Open Simulation Framework

openSF

ARCHITECTURAL DESIGN DOCUMENT

Code : openSF-DMS-ADD-001
Issue : 2.2
Date : 15/01/2014

	Name	Function	Signature
Prepared by	Enrique del Pozo	Technical Manager	
	Rui Mestre	Project Engineer	
Reviewed by	Ricardo Moyano	Review Team	
Approved by	Ricardo Moyano	Project Manager	
Signatures and approvals on original			

DEIMOS Space S.L.U.
Ronda de Poniente, 19, Edificio Fiteni VI, 2-2ª
28760 Tres Cantos (Madrid), SPAIN
Tel.: +34 91 806 34 50 / Fax: +34 91 806 34 51
E-mail: deimos@deimos-space.com

RESTRICTED

This page intentionally left blank

Document Information

Contract Data	
Contract Number:	22852/09/NL/FF
Contract Issuer:	ESA/ESTEC

Internal Distribution		
Name	Unit	Copies
Jose Antonio González Abeytua	Head of the Earth Observation Systems Business Unit	1
Ricardo Moyano	Earth Observation Systems Business Development	1
Enrique del Pozo	Earth Observation Systems	1
Rui Mestre	Earth Observation Systems	1
Internal Confidentiality Level (DMS-COV-POL05)		
Unclassified ☐	Restricted ☑	Confidential ☐

External Distribution		
Name	Organisation	Copies
Raffaella Franco	ESA/ESTEC	1
Lavinia Fabrizi	ESA/ESTEC	1
Paolo Bensi	ESA/ESTEC	1

Archiving	
Word Processor:	MS Word 2000
File Name:	openSF-DMS-ADD-001-22.doc

Document Status Log

Issue	Change description	Date	Approved
1.0	First issue of this document	21/12/2009	
1.1	Updated according to RID discussion during AR1. The following RIDs have been responded: ❑ OSF-AR1-20: Formatting of section 3.1.1.1, in what respects to the list of model elements. ❑ OSF-AR1-20: Formatting of Figure 5-17	15/03/2010	
1.2	Updated for openSF version 2.0 ❑ Removed Instrument concept ❑ Added architecture details about the openSF multi-repository capabilities. ❑ Added Parameter Editor design	30/06/2010	
1.3	Update after openSF v2.0 PDR	15/09/2010	
1.4	Update after openSF AR 2 ❑ Added new section for OSFI architecture ❑ Added diagram showing high level architecture of openSF and applications related.	12/11/2010	
2.0	Updated for openSF V3.0 PDR: ❑ Added section 5.4 regarding Parallel Processing; ❑ Added section 3.2.5 covering the design patterns used to support the parallel processing design; ❑ Added section 8 on OSFEG – openSF Error Generation Libraries; ❑ Added section 5.6 covering the Design approach for openSF V3 additional functionalities (for the remaining requirements); ❑ Added section 5.7 reflecting the Migration to openSF V3 approach.	18/04/2013	

2.1	Update after ESA comments on openSF v3.0 PDR ❑ Updated section 5.4.1 with details on the precautions model developers should have for ensuring that models are parallelizable; ❑ Added section 5.6.7 with the approach to Simplify session file and directory names. ❑ Updated section 5.6.1.5 with a more accurate definition of a Session for openSF v3.0.	05/06/2013	
2.2	Updated after ESA comments on openSF v3 AR meeting. Implemented the following RIDS: ❑ OSF-AR3-09: Updated section 8.3.2 with a reference to the error functions in the SRD; ❑ OSF-AR3-RF-07: Updated section 4.1 adding the openSF architecture evolutions through the different framework versions.	15/01/2014	

Table of Contents

List of Tables

List of Figures

1. INTRODUCTION

1.1. Purpose

This is the Architectural Design Document (ADD) for the openSF project as derived from the system requirements identified in [AD-SRD], and contains:

- Specification and introduction to design standards used for openSF.
- Description of the top-level architecture of openSF.
- Description of the major components of the system.
- Forward and inverse traceability matrices between ADD components and system requirements.

