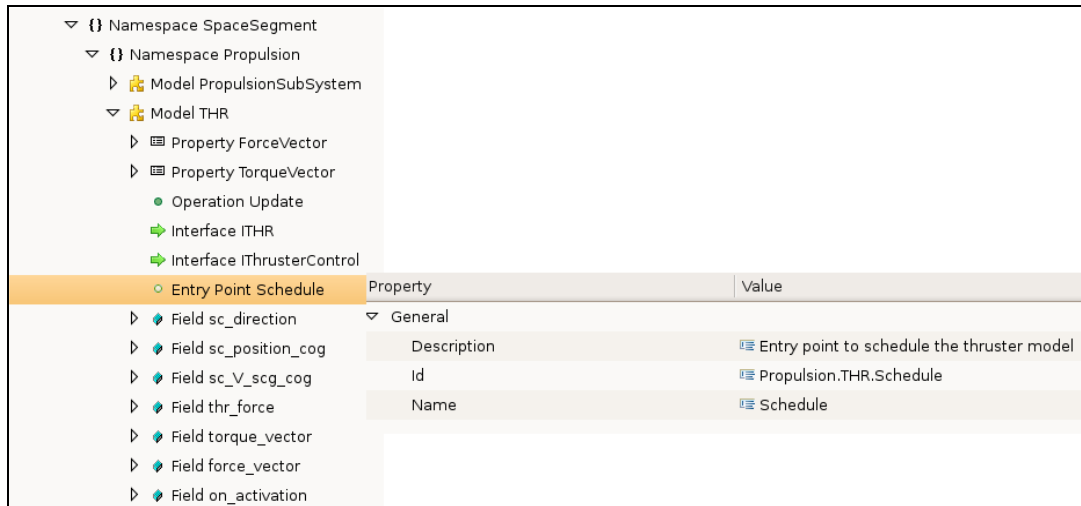


Figure 9-5: Thruster Model Schedule Entrypoint Definiton

Figure 9-6 shows a code snippet (usually generated) of the declaration and initialisation of an entrypoint.

```

// - - - - - Setup EntryPoints - - - - -
//
// EntryPoint: Schedule
Schedule
= new ::Smp::Mdk::EntryPoint( "Schedule",
                             "Entry point to schedule the thruster model",
                             this,
                             &THR::_Schedule );
// Use existing implementation to manage Entry Points
this->AddEntryPoint( Schedule );
    
```

Figure 9-6: Schedule Entrypoint Declaration

Figure 9-7 shows the entrypoint implementation code that a modellers typically provides as in this example to call the *Fire()* method of the thrusters model.

```

// ----- Entry Points -----
//
// Handler for Entry Point: Schedule
void THR::_Schedule()
{
    // TODO: add your code here
    //***** START HAND CODE *****
    Fire(1.0) ;
    //***** END HAND CODE *****
}
    
```

Figure 9-7: Schedule Entrypoint Implementation

9.3.3.4 Fields

(See also Section 4.3.2 of Volume 2: Metamodel)

Fields are used to hold model data or state. The SMP enumeration *VisibilityKind* specifies the visibility of a field to other simulator components(public, private, protected or package).

When the *State* attribute of a field is set to true the environment persistence mechanism will automatically store and restore field data when required.

Fields may also have default values and as mentioned previously when discussing entrypoints fields and data-flow-based design fields have *input* and *output* attributes.

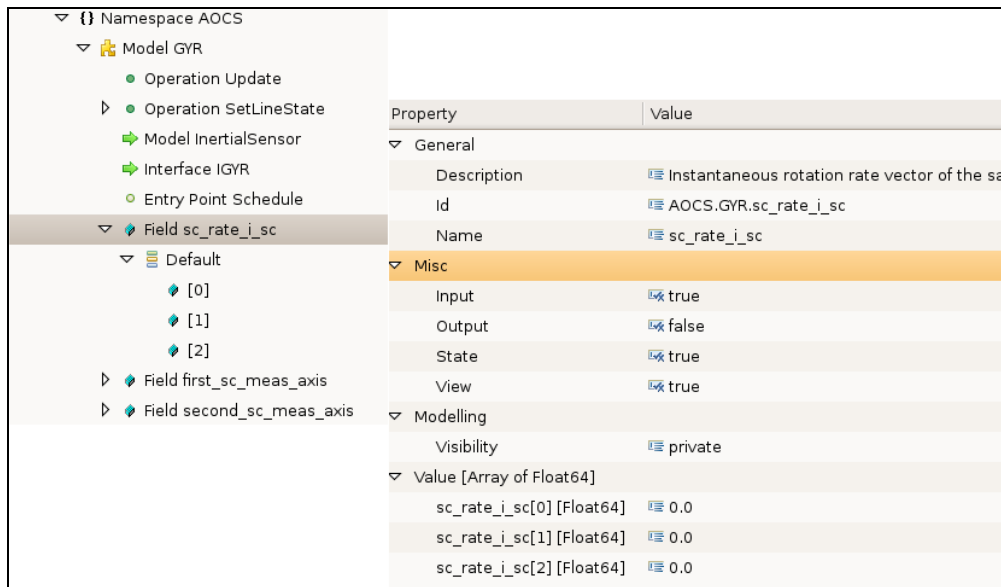


Figure 9-8: Catalogue View of a Model Field

Figure 9-9 and Figure 9-10 show the (typically generated) code required to initialise and publish fields. Note that in field publishing example that the receiver is implementing the *IPublication* interface and that this interface is platform specific in SMP (i.e. the PublishField operation is specific to the C++ mapping).

```
void GYR::_Initialise()
{
    // - - - - - Initialise private fields - - - - -

    //Array elements of sc_rate_i_sc
    sc_rate_i_sc.internalArray[0] = 0.0;
    sc_rate_i_sc.internalArray[1] = 0.0;
    sc_rate_i_sc.internalArray[2] = 0.0;
}
```

Figure 9-9: Field Initialisation Code

```
//----- Publish Fields -----
// Publish field sc_rate_i_sc
// make sure the Type is registered
::GeometryLibrary::_Register_DoubleArray3( registry );
receiver->PublishField( "sc_rate_i_sc", "Instantaneous rotation rate vector of the satellite with regards to the
inertial reference frame expressed in the satellite reference frame (rad/s)", (void*)
&sc_rate_i_sc, ::GeometryLibrary::Uuid_DoubleArray3, true, true, true, false );
```

Figure 9-10: Field Publication Code

9.3.3.5 Properties

(See also Section 4.3.3 of Volume 2: Metamodel)

A *property* provides an access mechanism for a field (via the *AttachedField* link). The access mechanism is specified by the *AccessKind* enumeration which provides the mechanisms read-write, read-only or write-only.

Unlike fields properties cannot directly represent a state of a model. The main advantage of properties over fields is that they encapsulate access to the field so that the model knows when they are being accessed and can therefore can, if required, provide a computation over the data supplied to the field.

The *AccessKind* enumeration allows SMP tools to automate the access mechanism such as only providing a “getter” method for read-only fields or a “setter” for write-only and so on.

Figure 9-11 shows the ForceVector property of the thruster model (THR). This property is attached to the force_vector field.

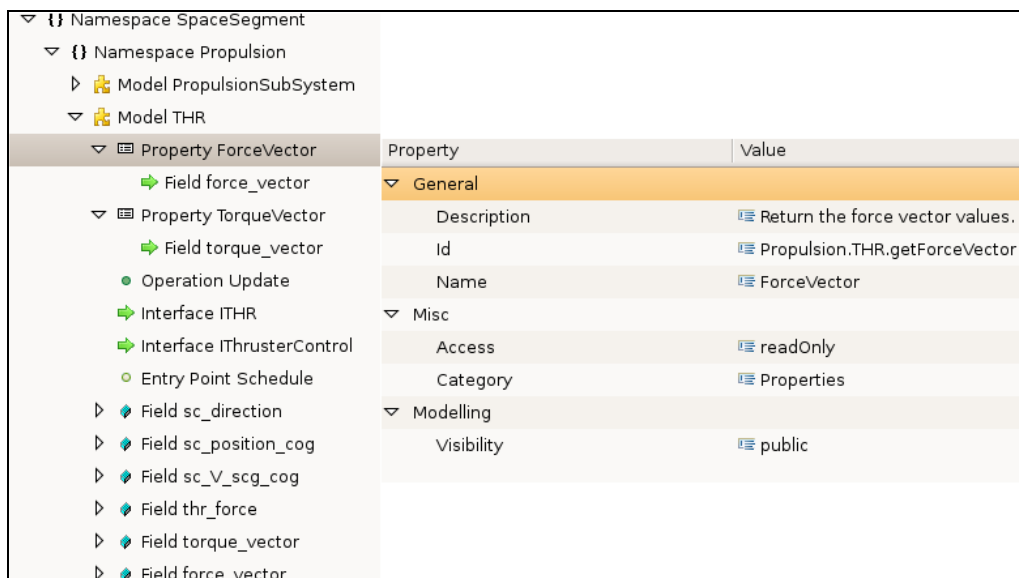


Figure 9-11: Definition of ForceVector Property

Figure 9-12 shows the code associated with implementing a property. This code would be typically generated for the developer from the catalogue details.

```

// ----- Properties -----
// ----- Properties -----
// ----- Properties -----
// Get ForceVector.
// Return the force vector values.
::GeometryLibrary::DoubleArray3 THR::get_ForceVector()
{
    return force_vector;
}

// Get TorqueVector.
// Return the torque vector values.
::GeometryLibrary::DoubleArray3 THR::get_TorqueVector()
{
    return torque_vector;
}
    
```

Figure 9-12: ForceVector Property Code

9.3.3.6 Model save and restore

An SMP environments will provide a mechanism that automatically stores and restores all model data published to the environment using the *IPublication* interface (e.g. fields with their *state* set to true).

However, SMP provides a model mechanism via the *IPersist* interface that allows model developers to code a custom store and restore mechanism.

This mechanism is useful when the standard environment mechanism is not suitable due to compatibility, performance or other issues. One example would be embedded models that need to load on-board software from a specific file. There are many other examples of when and how this mechanism could be used.

9.3.4 Implementing sub-systems

9.3.4.1 Introduction

Although the composition and aggregation mechanisms discussed in this section can apply to any SMP model they particularly relevant to models that represent sub-systems as these kinds of models tend use collections of other models.

9.3.4.2 Model Container Code

From the diagram in Figure 8-1 we saw that the Propulsion Sub-system contained the thruster models (*THR*). The Example Simulator catalogue declares this composition as a containment of *THR* models. In the diagram below we can this relationship being expressed in a catalogue editor. Not also that the Propulsion Sub-system offers two interfaces (*IPropulsion* and *IThrOutput*).