OpenSF has been developed using UML for modelling and Java as programming language. In addition it shall support models/algorithms written in C/C++, FORTRAN 77/90, Matlab or IDL.

1.2. Scope

The scope of openSF is to provide scientific users a framework that eases the orchestration and integration process of algorithm models within a complete E2E simulation chain:

- Consolidate the software engineering approach and architecture coming from the former ECSIM framework.
- Complete and improve the framework documentation.

This document shows all the software design issues for the development of openSF in response to the user requirements as defined in the Statement of Work [AD-SoW], and all related documentation.

This document is produced as part of the Acceptance Review (AR) Data Package. Therefore, it is applicable to the project from the AR onwards.

1.3. Document Structure

The document is structured as follows:

- Section 1 contains this introduction.
- Section 2 contains the list of applicable and reference documents, as well as the applicable standards to the project.
- Section 3 explains the design standards that shall be adopted for the architectural design. Here are presented the class diagrams and design patterns that are used in the next sections.
- Section 4 defines some useful general concepts to understand the solution reached in this moment of the design process. Overall, openSF is composed by models, simulations and sessions. Note that there is a logic sequence: simulations are made of models and sessions are made of simulations.
- Section 5 is a brief description of the system and its relationship with the surrounding components, that is, to provide a view of the system in its context.

- Section 6 explains the design of the Parameter Editor. This tool is aimed to ease the simulation definition and to provide a consistency checking mechanism prior launching the simulation chain. This section also contains a detailed architecture description of this openSF optional component. This software application has been developed in the frame of the Sentinel 3 Optical Simulator project (S3-OSPS contract TAS-F-1550001670).
- Section 7 contains the architecture design of the openSF Integration Libraries, OSFI from now on.
- Section 8 contains the architecture design of the openSF Error Generation Libraries, OSFEG from now on;
- Section 9 contains the forward and backward traceability matrices mapping system requirements with software architectural components identified in the previous section.

Through this document every term or component concerning the openSF software solution is written in *italics* to better sign them up.

1.4. Acronyms and Abbreviations

The acronyms and abbreviations used in this document are the following ones:

Acronym	Description
AD	Architectural Design Applicable Document
ADD	Architectural Design Document
API	Application Programming Interface
AR	Acceptance Review Analysis of Requirements
CM	Configuration Management Configuration Manager
COTS	Commercial Off-The-Shelf
DBMS	Database Management System
DMS	DEIMOS Space
E-R	Entity Relationship
GUI	Graphical User Interface
I/F	Interface
I/O	Input/Output
ICD	Interface Control Document
MDI	Multiple Document Interface
MMI	Man-Machine Interface
OO	Object-Oriented
OOAD	Object-Oriented Analysis and Design
OOP	Object-Oriented programming
PMS	Parameter Management System
RD	Reference Document
SCVR	Software Code Verification Report

Acronym	Description
SDD	Software Design Definition (Document)
SOW	Statement Of Work
SPR	Software Problem Report
SR	Software Requirements
SRD	Software Requirements Document
ST	System Test
SUM	System User Manual
SVTS	Software Validation Test Specification
SVVP	Software Verification & Validation Plan
SW	Software
TBC	To Be Confirmed
TBD	To Be Defined / Decided
TN	Technical Note
UML	Unified Modelling Language
VTS	Verification/Validation Test Specification
VTP	Verification/Validation Test Procedure
VTR	Verification/Validation Test Report
V&V	Verification & Validation

1.5. Definitions

The definitions of the specific terms used in this document are the following ones:

Definition	Meaning
Batch mode	It is the capability of the simulator to perform consecutive runs without continuous interactions with the user. Batch mode checks the agreement or not between the output of a given module and the input by the next one in the sequence of the simulation. Several modes of executions can be performed: ❑ Iteratively, executing one or more simulations ❑ Iteratively, executing the same simulation several times depending on the parameters configuration ❑ Same as above but by executing a batch script.
Configuration File	A small XML file that contains all the parameters necessary to execute a model. A configuration file instance must comply with the corresponding XML schema defined at model creation time.
Framework	Software infrastructures designed to support and control the simulation definition and execution. It includes the GUI, domain and database capabilities that enable to perform all the functionality of the simulator.

Definition	Meaning
Model	Executable entity that can take part in a simulation. A model can be understood, broadly speaking, also as an "algorithm". Basically, it contains the recipe to produce products function of inputs. A model contains also several rules to define the input, output and associated formats. Furthermore, its behaviour is controlled by one configuration file. Overall, the architecture of a model consists of: ❑ The source code and its binary compiled counterpart ❑ A configuration file with its parameters ❑ An input file that characterizes its inputs ❑ An output file that characterizes its outputs Models are not considered part of the framework.
Namespace	A logic entity used by the openSF Parameter Editor to specify a new parameter group. The number of namespaces in a configuration file is not limited but it is mandatory to specify at least one (XML root tag). Namespaces are specified as XML tags within a rules/configuration file.
OSFI	To integrate an external model into openSF, the models need to fulfill a series of interface requirements. The Open Simulation Framework Integration Libraries (OSFI from now on) will be used to ease the integration of models into the framework. The Integration Libraries activity will provide the model developer with a set of routines with a well-defined public interface hiding the implementation details. This set of routines is currently available in C++, ANSI C, Fortran 90 and 77, Matlab scripting language and IDL.
Parameter	A constant whose value characterizes a given particularity of a model. Parameters are user-configurable, they are fixed before launching a model and, for practical reasons, and not all of them shall be accessible from the HMI.
Repository	Set of entities involved in a single simulation chain. The entities found within a repository are descriptors, stages, parameters, models, sessions, simulations and tools. As functionality since openSF version 2, users can switch between different mission repositories within the same openSF instance.
Rule	A constraint or relationship applied to the parameters involved in a simulation chain. This entity is used to ensure the parameter consistency before executing a simulation chain.
Session	A session is defined as an execution of a simulation, an ordered set of simulations or an iterative execution of simulation(s) with different parameter values. There are no restrictions on how to concatenate these simulations, they do not have to be compatible between them but, if necessary, the final output files of a simulation can be used by the following simulation.
Simulation	A simulation is understood as a list of models (or even a model alone) that is run sequentially and produces observable results.
Stage	Entity that defines a phase in a simulation process. The stage order definition specifies the logic of the simulation sequence so a model must have associated a stage and a simulation will run the models of a stages series.

Definition	Meaning
Tool	A tool is an external executable file that performs a given action to a certain group of files. Used into the openSF platform and associated to a certain file extension these tools can be called to perform off-line operations to products involved in simulations.

2. RELATED DOCUMENTS

2.1. Applicable Documents

The following table specifies the applicable documents that shall be complied with during project development.

Table 1: Applicable documents

Reference	Code	Title
[AD-SRD]	openSF-DMS-SRD-001	OpenSF System Requirements Document
[AD-SUM]	openSF-DMS-SUM-001	OpenSF System User Manual
[AD-ICD]	openSF-DMS-ICD-001	OpenSF Interface Control Document
[AD-SVS]	openSF-DMS-SVS-001	OpenSF System Validation Specification
[AD-SoW]	EOP-SFP/2009-07-1404/MAr	Statement of Work for the OpenSF end-to-end Simulation Framework Maintenance
[AD-CCN1]	EOP-SFP/2012-12-1686/PB/ag	Change Request for the openSF V3 activities description.

2.2. Reference Documents

The following table specifies the reference documents that shall be taken into account during project development.