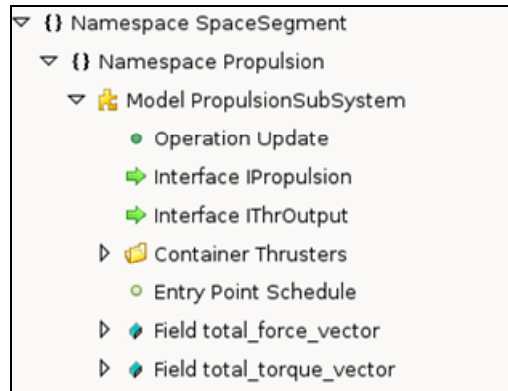


Figure 9-13: Catalogue View of Propulsion Sub-system Container For Thrusters

The implementation of containers consists of two parts that are provided by tools:

A code generator generates the SMP wrapper code which typically provides the container implementation code as shown below.

```

// - - - - - Setup Composition - - - - -
//
// Container: Thrusters
_Thrusters
= new ::Smp::Mdk::Management::ManagedContainer< ::SpaceSegment::Propulsion::THR>
  ( "Thrusters",
    "Propulsion sub system components : 8 Thrusters.",
    this,
    8,
    8 );
// Cast Mdk ManagedContainer to Smp IManagedContainer
Thrusters = _Thrusters;
// Use existing implementation to manage Containers|
this->AddContainer( Thrusters );

```

Figure 9-14: Model Container Generated Code Example

In the generated code above (see Figure 9-14) we can see that an MDK ManagedContainer is created and typed as *SpaceSegment::Propulsion::THR* model container. The container has a name (“Thrusters”), a description (“Propulsion sub system components:8 Thrusters”), the container object is the Propulsion Sub-systemmodel (“this”) and can contain exactly eight *THR* instances (lower and upper bound parameters are both eight).

The SMP simulator infrastructure will provide the instantiation/loading mechanism for the instances using the assembly.

The developer coding is kept to a minimum as the main SMP “plumbing” code is generated in the SMP wrapper file.


```

// --- Implementation of IThrOutput interface

// Get ForceVector.
// Force vector in the satellite reference frame generated by the thruster (in N)
::GeometryLibrary::DoubleArray3 PropulsionSubSystem::get_ForceVector()
{
    // TODO: return the field value (the attached field was not defined in the catalogue model)

    /******* START HAND CODE *****/
    // Sum the 8 force vectors return by each THR model.
    total_force_vector.internalArray[0] = 0.0 ;
    total_force_vector.internalArray[1] = 0.0 ;
    total_force_vector.internalArray[2] = 0.0 ;
    for (_ThrustersIterator iterTHR = _Thrusters->Begin(); iterTHR != _Thrusters->End(); iterTHR++)
    {
        total_force_vector.internalArray[0] += (*iterTHR)->get_ForceVector().internalArray[0] ;
        total_force_vector.internalArray[1] += (*iterTHR)->get_ForceVector().internalArray[1] ;
        total_force_vector.internalArray[2] += (*iterTHR)->get_ForceVector().internalArray[2] ;
    }
    return total_force_vector ;
    /******* END HAND CODE *****/
}
    
```

Figure 9-15: Model Container Developer Code

In this example we can see that the model developer provides the custom code to iterate through the *THR* model instances and compute the total force vector of all the thrusters. This is how the Propulsion Sub-system uses an SMP container to implement the *IThrOutput* interface (which provides a consolidated thruster output vector for the AOCS Sub-system).

SMP Sub-system models we frequently use this technique to aggregate the behaviour of the model instances they contain.

9.3.4.3 Model reference code

In this example we show in the AOCS Sub-system the AOCSController references the Propulsion Sub-systems Thruster models (*THR*) via the *IThrusterControl* interface to fire the thrusters. This example differs from the container example in that the *THR* model does not “belong” to the AOCSController, the *THR* model “belongs” to the Propulsion Sub-system (i.e. is in a container in the Propulsion Sub-system). In this case the mechanism for using the *THR* models is different.

The implementation consists of two parts:

In the first part an SMP code generator typically provides the implementation of the SMP reference mechanism:

```

// - - - - - Setup Aggregation - - - - -
//
// Reference: Thrusters
_Thrusters
= new ::Smp::Mdk::Management::ManagedReference< ::SpaceSegment::Propulsion::IThrusterControl>
  ( "Thrusters",
    "Reference to actuators to control the satellite attitude",
    this,
    1,
    -1 );
// Cast Mdk ManagedReference to Smp IManagedReference
Thrusters = _Thrusters;
// Use existing implementation to manage References
this->AddReference( Thrusters );
    
```

Figure 9-16: Model References Generated Code

An SMP infrastructure instantiates the models that are then referenced from the sub-system (i.e. pointers) based on the interface links provided in an assembly when the simulation is loaded. In the example in Figure 9-16 the Propulsion Sub-system model creates an MDK *ManagedReference* (called “Thrusters”), with a type of “SpaceSegment::Propulsion::IThrusterControl”, with a description (“Reference to actuators to control the satellite attitude”), the reference container object which is the *AOCSController* (“this”) and which contains a minimum of one and no identified maximum number of models which implement *IThrusterControl* interface. (the “1” and “-1” respectively).

The model developer then implements operations in the sub-system model (defined by the sub-system interface) that use the referenced models.

```

// 0020 de consigne de tuyere
int thrusterNumber = 0;
for ( _ThrustersIterator it=_Thrusters->Begin();it!=_Thrusters->End();it++){
    if ( _tuyere[thrusterNumber] ){
        (*it)->Fire(0.125);
    }
    else{
        (*it)->Fire(0.);
    }
    thrusterNumber = thrusterNumber+1;
}
    
```

Figure 9-17: Model References Developer Code

The example in Figure 9-17 shows how the *AOCSController* model (in the AOCS Sub-system) uses the references to the *THR* model instances in the Propulsion Sub-system via the *IThrControl* interface to *Fire()* all the thrusters.

9.3.5 Implement model interfaces

As we have seen previously in the example based on the *AOCSController*, its references to the *THR* models in the Propulsion Sub-system are based the *IThrusterControl* interface. In the catalogue the *THR* model has a link to the *IThrusterControl* interface meaning that this model must implement this interface. As we have seen this interface offers a *Fire()* operation.

The code generator will place the declaration of the *IThrusterControl Fire()* operation in the *THR* model header file.

```

namespace SpaceSegment
{
    namespace Propulsion
    {
        // ----- UUID -----
        static const ::Smp::Mdk::Uuid Uuid_IThrusterControl("22f18dd4-7f1a-44d8-883b-d51772776aa9");

        class IThrusterControl
        {
            // ----- Operations -----
            // -----

        public:
            // Fire a thruster
            virtual void Fire( const ::Smp::Float64 duration ) = 0;
        };
    }
}

```

Figure 9-18: THR Model Header File IThruster Control Interface

The code generator will generate the skeleton code for the operations of the models interface in the custom model code file. The developer then implements the required behavioural for the interfaces of the model.

```

// ----- Implementation of Interfaces -----
// -----

// --- Implementation of IThrusterControl interface

// Fire a thruster
void THR::Fire( const ::Smp::Float64 duration )
{
    // TODO: add your code here
    //***** START HAND CODE *****
    // Set on_activation: ON thruster or not.
    // Compute the Torque and Force vectors.
    if (duration>0.0)
        on_activation=true ;
    else
        on_activation=false ;
    Update() ;
    //***** END HAND CODE *****
}

// --- Implementation of ITHR interface

```

Figure 9-19: IThrusterControl Fire() Implementation

Figure 9-19 shows the model developer code for the Fire() operation being inserted in the place holders created by an SMP code generator.

9.3.6 Implement event emitters and handlers

The example will be based on the event handling mechanism described for the PSS and equipment models described in Section 8.3.5.

In this example we see how SMP EventSources emit events and how an SMP EventSink uses Entrypoints to handle the emitted events.

In Figure 9-20 we see an example of what the a code generator will typically provide in terms of the *eventsources* implementation.

```

// - - - - - Setup Event Sources - - - - -
//
// Event Source: MainBusPower
_MainBusPower = new ::Smp::Mdk::EventSource< ::Smp::Float32>(
    "MainBusPower", // Name
    "Indicates that power on the mainbus is available.", // Description
    this ); // Event source

// Cast Mdk EventSource to Smp EventSource
MainBusPower = _MainBusPower;

// Use existing implementation to manage Event Sources
this->AddEventSource( MainBusPower );

```

Figure 9-20: EventSource Wrapper Code for PSS Main Power Bus

The code generator is able to create the SMP *EventSource* objects for the PSS *MainBusGoesPower* event and add them to the eventsource container for the PSS.

Figure 9-21 shows what a code generator can typically provide in terms of an eventsink implementation in the SMP wrapper file.

```

// - - - - - Setup Event Sinks - - - - -
//
// Event Sink: PrimaryPower
_PrimaryPower = new ::Smp::Mdk::EventSink< ::Smp::Float32>(
    "PrimaryPower", // Name
    "Handles the change to the state of a powerline for a pieceof platform equipment.", // Description
    this, // Event consumer
    &::SpaceSegment::AOCS::Equipment::PrimaryPowerHandler ); // Event handler

// Cast Mdk EventSink to Smp EventSink
PrimaryPower = _PrimaryPower;

// Use existing implementation to manage Event Sinks
this->AddEventSink( PrimaryPower );

```

Figure 9-21: EventSink SMP Wrapper for S/C Equipment

In the initialisation of the *Equipment* model a new eventsink called *PrimaryPower* is created for the *MainBusPower* event and added to the eventsink container for the *Equipment* model.

In Figure 9-22 we can see that the customer developer code for the *UpdatePower()* entypoint for the PSS an event is emitted using the *EmitEvent_MainBusPower()*. This is convenience method generated for the developer using the MDK. This will cause the PSS to iterate through its *MainBusPower* eventsource container and call each of the eventsinks that were connected to it when the *Equipment* model instances were loaded using the assembly (refer back to 8.4.5).

```

// -----
// ----- Entry Points -----
// -----
// --OPENING ELEMENT--PowerDistribution::_UpdatePower--
// Handler for Entry Point: UpdatePower
void PowerDistribution::_UpdatePower()
{
    // MARKER: OPERATION BODY: START
    if ( MainBusAvailability )
    {
        logger->Log( this, "Main Bus Power Lost" );
        // Emit power event with 0 voltage on main bus
        EmitEvent_MainBusPower( 0 );
    }
    else
    {
        logger->Log( this, "Main Bus Power Restored" );
        // Emit power event with nominal 28 volts on main bus
        EmitEvent_MainBusPower( 28 );
    }
    // MARKER: OPERATION BODY: END
}
// --CLOSING ELEMENT--PowerDistribution::_UpdatePower--

```

Figure 9-22: Implementation of PSS MainBus EventSource

Figure 9-23 we can see the custom developer code to handle the PSS MainBusPower events via an eventsink and implemented in a handler in the Equipment model. The event handler is also passed the bus voltage as an argument and uses this value to changes its PowerStatus field.

```

// -----
// ----- Event Sinks -----
// -----
// --OPENING ELEMENT--Equipment::_PrimaryPower--
// Handler for Event Sink: PrimaryPower
void Equipment::PrimaryPowerHandler( ::Smp::IObject* sender, ::Smp::Float32 arg )
{
    // MARKER: OPERATION BODY: START
    if (arg >= MinVoltage)
    {
        PowerStatus = true;
    }
    else
    {
        PowerStatus = false;
    }
    // MARKER: OPERATION BODY: END
}
// --CLOSING ELEMENT--Equipment::_PrimaryPower--

```

Figure 9-23: Implementing an a EventSink For MainBus Events

The skeleton code for these event handlers are generated the developer just needs to add the behaviour required when the event is received (i.e. in this case log a message and set the state to on/off).

9.4 Task: Model re-use - Incorporate existing SMP models

9.4.1 Introduction

There are a number of scenarios that could be used when considering re-using existing SMP models. But most of these scenarios are variations of two main possibilities:

- Re-using an existing model as is
- Extending the functionality of an existing model in order to meet the needs of a different simulator

The existing models could also exist in two formats :

- As shared source code
- As shared binary components

The first factor in re-using any component in SMP is having a catalogue. The SMP catalogue is the specification for what is available to be re-used. Therefore it is important that catalogues are adequately documented.