Table 2: Reference documents

Reference	Code	Title	Issue
[RD UML]	ISBN 0-201-57168-4	The Unified Modelling Language User Guide, Grady Booch, James Rumbaugh, Ivar Jacobson.	-
[RD ECSIM ICD]	ECSIM-DMS-TEC-ICD01-R	EarthCARE Simulator Interface Control Document	-
[RD-OSFI-DM]	OSFI-DMS-TEC-DM	OpenSF Integration Libraries Developers Manual	-
[RD GOF]	ISBN 0-201-63361-2	Design Patterns: Elements of Reusable Object-Oriented Software. E. Gamma, R. Helm, R. Johnson and J. Vlissides	-
[RD SWING]	http://java.sun.com/javase/technologies/desktop/	JAVA SWING technologies overview	-
[RD SWING API]	http://java.sun.com/j2se/1.6.0/docs/guide/swing/	JAVA SWING API specifications	-

Reference	Code	Title	Issue
[RD GAL MMI]	GAL-TD-GLI-SYST-A/1070	Guidelines for the development of man/machine interfaces for GALILEO applications	1

2.3. Standards

The following table specifies the standards that shall be complied with during project development.

Table 3: Standards

Reference	Code	Title	Issue
[ECSS-E40C]	ECSS-E-ST-40C	Space engineering - Software	3 - 6 March 2009

3. DESIGN STANDARDS

3.1. UML

The Unified Modelling Language (UML) has quickly become the de-facto standard for building Object-Oriented software.

UML is a graphical language for visualizing, specifying, constructing, and documenting the artefacts of a software-intensive system. UML offers a standard way to write a system's blueprints, including conceptual aspects such as business processes and system functions as well as concrete aspects such as programming language statements, database schemas, and reusable software components.

The main diagrams used in UML are described in the following sections, and in particular, in the design of openSF. For more information the reader is referred to any of the books and Internet links available in the extensive documentation about UML.

3.1.1. Types of diagrams

3.1.1.1. Class diagrams

Class diagrams are widely used to describe the types of objects in a system and their relationships. They model class structure and contents using design elements such as classes, packages and relationships.

The Class Model is at the core of object-oriented development and design – it expresses both the persistent state and the behaviour of the system. A class encapsulates a *state* (attributes) and offers services called *methods* to manipulate that state (behaviour). Good object-oriented designs limit direct access to class attributes and offers services, which manipulate attributes on behalf of the caller. This hiding of data and exposing of services ensures data updates are only done in one place and according to specific rules – for large systems the maintenance burden of code which has direct access to data elements in many places is extremely high.

The top compartment contains the class name; if the class is abstract the name is italicised. The middle compartment contains the *class attributes*. The bottom compartment contains the *class methods* (also called operations). Like the class name, if a method is abstract, its name is italicised. Depending on the level of detail desired, it is possible to omit the properties and show only the class name and its methods, or to omit both the properties and methods and show only the class name. This approach is illustrated in the figure below, whose initial version was already presented in Figure 3-1.

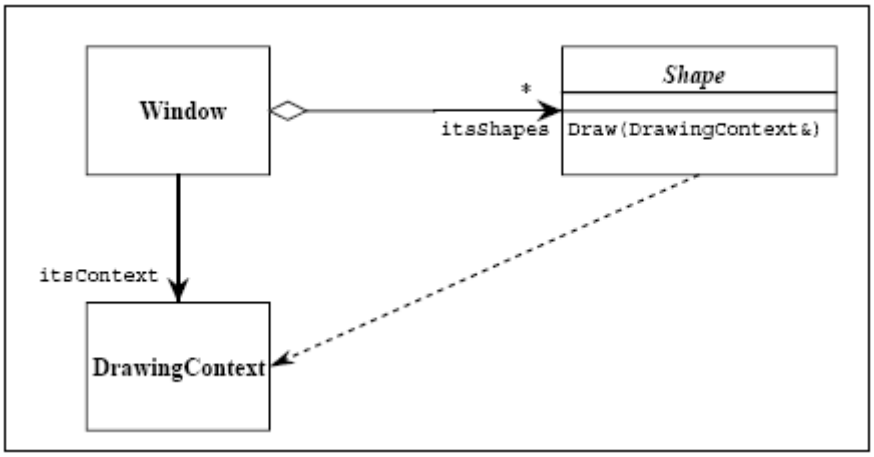

Figure 3-1: Class diagram

Class diagrams can consist of the following model elements:

- Packages. They are used to structure the model. They can also be placed into class diagrams to show this hierarchy more explicitly. Classes can then be nested inside them. Or they can exclusively be used in a diagram to express the interdependencies of packages.
- Dependencies between packages. This expresses that could be classes within a package using classes from the package it depends on.
- Classes. Classes are the most important concept of object-orientation and as well as of UML. Classes hold operations and attributes and have relations to other classes via association or inheritance relations. It has a few properties of its own like its name, a stereotype and a visibility, but the more important aspect is its relation to other classes.
- Inheritance relations. Between interfaces or between classes. This is not allowed between an interface and a class. A class can inherit properties and operations from a parent class or super-class.
- Implementation relations. Only between interfaces and classes.
- — Association relations. Associations are relations between classes. This type of relations can be specialized to an aggregation or a composition.

3.1.1.2. Sequence diagrams

Sequence diagrams represent a dynamic view of the system, showing object interaction in a time-based sequence of calls to methods provided by objects. They are used to cover all the operations offered to the user through the use cases. The operations exchanged in sequence diagrams are provided by objects (classes) represented in class diagrams.

A sequence diagram therefore shows a sequence of operation usage between class instances presenting how operations exported by class instances are used in a timely manner. It is constituted by:

- Objects (shown as a box plus a vertical bar) must correspond to instances of classes from the class usage diagram.
- Events (shown as a horizontal arrow) have to correspond to provide operations according to usage associations in the class usage diagram. Operations are labelled with the operation name used from the corresponding object.

3.2. Design Patterns

Design patterns form a cohesive language that can be used to describe classic solutions to common object-oriented design problems. By using design patterns to solve programming problems, the proper perspective on the design process can be maintained.

The Gang of Four described in [RD GOF] patterns as "a solution to a problem in a context". These three things – problem, solution, and context – are the essence of a pattern. For documenting the pattern it is additionally useful to give the pattern a name, to consider the consequences using the pattern will have, and to provide an example. Different cataloguers use different templates to document their patterns. Different cataloguers also use different names for the different parts of the pattern. Each catalogue also varies somewhat in the level of detail and analysis devoted to each pattern.

The design of openSF makes only use of creational patterns, which prescribe the way objects are created. These patterns are used when a decision must be made at the time a class is instantiated. Typically, the details of the classes that are instantiated – what exactly are they, how, and when they are created – are encapsulated by an abstract super-class and hidden from the client class, which knows only about the abstract class or the interface it implements. The specific type of the concrete class is typically unknown to the client class.

The design patterns used in the design of this System are expressed as stereotypes in the UML form.

3.2.1. Singleton Pattern

The singleton design pattern is used to restrict instantiation of a class to only one object. This is useful when exactly one object is needed to coordinate actions across the system. Sometimes it is generalized to systems that operate more efficiently when only one or a few objects exist.

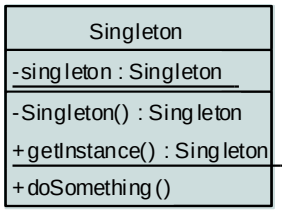

Figure 3-2: Singleton pattern example

The singleton pattern is implemented by creating a class with a method that creates a new instance of the object if one does not exist. If an instance already exists, it simply returns a reference to that object. To make sure that the object cannot be instantiated any other way, the constructor is made either private or protected. Note the distinction between a simple static instance of a class and a singleton. Although a singleton can be implemented as a static instance, it can also be lazily constructed, requiring no memory or resources until needed.