9.4.2 Sharing source code

Sharing source code might be a typical example of re-use within an organisation.

By re-use in this context we do not mean taking existing source code and using it as a basis for modification.

With the source code and a catalogue in hand a developer can choose to:

- Re-use the existing implementation as is
- Inherit from existing models adapting an existing model to the specification of a new simulator project

Depending on the scenario the models may be integrated based on an existing assembly or re-configured to meet the needs of a new simulator by creating a new assembly or modifying the existing one.

9.4.3 Share binary component

Sharing an SMP binary component might be more typical of the situation where a sub-contractor or other third party is supplying simulation models as part of a contract. The third party may wish to supply the simulator components on the basis that they are not for modification either because it cannot be supported or because of IPR issues.

In this scenario the catalogue and the specification are still important. Additionally, the assembly and schedule may be deliverables because they may provide the basis on which the component has been delivered because this is how they have been tested to meet requirements.

However, it is also possible that the binary component is being delivered as a product, with its associated catalogue, specification and documentation and can be reconfigured to integrate with the customer's simulator. For example a binary component that uses dataflow can be "wired" into a simulator using a new assembly, schedule and configuration files only.

The SMP standard does not cover the installation of the binary component, this will depend on the tools and the SMP infrastructure used. However, what happens when the component is integrated, loaded and executed using an SMP environment is covered.

9.4.4 Re-using models without changing functionality

An important benefit of the SMP component-based approach to simulator and model development is the facilities it provides for incorporating existing models into simulators without adapting source code.

Traditionally this type of activity involved effort by developers to write code in order to integrate models or re-use models.

With SMP existing models can be re-used using purely integration techniques without the need for the development of specific code. However, developers may create, generate via tools or indeed re-use model stubs during their unit testing activities.

This facility is primarily provided by SMP Assemblies. The role of Assemblies in re-use and integration is discussed in more detail in Section 11 with respect to their role in simulator model integration.

9.4.5 Re-using models by extending existing functionality

(See also Section 4.3.3 of Volume 2: Metamodel)

Re-using an existing model by extending its functionality also relies on the use of assemblies in a similar fashion to that described for models that are being re-used as is. However, it also requires the creation of the catalogue elements required to describe what is being extended and the code for the extended functionality to be written. So for this scenario some model development activity is still required.

The mechanism used in SMP to describe this extended functionality is inheritance.

A brief word on inheritance. Inheritance is the object oriented mechanism to extend the functionality of a class. The C++ programming language also supports multiple inheritance where a class can inherit the functionality of many parent classes through their implementations. This type of inheritance, however, can cause conflicts in the functionality of the inheriting class. Therefore, SMP supports the mechanism more commonly found in modern object oriented languages which is single implementation inheritance and multiple interface inheritance. This means that the inheriting class has one parent class for any implementation code it extends but can inherit many other

interfaces which it then must implement. In practice this is simpler to understand and leads to clearer design.

In SMP inheritance is defined using the *base* link meta model element in a catalogue.

In order to manage this type of re-use effectively the inherited code should be based on a recognised version of the code obtained from a managed source and should not be a copy of the code. The versioning avoids unwanted changes made as the original parent evolves being incorporated into the inheriting model in an uncontrolled fashion. By making copies of the original code you risk losing the ability to manage changes effectively.

Inheritance also provides a way of re-using legacy models in SMP by inheriting from a non-SMP class.

Figure 9-24 shows the *Base* link being created in a catalogue between the AOCS GYRO model and the *InertialSensor* model. Note that the GYRO model also inherits the IGYR interface. This difference between the two types of inheritance is that for the *InertialSensor* inheritance the *GYRO* model will inheritance actual implementation code as will be shown below. For the IGYR interface inheritance the GYRO model will have to implement this itself.

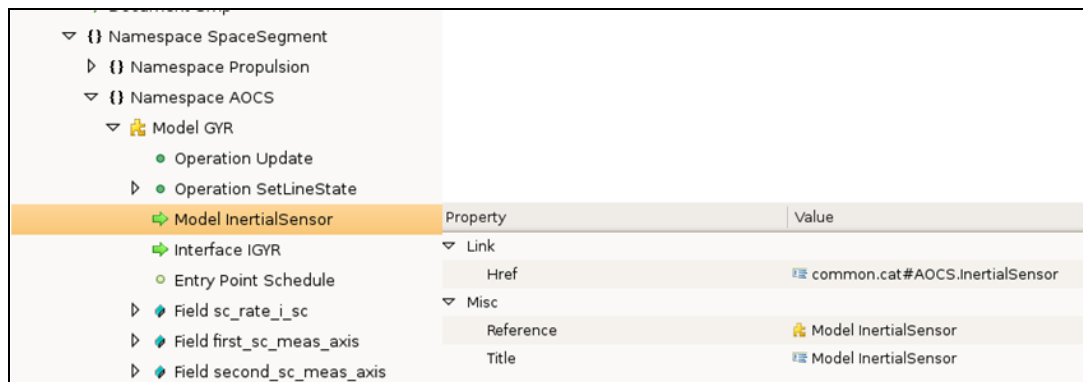


Figure 9-24: GYRO Inherits From InertialSensor

Figure 9-25 shows the functionality provided by the *InertialSensor* model that will be inherited by the GYRO model (i.e. the field pointingDirection and all the other functionality that *InertialSensor* inherits from *Equipment* model).

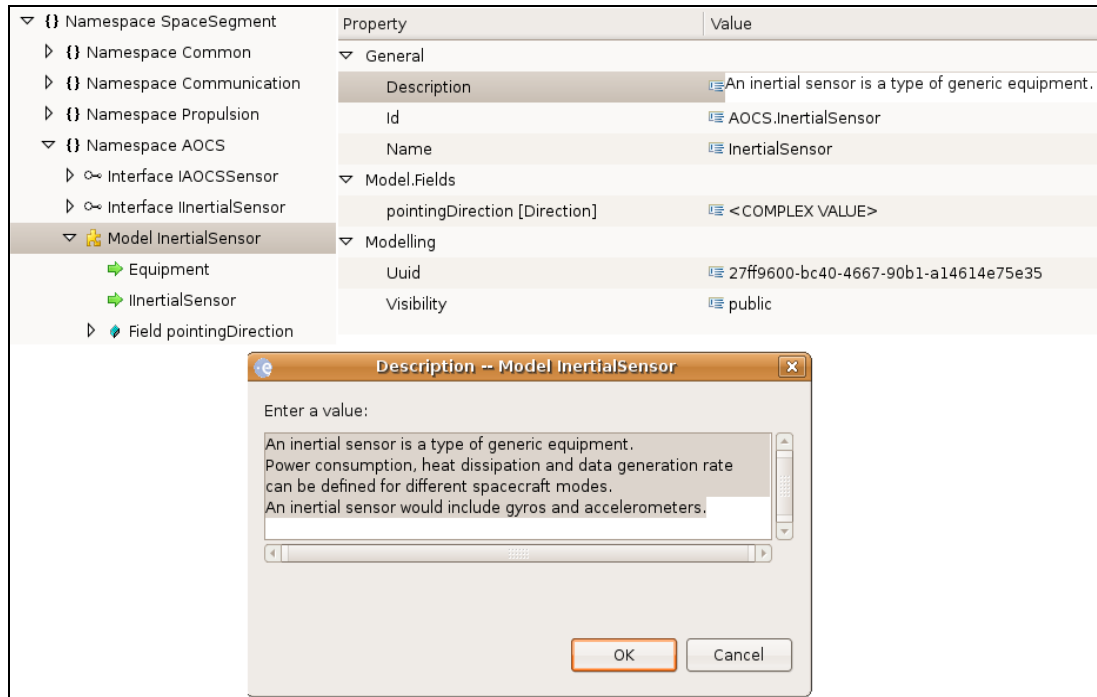


Figure 9-25: InertialSensor Functionality

Figure 9-26 shows a code snippet of code generated from the catalogue definition of the Base link between the *GYRO* model and the *InertialSensor* model.

```

namespace SpaceSegment
{
    namespace AOCS
    {
        // ----- UUID -----

        static const ::Smp::Mdk::Uuid Uuid_GYR("d122b457-5fff-4ac7-9504-e50ed71e1873");

        class GYR:
        virtual public ::Smp::IDynamicInvocation,
        virtual public ::Smp::Mdk::Management::ManagedModel,
        virtual public ::Smp::Mdk::Management::EntryPointPublisher,
        virtual public ::SpaceSegment::AOCS::InertialSensor,
        virtual public ::SpaceSegment::AOCS::IGYR
        {

            // ----- Constructors/Destructor -----
    
```

Figure 9-26: Inheritance of the InertialSensor Model Code

Figure 9-27 shows how the model developer utilises this inheritance in the *GYRO* model *Update()* method inherited from the *InitialSensor* model. From this *GYRO* model method the developer writes code to call the *Update()* method in the parent *InitialSensor* model class. This code would not normally be generated as there is a choice to be made as to whether the *GYRO* model re-uses the *InertialSensor.Update()* (as is this case) or overrides it by completely re-implementing the *Update()* method.

```

// -----
// ----- Operations -----
// -----

// Update the Gyro
void GYR::Update( void )
{
    // TODO: add your code here
    //***** START HAND CODE *****//

    // Call InertialSensor Update
    ::SpaceSegment::AOCs::InertialSensor::Update() ;

    // Check if the line and the gyr are ON
    if (lineState && powerLineIsOn && state)
    {
        switch (m_GYRNumber)
        {
            case 1 :
                rtU.sc_rate_i_sc[0] = sc_rate_i_sc.internalArray[0] ;
                rtU.sc_rate_i_sc[1] = sc_rate_i_sc.internalArray[1] ;
                rtU.sc_rate_i_sc[2] = sc_rate_i_sc.internalArray[2] ;
        }
    }
}
    
```

Figure 9-27: Implementing Inherited Functionality

9.5 Task: Model re-use incorporating non-SMP models

9.5.1 Introduction

This section discusses how to use a model from another source that is not SMP compliant.

To be clear, in this section we are talking about creating the legacy model wrapper. For re-using a legacy model that has already been wrapped the techniques will be the same as those discussed in “Incorporate Existing SMP Models”.

There are tools already available to automate this kind of SMP wrapping leaving the developer to integrate an SMP model using a catalogue, assembly and schedule only.

9.5.2 Wrapping a non-SMP model with SMP interfaces.

By “wrapping” a model we intended to provide a thin layer of relatively trivial code to incorporate a model from a non-SMP external source and that we wish to re-use. The sequence diagram in Figure 9-28 shows the essence of the wrapping concept. In this example our SMP model offers an interface with an Update() entrypoint that will utilise functionality and data from two non-SMP models.

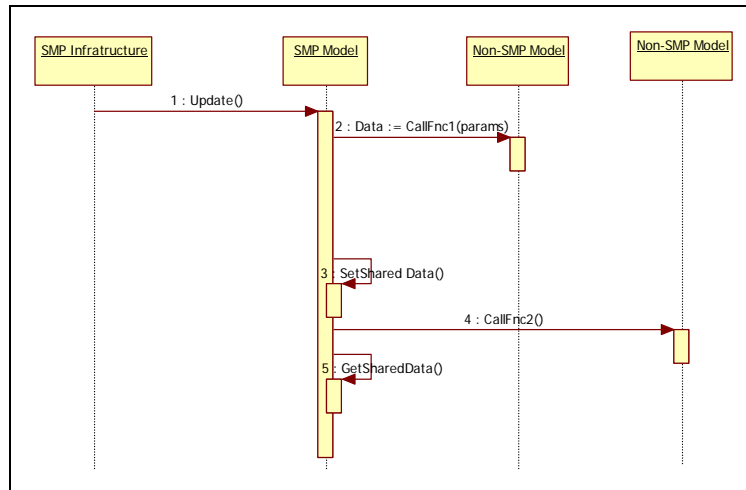


Figure 9-28: Model Wrapping Concept

In this diagram there are also two basic scenarios. Firstly, that a non-SMP model can be called by passing SMP model data as parameters and receive a return value. Secondly, that the SMP and non-SMP models exchange data through shared memory and the SMP model sets the shared data as input before calling the external function and then retrieves the data that has been updated by the external function once it has been called.