The singleton pattern must be carefully constructed in multi-threaded applications. If two threads are to execute the creation method at the same time when a singleton does not yet exist, they both must check for an instance of the singleton and then only one should create the new one. If the programming language has concurrent processing capabilities the method should be constructed to execute as a mutually exclusive operation.

An example for the usage of this pattern shall be the Logger class, where different classes need to access to the same log session when reporting the simulation messages during the session execution.

3.2.2. Factory Pattern

In addition to the Singleton pattern, another common example of a creational pattern is the Factory Method. This pattern is used when it must be decided at run-time which one of several compatible classes is to be instantiated.

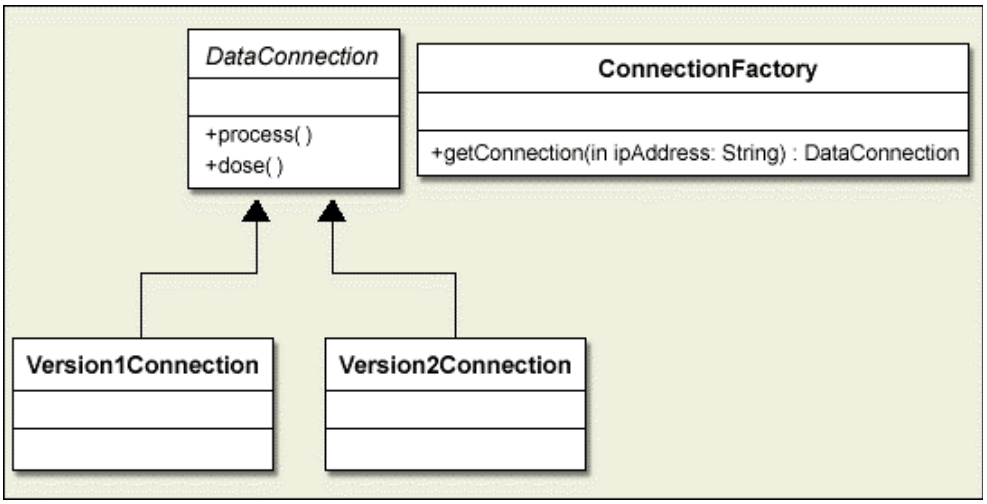

Figure 3-3: Factory Pattern example

3.2.3. Prototype Pattern

The Prototype design pattern consists on the cloning of an object avoiding the creation of it. This pattern is used when the cost of creating a new object is large and the clone of it is preferred.

Regarding the development of the framework this design pattern has been used in the visualization package when domain elements need to be edited or visualized and the cost of creation is large as it implies a SQL transaction.

Figure 3-4 shows a diagram illustrating a simple example that makes use of this design pattern.

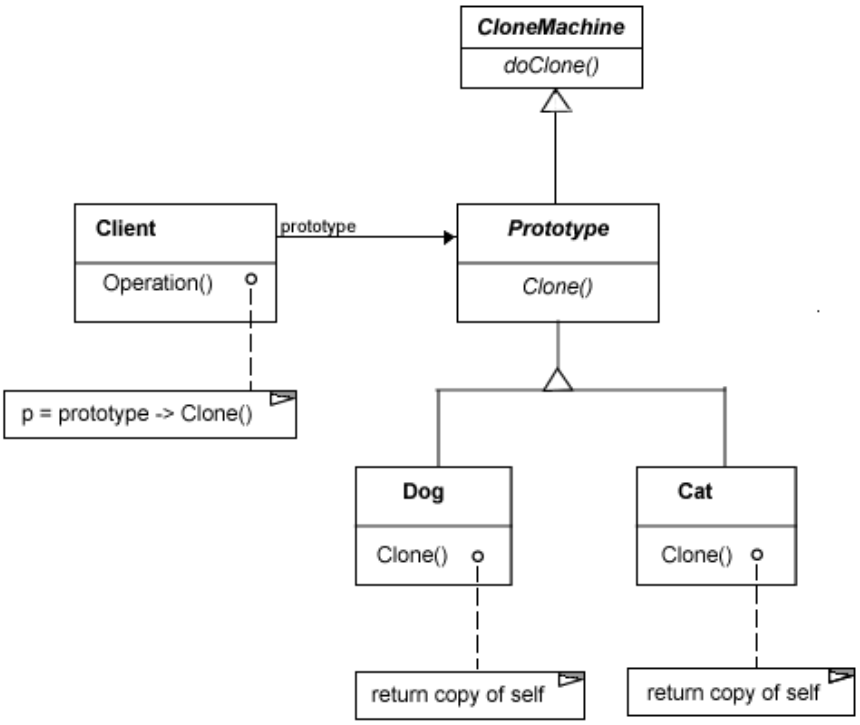

Figure 3-4 Prototype pattern example

3.2.4. Model-View-Controller Pattern

Model-View-Controller (MVC) is a design pattern used in software engineering. In complex computer applications that present lots of data to the user, one often wishes to separate data (Model) and user interface (View) concerns, so that changes to the user interface do not impact the data handling, and that the data can be reorganized without changing the user interface. The Model-View-Controller design pattern solves this problem by decoupling data access and business logic from data presentation and user interaction, by introducing an intermediate component: the Controller.

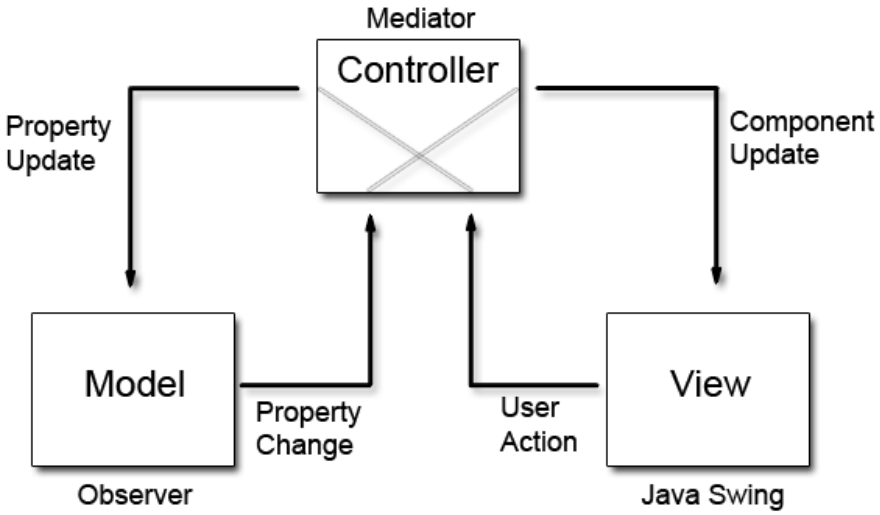

Figure 3-5: A simple diagram depicting the relationship between the Model, View, and Controller

The MVC design pattern (aka the MVC paradigm) is one of the oldest described patterns. It is usually based on top of smaller design patterns used in coordination with each other, such as the Observer pattern, the Command pattern, the Factory pattern, and the Facade pattern.

It is common to split an application into separate layers: presentation (UI), domain, and data access. In MVC, the layers become: View (UI), Controller, domain, and data access. The MVC pattern sees domain and data access as one single component: the Model.

MVC encompasses more of the architecture of an application than is typical for a design pattern. The components of the MVC pattern are:

- **Model**: the domain-specific representation of the information on which the application operates. The model is another name for the domain layer. Domain logic adds meaning to raw data (e.g., calculating if today is the user's birthday, or the totals, taxes and shipping charges for shopping cart items).

  Many applications use a persistent storage mechanism (such as a database) to store data. MVC does not specifically mention the data access layer because it is understood to be underneath or encapsulated by the Model component.

- **View**: renders the model into a form suitable for interaction, typically a user interface element. MVC is often seen in web applications, where the view is the HTML page and the code which gathers dynamic data for the page.

- **Controller**: processes and responds to events, typically user actions, and may invoke changes on the model and view.