9.5.3 Simple example with stateless external model

There are several places in the Example Simulator where SMP wrapper models have been used to encapsulate the use of Matlab models. In the example shown here the THR (thruster) model uses a Matlab algorithm to compute torque and force vectors.

```

// Default constructor.
THR::THR()
    : thr_force( 10.0 ),
      on_activation( false )
{
    //***** START HAND CODE *****/
    // Initialize the algorithm and matlab data.
    initAlgorithmTHR();
    //***** END HAND CODE *****/

    // Initialise private fields
    _Initialise();
}
    
```

Figure 9-29: Initialising External Models Example

In the code snippet in Figure 9-29 the constructor in the SMP THR model is used to initialize the Matlab thruster model using the *initAlgorithmTHR()* function.

```

// Update the thruster
void THR::Update( void )
{
    // TODO: add your code here
    //***** START HAND CODE *****//
    // Compute the torque and force vectors only if the THR has been activated.
    // else return vectors set to 0.0.
    if (on_automation)
    {
        rtU.tau_on = 0.97 ;

        rtU.sc_V_scg_cog[0] = sc_V_scg_cog.internalArray[0] ;
        rtU.sc_V_scg_cog[1] = sc_V_scg_cog.internalArray[1] ;
        rtU.sc_V_scg_cog[2] = sc_V_scg_cog.internalArray[2] ;

        rtU.thr_force = thr_force ;

        rtU.sc_thr_dir[0] = sc_direction.internalArray[0] ;
        rtU.sc_thr_dir[1] = sc_direction.internalArray[1] ;
        rtU.sc_thr_dir[2] = sc_direction.internalArray[2] ;

        rtU.sc_V_scg_thr[0] = sc_position_cog.internalArray[0] ;
        rtU.sc_V_scg_thr[1] = sc_position_cog.internalArray[1] ;
        rtU.sc_V_scg_thr[2] = sc_position_cog.internalArray[2] ;

        // Execute the matlab algorithm
        executeAlgorithmStepTHR();

        // Set algorithm output data array.
        torque_vector.internalArray[0] = rtY.sc_Trqthrsc_Fthr[0] ;
        torque_vector.internalArray[1] = rtY.sc_Trqthrsc_Fthr[1] ;
        torque_vector.internalArray[2] = rtY.sc_Trqthrsc_Fthr[2] ;

        force_vector.internalArray[0] = rtY.sc_Trqthrsc_Fthr[3] ;
        force_vector.internalArray[1] = rtY.sc_Trqthrsc_Fthr[4] ;
        force_vector.internalArray[2] = rtY.sc_Trqthrsc_Fthr[5] ;
    }
}
    
```

Figure 9-30: Calling a Matlab Function from An SMP Model

In Figure 9-30 we can see an example of using a Matlab function called `executeAlgorithmTHR()` to compute the thruster torque and force vectors. The snippet also shows the shared vector data (`rtU.sc_etc`) being updated with the SMP model data (`sc_direction` etc) before the Matlab function is called. After the Matlab function the SMP model torque and force vectors are updated from the shared memory (`torque_vector` and `force_vector`).

```

// Virtual destructor that is called by inherited classes as well.
THR::~THR()
{
    //***** START HAND CODE *****//
    // Correctly end the THR matlab algorithm.
    exitAlgorithmTHR();
    //***** END HAND CODE *****//
}
    
```

Figure 9-31: Finalising External Models Example

Figure 9-31 shows that when the SMP THR model is finalised (i.e. destroyed) the model destructor method properly exits from the external Matlab execution process by calling `exitAlgorithmTHR()`.

9.5.4 Instantiation issues with “Statefull” external models

The previous example discussed a simple case where the external model has no internal state and the SMP and external models share a pool of data.

In the situation where the external model does have internal state and there are multiple instances of the SMP model wrapper this simple approach will not work.

There are SMP tools available that resolve these issues but they are not specifically addressed by the SMP standard itself.

9.6 Task: Prepare models for re-use

SMP provides many mechanisms to facilitate the re-use of models. However, it will still be important to ensure that models intended for re-use:

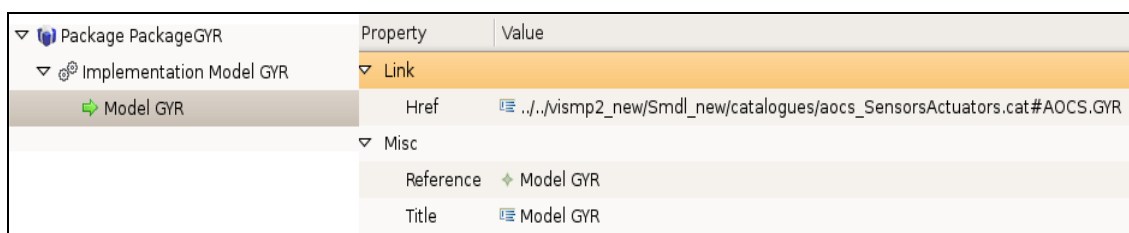
- Have a catalogue that represents their specification
- Are clearly documented
- Have all dependencies declared
- Do not have non-SMP dependencies that compromise their re-use

Importantly models intended for re-use should focus on the using the three main integration techniques described in this document, interface-based, event-based and dataflow-based design. Using these techniques allows a degree of separation between a models implementation and its interface. This is a major benefit of using SMP.

Developers should consider making this separation of concerns explicit by creating separate catalogues for their model types and their interface types.

Of the three techniques dataflow-based design offers the greatest ease of use but offers less flexibility when it comes to re-use and configuration. As we saw in the discussion on dataflow in 8.3.4 dataflow-based design tends to rely on dependencies between instances rather than model types (e.g. the HKTMPacketGenerator model had three lots of equipment TM “hardcoded” in its dataflow).

The SMP Package artefact is useful when considering the re-use of models. A Package describes all the implementation elements required for a model or a set of models (such as a simulator sub-system) that are defined in a Catalogue. In the SMP C++ Mapping (See also [NR2]) a Package maps a software library (i.e. either a DLL or an DSO depending on whether you are using a Windows or Unix OS). An SMP tool could use the information in a Package to create a build system for the packaged elements thus automating this process for the developer.



Property	Value
Link	
Href	../vismp2_new/Smdl_new/catalogues/aocs_SensorsActuators.cat#AOCS.GYR
Misc	
Reference	Model GYR
Title	Model GYR

Figure 9-32: Packaging the GYRO Model

From a Package file like this an SMP tool could generate the required code for the build system (i.e. makefile) to make the model object library and other code required by the SMP C++ Mapping to register models with an SMP simulator infrastructure so that instances of models can be created dynamically (this is required for an SMP infrastructure to be able to support creating instances on the fly using an Assembly – the mechanism is specified in the C++ Mapping).

As the name suggests an SMP Package can be used to package up models neatly so that they can be delivered, built and eventually installed.

10

The simulation environment

10.1 Introduction

The Simulation Environment always has to be in a well-defined state. These states cover the different operational phases of a simulator.

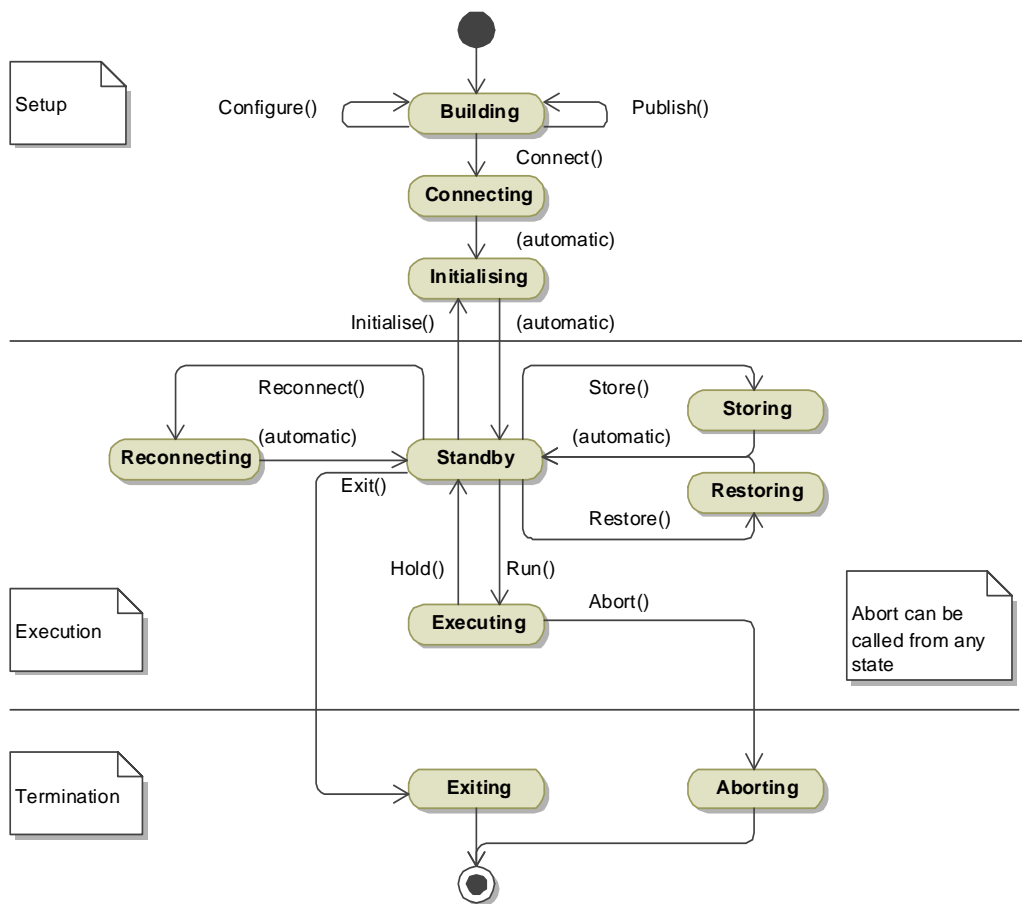


Figure 10-1: Simulation Environment State Diagram

While SMP standardises on a minimal number of states, a simulation environment may well support other states. A typical additional state would be *Debugging*. However, SMP does neither mandate additional states nor specific state transitions between them.

The simulation environment is required to emit an event via the event manager service whenever it leaves the current state or when it enters a new state. For model instances, the following six states are of main interest:

- *Standby*, to allow models to implement different behaviour in standby and executing modes. In *Standby* state, simulation time is not progressing.
- *Executing*, to allow models to implement different behaviour in standby and executing modes. In *Executing* state, simulation time is progressing.
- *Storing*, to allow models to update their state before it gets stored.
- *Restoring*, to allow models to update their state after it got restored.
- *Exiting*, to allow models clean up properly, e.g. releasing resources such as file handles.
- *Aborting*, which indicates an abnormal termination of a simulation, where only critical clean up (e.g. releasing external resources) shall be performed by the models.

For the other states, either no model instances do exist (*Creating* state), or the simulation environment calls them anyway (*Publishing*, *Connecting*, and *Initialising* state).

All mandatory states are explained below. State transitions are either initiated by calling methods of the `ISimulator` interface (see Section 3.6.1.2 of Volume 3: Component Model) or automatic after all tasks within a state have been completed. Standard events are associated with each state transition, which the simulation environment has to emit via the Event Manager service on leaving or entering each state. The event names thereby follow the scheme `Leave<State>` and `Enter<State>`. For details see the SMP Component Model (Volume 3: Component Model, Section 4.1.4.2).

10.2 The BUILDING state

This state is entered automatically after the simulation environment has performed its private initialisation (e.g. creation of simulation services).

In *Building* state, the model hierarchy is created and configured. This task includes

- creating all model instances,
- building the hierarchy of model instances,
- asking the model instances to publish their field, operations and properties,
- possibly setting initial values of fields (may also be done in the *Configuring* state),
- possibly connecting model instances by links (may also be done in the *Configuring* state).

This process can be done in one of the two following ways:

An external component creates a static scenario of model instances, sets their field values, and creates their relationships.

The simulation environment loads an SMDL assembly document, creates all model instances, sets their initial field values, and connects those using links.

While in `Building` state, the `Publish()` method can be called any time. This asks the simulator to walk through the hierarchy of model instance and call the `Publish()` method of all newly created model instances. This allows setting their field values, e.g. from information in an SMDL assembly document.

While in `Building` state, the `Configure()` method can be called any time. This asks the simulator to walk through the hierarchy of model instance and call the `Configure()` method of all model instances that are still in `Publishing` state. This should be done when these model instances have been fully configured.

In any case, the end of this phase has to be indicated by calling the `Connect()` method of the simulator (in other words: by initiating the `Connect()` state transition).

10.3 The **CONNECTING** state

This state is entered using the `Connect()` state transition.

The simulation environment traverses the complete model hierarchy and calls the `Connect()` method of each model, passing the `ISimulator` interface that allows models to query for services they need.

The simulation environment has to ensure that all services have been created and are operational when entering the `Connecting` state. It needs to provide all mandatory simulation services, and all optional simulation services that it chooses to provide.

All services must be held in the "`Services`" container of the simulator, which provides an alternative way to access services. This allows, for example, iterating on all existing services in order to query services by type/interface – instead of querying by name via the `GetService()` method.

At the end of this phase, the simulation environment automatically enters the `Initializing` state.

10.4 The **INITIALISING** state

This state is entered automatically after the `Connecting` state, or using the `Initialise()` state transition from the `Standby` state.

The latter may be used to implement a custom *reset mechanism* in a simulation environment, which may first restore a state vector that has been initially stored after the `Connecting` state, and then allow models to perform their custom initialisation again, based on the values from the restored state vector.

In `Initialising` state, the simulation environment calls dedicated initialisation entry points (in the order they have been added to it using the `AddInitEntryPoint()` method), allowing them to perform any custom

initialisation. Thereby it is guaranteed that all models have their initial values and are properly linked together.

At the end of this state, the simulation environment automatically enters the Standby state.

10.5 The STANDBY state

This state is entered automatically after the Initialising state, or using the Hold() state transition from the Executing state. It is also automatically entered from the Storing and Restoring states.

When in this state, the simulation environment (namely the Time Keeper Service) does not progress simulation time. Therefore, entry points registered with the Scheduler Service using simulation, mission, or epoch time are not executed. However, as Zulu time (which is typically connected to the system time of the computer) still progresses, events specified relative to this time will get executed.

In any case, the end of this state has to be indicated by calling one of the Run(), Store(), Restore(), Initialise(), or Exit() methods of the simulator.

10.6 The EXECUTING state

This state is entered from the Standby state using the Run() state transition.

In this state, the simulation environment does progress simulation time. Entry points registered with any of the four available time kinds may get executed.

In any case, the end of this state has to be indicated by calling the Hold() method of the simulator.

10.7 The STORING state

This state is entered from the Standby state using the Store() state transition.

In this state, the simulation environment first stores the values of all fields published with the State attribute to storage (typically a file). Afterwards, the Save() method of all components implementing the optional IPersist interface is called, to allow custom storing of additional information.

While in this state, simulation time is not progressed, and models must not change their persistent state in order to ensure that consistent state information is stored.

At the end of this state, the simulation environment automatically enters the Standby state.

10.8 The RESTORING state

This state is entered from the `Standby` state using the `Restore()` state transition.

In this state, the simulation environment first restores the values of all fields published with the `State` attribute from storage. Afterwards, the `Load()` method of all components implementing the optional `IPersist` interface is called, to allow custom restoring of additional information.

While in this state, simulation time is not progressed, and models must not change their persistent state in order to ensure that consistent state information is restored.

At the end of this state, the simulation environment automatically enters the `Standby` state.

10.9 The RECONNECTING state

In `Reconnecting` state, the simulation environment makes sure that models that have been added to the simulator after leaving the `Building` state are properly published, configured and connected.

This state is entered using the `Reconnect()` state transition.

After connecting all new models, an automatic state transition to the `Standby` state is performed.

10.10 The EXITING state

This state is entered from the `Standby` state using the `Exit()` state transition.

In this state, the simulation environment is properly terminating a running simulation. All components (especially model instances) are required to perform a clean up when required, e.g. to close opened files and release the file handlers. Note that models are not explicitly called during this state. However, they may register with the standard `EnterExiting` event to get a notification.

After exiting, the simulator is in an undefined state.

10.11 The ABORTING state

This state is entered from any other state using the `Abort()` state transition.

In this state, the simulation environment performs an abnormal simulation shutdown. Only critical clean up (e.g. release of external resources) shall be performed. Note that models are not explicitly called during this state. However, they may register with the standard `EnterAborting` event to get a notification.

After aborting, the simulator is in an undefined state.

11

The simulator integrator and tester tutorial

11.1 Introduction

In order to be clear about the goals of software verification and validation, we shall restate the meanings of these two key activities (a more formal definition of these terms is provided in ECSS-E-ST-40C:

Validation	process to confirm that the requirements baseline functions and performances are correctly and completely implemented in the final product
Verification	process to confirm that adequate specifications and inputs exist for any activity, and that the outputs of the activities are correct and consistent with the specifications and input

The distinction between these two terms is important to scoping the task of simulator integration.

This section provides examples of partitioning catalogues and simulator components from the point of view of system integration and test activities.

It calls on an existing example performed on the VISMP simulator.

The section is divided into the 3 stages:-

- Verify model
- Integrate models
- Verify Simulator

11.2 Set up integration environment

Obviously there are potentially many and varied steps associated with this task. However, as far as SMP is concerned probably the most important to mention is the installation and configuration of an SMP environment, primarily an Integrator/Tester will require at least:

- An SMP runtime infrastructure in order to run the simulator
- A tool to view SMP Catalogues

- An SMP Assembly editor to either read or create the simulator integration
- AN SMP Schedule editor to either read or create the simulator integration

Additionally setting up the integration environment involves deploying the delivered SMP simulator models. These can be delivered to the integrator in either source or binary form.

SMP *Bundles* (Specifically the SMP C++ Mapping, see Volume 4: C++ Mapping) to support this activity. Bundles are compressed file such as a tar or zip file containing the deliverable items for a model or set of models. A Bundle can contain both SMP artefacts (such as Catalogues, Assemblies, Schedules, Packages and Configurations) and other files not standardised by SMP. The structure of Bundle is not standardised by SMP. Importantly a Bundle contains a manifest that describes its contents as key-value pairs.

It will be important for the integrator to ensure that any binary files received are have been built using the same platform (compiler and OS) as the SMP infrastructure and environment they are intended to run on.

(see also Volume 4: C++ Mapping section 6.6 and Volume 2: Metamodel section 8)

Table 11-1 below describes the keys that are defined in the standard and their associated rules.

Table 11-1: Bundle SMP Manifest

Key	Meaning
Bundle-Copyright	Copyright statement for the bundle.
Bundle-ContactAddress	Full address of a person or company that can be contacted.
Bundle-DocURL	URL where documentation for the bundle can be retrieved from.
Bundle-Description	Textual description of the bundle and its content.
Bundle-ManifestVersion	A bundle manifest may express the version of the OSGi manifest header syntax in the Bundle-ManifestVersion header. If specified, the bundle manifest version must be '2'.
Bundle-Name	The Bundle-Name header defines a readable name for this bundle. This should be a short, human-readable name that can contain spaces.
Bundle-SymbolicName	The Bundle-SymbolicName manifest header is a mandatory header. The bundle symbolic name and bundle version allow a bundle to be uniquely identified in the Framework. That is, a bundle with a given symbolic name and version is treated as equal to another bundle with the same (case sensitive) symbolic name and exact version. The installation of a bundle with a Bundle-SymbolicName and Bundle-Version identical to an existing bundle must fail.
Bundle-Vendor	The Bundle-Vendor header contains a human-readable description of the bundle vendor.
Bundle-Version	Bundle-Version is an optional header; the default value is 0.0.0. A version consists of major, minor and micro version components. If the minor or

	<p>micro version components are not specified, they have a default value of 0.</p> <p>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.</p>
Require-Bundle	<p>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:</p> <pre><Bundle-SymbolicName> [; Bundle-Version="<Bundle-Version>"]</pre>
Compiler-Name	Name of the compiler that has been used to compile the source code.
Compiler-Version	Version of the compiler that has been used to compile the source code.
OS-Name	Name of the Operating System.
OS-Version	Version of the Operating System.

An integrator could use a Bundle and its manifest to unpack and install the contents of the SMP simulator deliverables he receives. Alternatively an SMP tool might read the Bundle and do this for him automatically.

A Bundle might contain all the deliverables for a simulator sub-system including items such as:

- A Catalogue defining all the SMP types in the sub-system
- An Assembly for the sub-system
- A an example Schedule
- A Configuration file
- A SMP Package file for sub-system
- The source code
- Installable binaries (complying with the specification the manifest for compiler and OS versions)
- Specification and design documentation for the sub-system
- A user manual (SUM) for the sub-system describing how to install and configure the sub-system for use in a simulator

The required Bundle contents would be specified as part of the simulator project deliverables.

Where models are delivered as source code and not binary files the integrator would be required to build the models in the integration environment. As was described in Section 9.6 the SMP Package artefact can help in automating this process.

11.3 Model Verification

This activity is concerned with verifying that the model/s being delivered is of suitable quality and stability to be integrated as part of the final simulation configuration.

This is a repeat of the module tests that are performed by the developer

At this stage model stubs could be used to represent external links required of the model. The SMP catalogue typically contains enough information to generate such stubs using a generator tool.

There are no specific SMP elements concerned with addressing this use case. However, one goal of SMP is to provide the tools using the metadata provided by the standard in order to facilitate tools to speed up these kinds of activities through code generation.

11.4 Integrate simulator in stages

11.4.1 Overview

This activity covers the systematic build-up of the final simulation configuration by assembling and integrating each model delivery when it has been verified to be ready.

An important benefit of the SMP component-based approach to simulator and model development is the facilities it provides for incorporating existing models into simulators without adapting source code.

This facility is primarily provided by SMP Assemblies. The following sections re-cap on the SMP design techniques discussed in previous sections in the context of simulator integration.

11.4.2 Re-use with containers (composition)

(See also Section 8.4.3 of this document)

Containers are a component-based design technique in SMP and as such they allow the architect and developer to defer the implementation of the models that are being contained since a Container relies only on an interface and not the implementation itself.

This has two benefits. Firstly, it allows us to re-use existing models that implement the required interface and secondly it allows us to integrate models as contained elements of another model using only an Assembly.

The examples below show how in the Example Simulator Power sub-system the integration of a Solar Array Model a power generation model is achieved using an SMP model Container.

In the simulator design an *IPowerGenerator* interface is defined as shown in Figure 11-1. This is the interface that will be implemented by the *SolarArray* model.

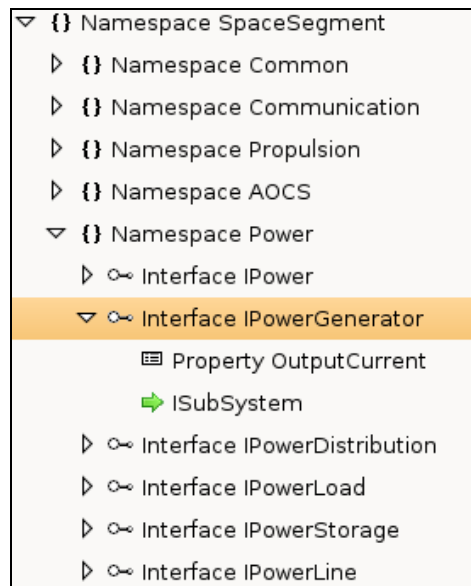


Figure 11-1: IPowerGenerator Interface

In the design the *Power* Sub-system model defines a Container called *PowerGeneration* that will contain elements implementing the *IPowerGeneration* interface as shown in Figure 11-2.

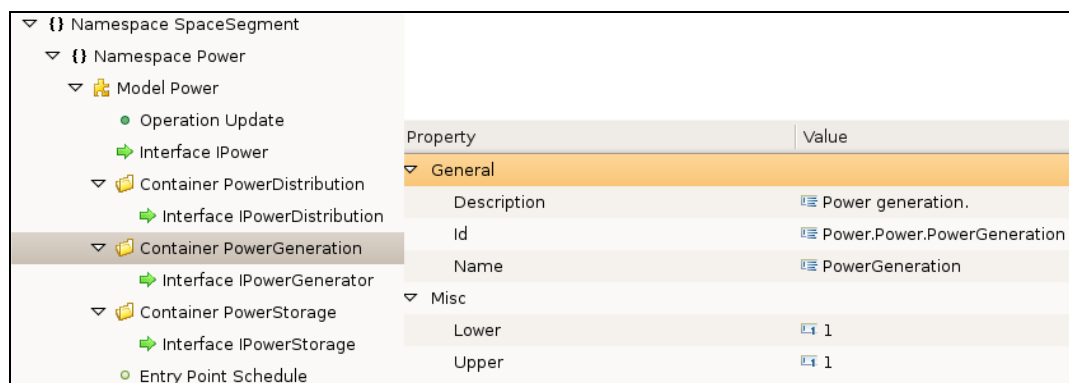


Figure 11-2: PowerGeneration Container

Based on this information from architect's design or on, for example, a System User Manual supplied with the components to be integrated, the integrator now creates the model instances for the *Power* sub-system and the *SolarArray*. In Figure 11-3 the integrator has created n instance of the *Power* Model called *Power* and an instance of the *SolarArray* Model called *PowerGenerator*. The integration of these two elements is made by the *Container* link shown for the *PowerGenerator* instance (*SolarArray* Model) which make the *SolarArray* a power generator for the *Power* Sub-system and allows the *Power* Sub-system to communicate with it using the *IPowerGenerator* interface which implemented by the *SolarArray* Model. The integrator could add more than one instance of the *SolarArray* model this way or any other power generation model that implements the *IPowerGeneration* interface.

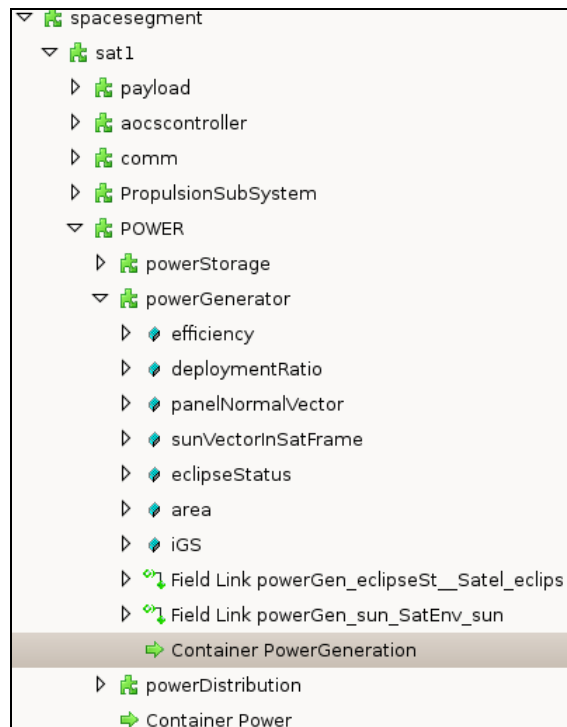


Figure 11-3 Power Sub-system Assembly

11.4.3 Re-use with references (aggregation)

(See also Section 8.3.2 of this document)

Using *References* is a key SMP component-based technique for integrating models that may be in different sub-systems (i.e. that are not contained within a sub-system).

To re-cap on this previously discussed example. The *AOCSController* models uses a *Reference* to an *IThrusterControl* interface to command the thrusters in the Propulsion Sub-system. This interface provides a *Fire()* operation with a duration parameter.

Figure 11-4 and Figure 11-5 show the definition of this relationship as part of the simulator design in a Catalogue.

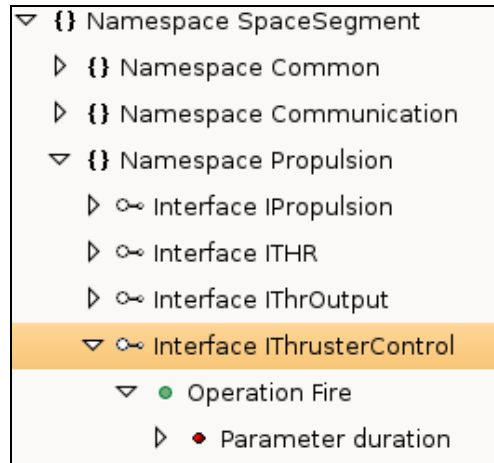


Figure 11-4: IThrusterControl Interface

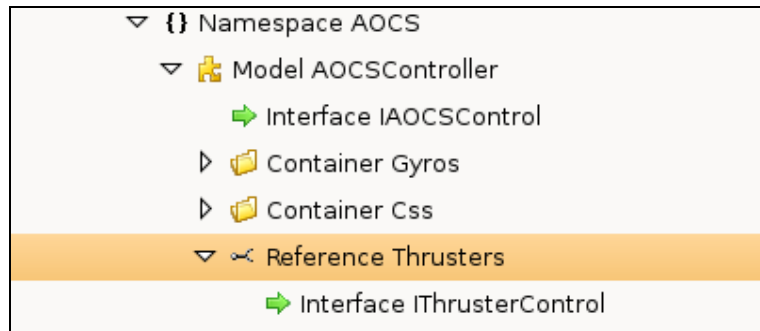


Figure 11-5: AOCSControl “Thrusters” Reference

Figure 11-6 shows the Interface links that need to be created by the integrator in order to integrate the *AOCSController* (instance *aocscontroller* in the figure) model in the *AOCS* Sub-system with the *THR* (thruster) model instances in the *Propulsion* Sub-system. The *aocscontroller* has a link for each of the eight thrusters in the *Propulsion* Sub-system.

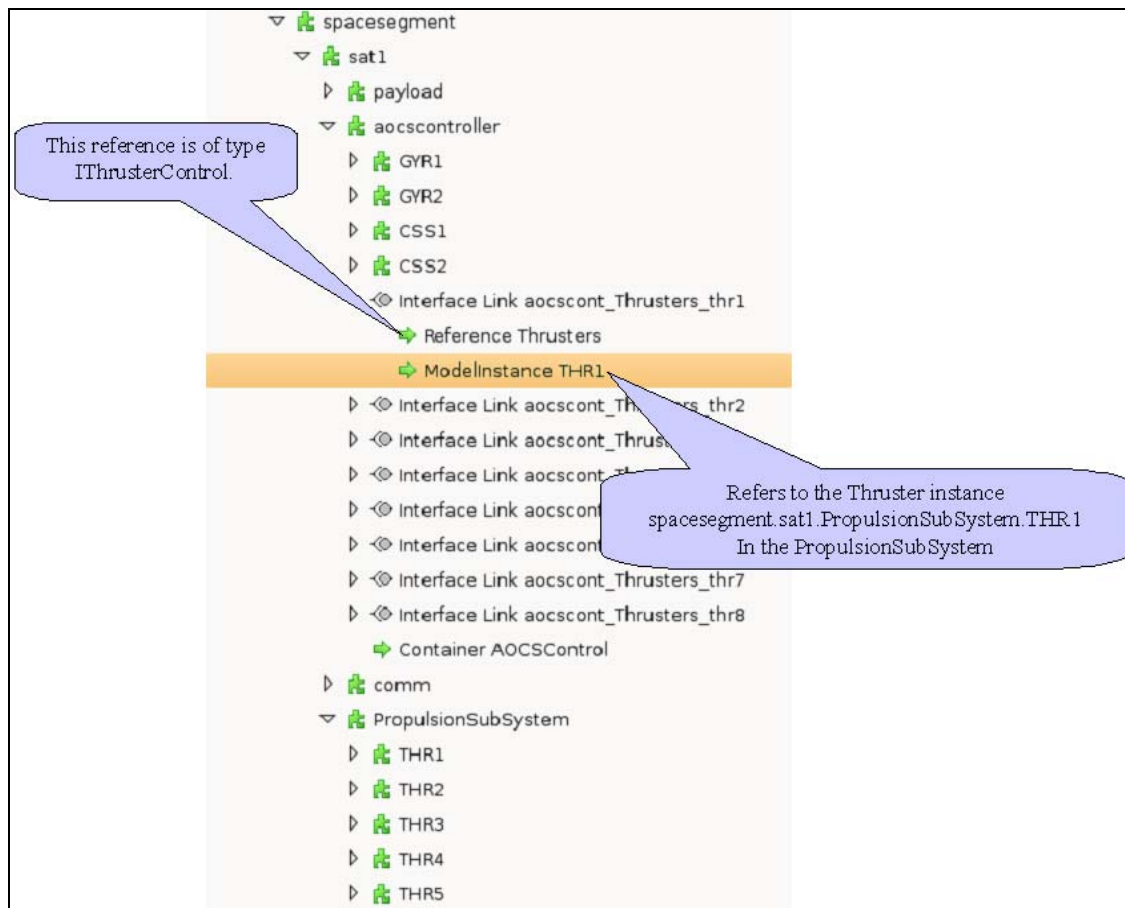


Figure 11-6: Integrating AOCSSControl With Propulsion Sub-system Thrusters

11.4.4 Re-use with dataflow

(See also Section 8.4.4 of this document)

Models that are to be integrated using SMP Dataflow-based design will need the integrator to create an assembly providing the FieldLinks between the fields of different models defined in a catalogue.

The integrator will be creating these FieldLinks between instances of these models that he has declared in an Assembly.

Figure 11-7 re-caps on the design of the housekeeping TM packet generating model (*HKTMPacketGenerator*) and the Equipment Model's housekeeping fields (i.e. *HKTM1*, *HKTM2*, *HKTM3*).

Property	Value																												
<ul style="list-style-type: none"> Model HKTmPacketGenerator <ul style="list-style-type: none"> Entry Point TransmitHKTM Array EquipmentTmPacket Array PktHdr Field Payload_HKTM1 <ul style="list-style-type: none"> Default Field Payload_HKTM2 Field Payload_HKTM3 Field Platform_HKTM1 Field Platform_HKTM2 Field Platform_HKTM3 Field PktForTrans Field HKPktHdr Model Equipment <ul style="list-style-type: none"> Field HKTM1 Field HKTM2 																													
<table border="1"> <thead> <tr> <th>Property</th> <th>Value</th> </tr> </thead> <tbody> <tr> <td colspan="2">General</td> </tr> <tr> <td>Description</td> <td>HK tm item 1 for Payload Equipment.</td> </tr> <tr> <td>Id</td> <td>ID_1dc8abfa-0bca-45ca-9d28-87a08222dd07</td> </tr> <tr> <td>Name</td> <td>Payload_HKTM1</td> </tr> <tr> <td colspan="2">Misc</td> </tr> <tr> <td>Input</td> <td>true</td> </tr> <tr> <td>Output</td> <td>false</td> </tr> <tr> <td>State</td> <td>true</td> </tr> <tr> <td>View</td> <td>true</td> </tr> <tr> <td colspan="2">Modelling</td> </tr> <tr> <td>Visibility</td> <td>private</td> </tr> <tr> <td colspan="2">Value</td> </tr> <tr> <td>Short Value</td> <td>0</td> </tr> </tbody> </table>		Property	Value	General		Description	HK tm item 1 for Payload Equipment.	Id	ID_1dc8abfa-0bca-45ca-9d28-87a08222dd07	Name	Payload_HKTM1	Misc		Input	true	Output	false	State	true	View	true	Modelling		Visibility	private	Value		Short Value	0
Property	Value																												
General																													
Description	HK tm item 1 for Payload Equipment.																												
Id	ID_1dc8abfa-0bca-45ca-9d28-87a08222dd07																												
Name	Payload_HKTM1																												
Misc																													
Input	true																												
Output	false																												
State	true																												
View	true																												
Modelling																													
Visibility	private																												
Value																													
Short Value	0																												

Figure 11-7: Fields of the HouseKeeping PacketGenerator

Figure 11-8 shows how the integrator creates an Assembly with a instance of the *HKTMPacketGenerator* model called *EquipHKTMGen* and two instances of *Equipment* Model called *PlatformEquipment* and *PayloadEquipment*. The integrator then creates the FieldLinks for these models by linking the Inputs to the *EquipHKTMGen* instance to the outputs of the *PayloadEquipment* and *PlatformEquipment* instances.

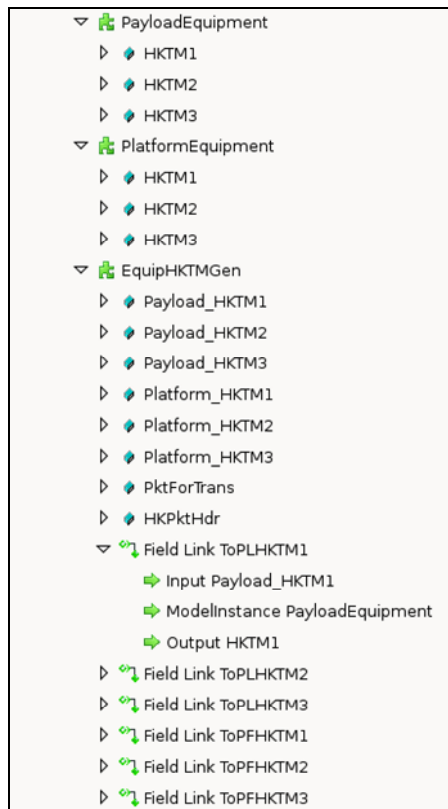


Figure 11-8: Housekeeping Packet Fields links

11.4.5 Re-use with events

(See also Section 8.4.5 of this document)

Where models are to be integrated using SMP Event-based design techniques, the job of the integrator will be to connect up their *EventSources* and *EventSinks*.

In our example the *PowerDistribution* model provides a *MainBusPower* *EventSource* and the *Equipment* models have an *EventSink* called *PrimaryPower* (shown in Figure 11-9).

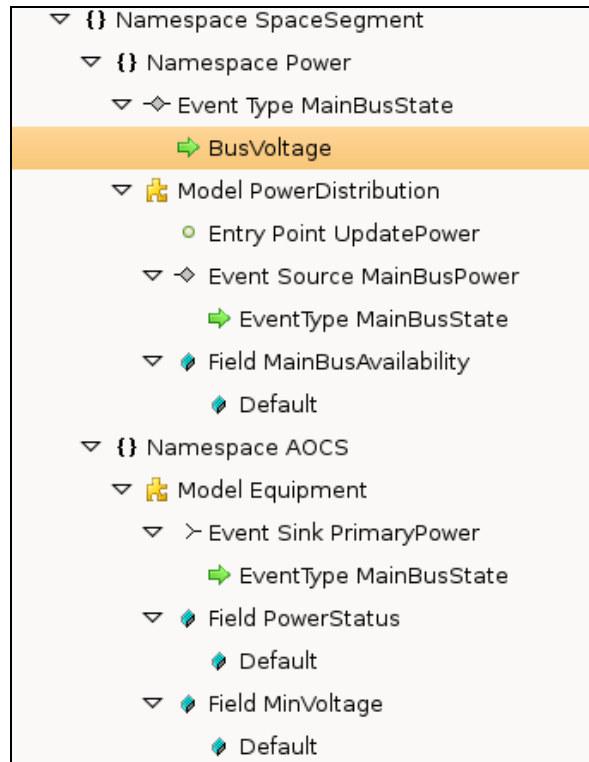


Figure 11-9: Power Sub-system MainBusPower EventSource

The integrator then creates the *EventLinks* between the *PrimaryPower* *EventSinks* of the *PayloadEquipment* and *PlatformEquipment* instances and the *PowerDistribution* instance *MainBusPower* *EventSource* as shown in Figure 11-10.

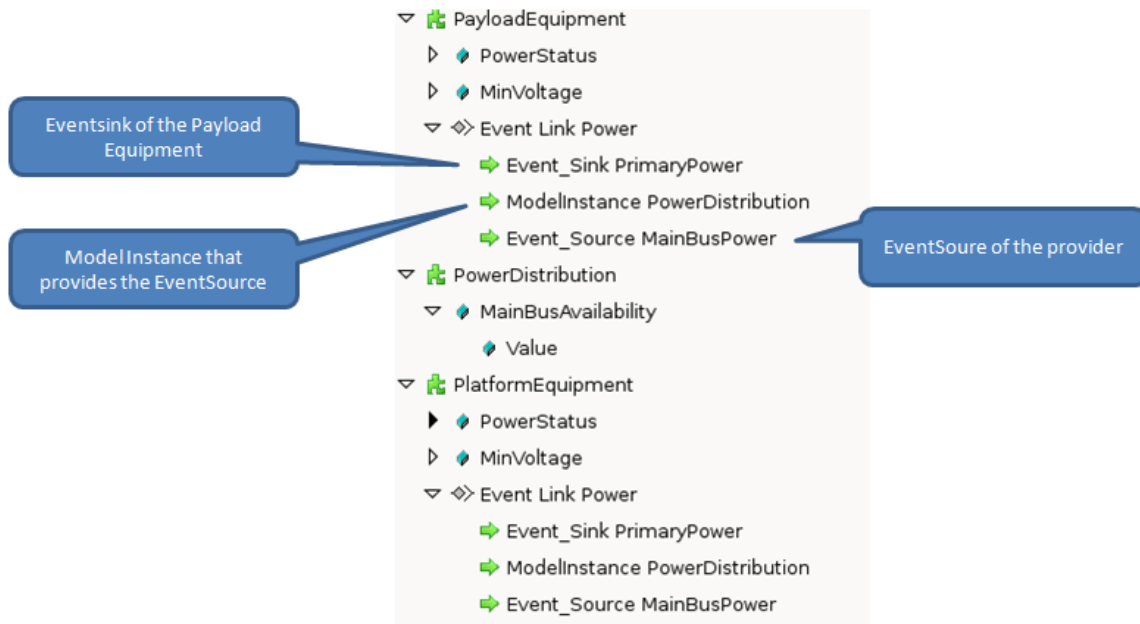


Figure 11-10: Creating Links for the Power Sub-system

The simulator can be partitioned up by using separate catalogues for interfaces and models.

11.5 Test complete simulation

11.5.1 Simulator Integration

After all models have been integrated and tested the simulation is ready for verification. As this is the first opportunity to run the complete simulation, with all the models integrated, the scope of testing is extended to cover:-

- System end-to-end functionality covering all inter-model interfaces
- Accuracy of outputs against requirements
- System performance and schedule optimisation

There are a number of ways that steps between integrating the various parts of the simulator and testing the complete simulator can be achieved. These steps are not prescribed by the SMP standard.

For this example we have chosen to illustrate some other SMP features by assuming that the complete simulator is integrated as a number of separate sub-systems provides by various in-house and outside parties.

The table below list the models that have been provided for integration and the relationship between the SMP design and implementation elements in our example.

Table 11-2: Example Simulator Integration Elements

Supplier	Sub-system	Models	Catalogue	Package	Binary
1	AOCS	AocsController	Aocs_Control	Pkg_supplier1	Pkg_supplier1.so
1	PSS	Battery, PSS, PowerDistribution	Pss_EPS	Pkg_supplier1	Pkg_supplier1.so
2	AOCS	CSS	Aocs_Sensors_Actuators	PackageCSS	PackageCSS.so
2	AOCS	GYR	Aocs_Sensors_Actuators	PackageGYR	PackageGYR.so
2	Environment	SatelliteEnvironment	Aocs_Sensors_Actuators	PackageENV	PackageENV.so
2	Propulsion	PropulsionSubSytem, Thruster	Aocs_Sensors_Actuators	PackageTHR	PackageTHR.so
2	Communications	Transmitter	Com_Trans	PackageTransmitter	PackageTransmitter.so
3	PSS	Solar Array	Pss_SA	Pkg_supplier3	Pkg_supplier3
3	Communications	CommunicationSubSytem, Receiver, FixAntenna	Com_Gen	Pkg_supplier3	Pkg_supplier3
3	GroundSegment	GroundStation, Visibility	Ground_Gen	Pkg_supplier3	Pkg_supplier3
3	All		common	Pkg_common	Pkg_common.so

The columns in this table show:

- **Supplier** – who provided the model implementations
- **Sub-system** – how the design organised the models in the simulator architecture
- **Models** – the models that the sub-system contains
- **Catalogue** – which catalogues the models were defined in
- **Package** – how the supplier has partitioned the model implementations
- **Binary** – the final model library that will need to be installed in the SMP infrastructure for the integrator to be able to test them.

The choices for the way in which to integrate the simulator are not prescribed by SMP, this is a choice for the architect and the project.

However, the table illustrates the relationships between the design elements and their final realisation as executable components ready to be tested.

The approach to using the SMP techniques in this respect needs to be considered carefully by the project or organisation especially where re-use is concerned.

Of particular importance is the partitioning of the Catalogues as these are the main elements shared between the different parties exchanging models.

A recommended practice is to separate the Catalogues containing the interfaces of the models to be used in the simulator from the Catalogues containing the definitions of the models using those interfaces. In our example such definitions are contained in the “common” Catalogue. This enables a common and re-useable set of interfaces to be defined and reused across different projects. If there are separate Catalogues containing the various model definitions these can be project specific meaning that eventually the components realising the interfaces can be defined and packaged in different ways according to the needs of different simulators or projects.

The example table above also illustrates that granularity of the final packaging is also flexible. For example some packages contain only individual model realisations (e.g. *PackageCSS* and *PackageGYR*) while others contain parts of

different sub-systems (e.g. *PackageSupplier1* and *PackageSupplier3*). Others still could contain complete subsystems (n.b. not in the example).

11.5.2 Sub-Assemblies Simplify Integration

In the example we have a single large Assembly that defines all of the instances and links needs to integrate the complete simulator. However, SMP contains the notion of *sub-Assemblies* as well.

A *sub-Assembly* can be used to encapsulate the integration of, for example, a complete sub-system. This way the integrator integrating the complete simulator could receive the sub-Assembly files from the suppliers of the sub-systems and focus on integrating only the sub-systems and treat their contents as “black boxes”. Treating the sub-systems in a “black box” in this way of course depends on the design of the sub-system interfaces.

11.6 Verify simulator

This activity involves running the simulator in its full delivery configuration as the “system under test”. Formal test are performed against the system requirements and the Requirement Baseline.

For this activity there is minimal specific inter-action with SMP artefacts.

Annex A (informative)

Model Development Guidelines

A.1 Model Development Guidelines

This section provides the complete set of SMP model development guidelines. For each guideline a description, rationale and compliance test are provided.

This section does not discuss the portability aspects of individual languages, as it is assumed that the model developers will be sufficiently experienced in their own language to be aware of language specific portability issues.

A set of general guidelines is presented here, that provide a general checklist of coding practices and compliance tests, aimed at maximising code portability and re-use. The guidelines are presented in the following subsections.

A.2 Provide Documentation

Good documentation is one of the most important factors for software reuse as it improves the ease of understanding it. Software that is easy to understand is much more likely to be reused than software that is incomprehensible. For this reason we recommend that each model is accompanied by an SMDL Catalogue file and additional documentation describing the model and its use. Adherence to some documentation template (such as that previously defined for SMP1) is advised. It is recognised that the required documentation highly depends on the deployment process adopted within the organisation, and that the template needs to be adapted to SMP. It may become part of a future release of this standard.

Guideline:	Provide SMDL Catalogue file and complete the document template.
Rationale:	Good documentation increases ease of understanding and correspondingly the re-use potential.
Compliance Test:	Inspect catalogue and documentation.

A.3 Use Standard Languages

SMP is a platform independent standard. However, the ANSI/ISO C++ mapping (see Volume 4: C++ Mapping) is currently the only platform mapping that is part of the standard. Therefore, ANSI/ISO C++ can be seen as today's SMP

language integration platform. As SMP models are typically developed by different organizations, it is unrealistic to require every model to be implemented in ANSI/ISO C++. However, every SMP compliant model implementation must have an ANSI/ISO C++ interface. Models written in other languages can be integrated into an SMP environment by providing an appropriate ANSI/ISO C++ wrapper, where certain rules have to be followed.

- Guideline:** SMP models must have an ANSI/ISO C++ interface. When using other languages for the implementation of model functionality, one of the following recommended standard languages should be used, provided that binary integration with ANSI/ISO C++ is possible via an appropriate ANSI/ISO C++ wrapper:
- ANSI/ISO C++ (required for the interface)
 - ANSI C
 - Fortran 77/90/95/2000
 - Ada 83/95
- Rationale:** Using one of these standard languages maximises the chances of compiler availability and reuse on another platform.
- Compliance Test:** Compilation with one of the standard compilers and integrating into an ANSI/ISO C++ wrapper using an ANSI/ISO conformant C++ compiler on the target platform.

A.4 Avoid Compiler Specific Language Extensions

The restriction to a set of standard languages is not sufficient to ensure that re-compilation across platforms is achievable. Different compilers often have language extensions to “improve” the language. While the features are often useful, they are usually compiler specific and hence reduce portability of the code.

- Guideline:** SMP models should use the particular language’s standard language features only.
- Rationale:** Restricting the code to standard language features only maximises the chances of recompilation in different compilers.
- Compliance Test:** Compile with compiler option to flag non-standard language features as errors.

A.5 Use Standard Libraries Only

Restricting code development to standard language features maximises the possibility of re-compilation with different compilers. However any realistic code development will need to make use of standard libraries to perform essential tasks.

Guideline: SMP models should only use the libraries listed in the following table

Language	Standard Libraries
ANSI/ISO C++	Standard C Library Functions Standard Template Library (STL)
ANSI C	Standard C Library Functions
Fortran 77/90/95/2000	Standard Fortran Library Functions
Ada 83/95	Library packages recommended in LRM

Rationale: Use of standard language libraries should ensure their availability on each platform.

Compliance Test: Search for included library files in source code and verify by inspection that only standard libraries are referenced. Alternatively, try and link the model into a system that is known to have only standard libraries.

A.6 Limit Use of Essential Non-standard Libraries

Sometimes it is completely impractical to develop a model without using a library that is not one of the standard libraries. Where a non-standard library is used, the library should itself be equally portable or widely available so that the model's portability is not compromised by its use.

Guideline: Non-standard libraries should only be used if they are themselves portable. The library should either be provided with the model, or the dependence clearly documented.

Rationale: The model can use non-standard libraries and still maintain portability if the libraries themselves are portable.

Compliance Test: Investigate library availability on different platforms or verify its portability to different platforms.

A.7 Use the SMP Interfaces to interact with the Simulation Environment

The models need to interact with the simulation environment to receive the support services that it provides. SMP provides standardised interfaces for the model to use, so that it may interface to the environment in a fixed way. The

SMP interfaces (component model and simulation services) act as an adapter translating the standardised environment actions to native environment specific ones.

- Guideline:** All interactions with the environment should be through SMP standard interfaces.
- Rationale:** By using only SMP standard interfaces the model will be more portable.
- Compliance Test:** Search for used interfaces/services and verify that they are SMP interfaces/services.

A.8 Avoid Direct Input / Output Operations

Unfortunately even with standard languages, input / output operations remain prone to portability problems. For example, file-opening operations allow operating system specific modes of file operation even with standardised languages.

Because of the difficulty in producing code that deals with input / output that is reliably portable, input / output should be avoided, though in practice, it is recognised that this may not always be achievable.

In cases where avoidance is not possible, file input should be done external to the model, and SMP mechanisms shall be used to get data from files into the models. Output should be through setting published field values or using the logger service only.

- Guideline:** Avoid input / output operations.
- Rationale:** Removing all input and output operations from the model to prevent potential portability problems.
- Compliance Test:** Search for file-access commands (such as open) in the source code.

A.9 Do Not Rely on Internal Representation of Data

Even if standard language and standard library features are being used, it is still possible to produce code that is not portable. One of the main causes of this is the reliance on the representation in memory of data items.

A common example of this type of reliance is mapping different variables to the same location, in order to perform conversions. In addition, the following elements are dangerous in C/C++:

C/C++ bitfields

C/C++ structures: In particular these may contain padding between components.

C/C++ enums: These can have rather variable sizes

C/C++ unions: In particular it is often thought that it is safe to store a value into one component of a union and then to read it using a different component. The C standard is quite explicit that the result of this is undefined.

- Guideline:** Do not write code that relies on the representation of data in memory.
- Rationale:** The representation of data in memory is compiler specific and cannot be guaranteed to be the same across platforms.
- Compliance Test:** General inspection. Check for language constructs that allow the mapping of variables to the same location.

A.10 Avoid Global Data Declarations

When integrating models together, name clashes will often occur if models declare data names that are visible in the global namespace. Because it is quite likely that two models could have data items with the same name, the model data should be kept out of the global data namespace.

- Guideline:** Avoid putting data names into the global namespace.
- Rationale:** Global data names lead to name clashes, which reduces the re-use potential of the model.
- Compliance Test:** Search for global data declarations and list all global data declarations.

A.11 Avoid Common Global Names

With some languages it is necessary to place the model operation names in the global namespace, as otherwise they would not be visible at all. When naming items that must be placed in the global namespace they should be provided with a unique prefix that identifies that the service is related to the model.

- Guideline:** Give all global names associated with the model the same prefix.
- Rationale:** The prefix should maximise the chances of the names being unique, reducing the chances of name clashes and hence, maximising re-use potential.
- Compliance Test:** Search for all global name declarations and verify they all have the same prefix.

A.12 Enable Multiple Instances

Quite often a model user may want several instances of the same model in their simulation. It may be that a model is originally developed in a system where

only a single instance is required. It is then quite possible that a future user could not have multiple instances of that model in their simulation.

- Guideline:** If it is likely that multiple instances are going to be required in a future system, then design the model so that multiple instances of the model may exist in the final system.
- Rationale:** The re-use potential for a model that is likely to be used with multiple instances is improved by designing it so that multiple instances are possible.
- Compliance Test:** Incorporate multiple instances of model in test harness. Run model tests on several instances and verify they behave correctly and independently.

A.13 Minimise Number of Model Interfaces

Minimising the number of interfaces to the model improves portability and re-use potential by making the model simpler to understand and use.

- Guideline:** Design the model so that the developer needs only a few interfaces to use the model.
- Rationale:** Fewer interfaces simplify the protocol of model use, which improves re-use potential by making the model understandable.
- Compliance Test:** Count the services to the model.

A.14 Simplify Model Interfaces Provided

Minimising the number of interfaces will not improve the ease with which the model can be used if the interfaces are very complex. As well as trying to reduce the number of interfaces, the complexity of each interface should be kept as simple as possible.

- Guideline:** Design model interfaces to be as simple as possible.
- Rationale:** Simpler interfaces simplify the protocol of model use, which improves re-use potential by making the model understandable.
- Compliance Test:** Get the interface reviewed by another developer.

A.15 Only Select Suitable Candidates for Reuse

Experience producing generic models has shown that what constitutes reusable models is not always immediately obvious. For example AOCS subsystems exist on every spacecraft in some form or other, and so one might expect that an AOCS subsystem is a good candidate for a generic model.

After some investigation it was found that producing a general overall generic model was unfeasible as the numerous variety of control law sensors and actuators made it unfeasible to provide a black box generic AOCS model. Instead it was decided a library of generic AOCS components should be produced, as it seemed that these could be reasonably useful. The wide diversity of individual AOCS components meant each one was provided with numerous tailoring features and functionality that could be adjusted to be able to make the models represent all forms of the particular AOCS component.

After providing the generic AOCS models to the first users it was found that due to the complexity of tailoring capabilities, the users had to perform more work to tailor the generic AOCS components to perform the specific task required. The AOCS generic model showed that there is a point at which the reuse requirements make the model so generic that its complexity of use outweighed the reuse benefits.

Guideline:	Only make models reusable if they will be effective to reuse.
Rationale:	It can actually be <i>more</i> difficult to reuse a model when it needs more tailoring for specific use than is required to develop the functionality from scratch.
Compliance Test:	None.

Bibliography

The following non-normative documents are referenced in this document

ECSS-E-TM-40-07 Volume 1A	Simulation Model Platform – Volume 1: Principles and requirements
ECSS-E-TM-40-07 Volume 2A	Simulation Model Platform – Volume 2: Metamodel
ECSS-E-TM-40-07 Volume 3A	Simulation Model Platform – Volume 3: Component model
ECSS-E-TM-40-07 Volume 4A	Simulation Model Platform – Volume 4: C++ mapping