Though MVC comes in different flavours, control flow generally works as follows:

1. The user interacts with the user interface in some way (e.g., user presses a button)
2. A controller handles the input event from the user interface, often via a registered handler or call-back.
3. The controller accesses the model, possibly updating it in a way appropriate to the user's action (e.g., controller updates user's shopping cart).
4. A view uses the model to generate an appropriate user interface (e.g., view produces a screen listing the shopping cart contents). The view gets its own data from the model. The model has no direct knowledge of the view. (However, the observer pattern can be used to allow the model to indirectly notify interested parties – potentially including views – of a change.)
5. The user interface waits for further user interactions, which begins the cycle anew.

3.2.5. Producer-Consumer Problem

The producer-consumer problem is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common queue. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time.

The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be reached by means of inter-process communication, typically using semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

3.2.5.1. Scheduler design pattern

The Scheduler pattern controls the order in which threads are scheduled to execute single threaded code using an object that explicitly sequences waiting threads. The Scheduler pattern provides a mechanism for implementing a scheduling policy. It is independent of any specific scheduling policy.

3.2.5.2. Producer-consumer design pattern

The Producer-Consumer pattern can be viewed as a special form of the Scheduler pattern that has scheduling policy with two notable features:

- The scheduling policy is based on the availability of a resource.
- The scheduler assigns the resource to a thread but does not need to regain control of the resource when the thread is done so it can reassign the resource to another thread.

3.2.5.3. Thread Pool design pattern

The general idea for the Thread Pool pattern is to use an object pool whenever there are several clients who need the same stateless resource which is expensive to create.

Figure 3-6 shows a diagram illustrating a simple example that makes use of this design pattern.

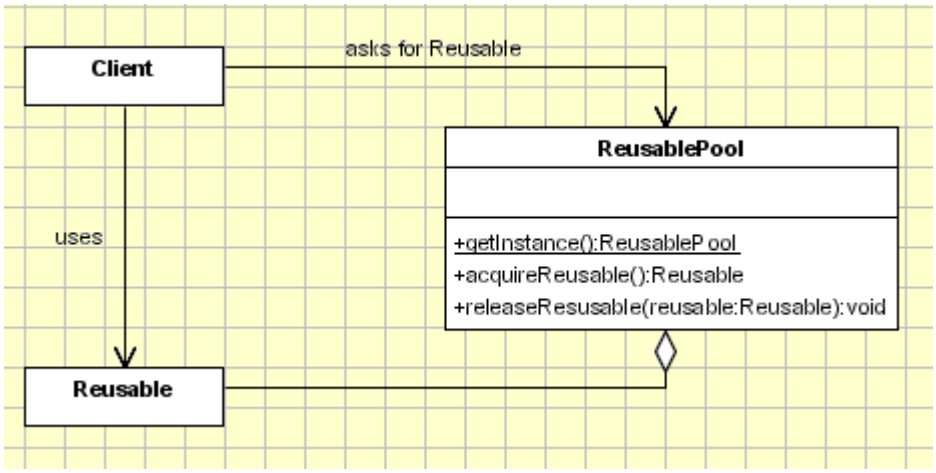

Figure 3-6 Thread Pool pattern example

The design pattern is composed of the following components:

- *Reusable* - Instances of classes in this role collaborate with other objects for a limited amount of time then they are no longer needed for that collaboration;
- *Client* - Instances of classes in this role use Reusable objects;
- *ReusablePool* - Instances of classes in this role manage Reusable objects for use by Client objects.

Usually, it is desirable to keep all *Reusable* objects that are not currently in use in the same object pool so that they can be managed by one coherent policy. To achieve this, the *ReusablePool* class is designed to be a singleton class. Its constructor(s) are private, which forces other classes to call its `getInstance` method to get the one instance of the *ReusablePool* class.

A *Client* object calls a *ReusablePool* object's `acquireReusable` method when it needs a *Reusable* object. A *ReusablePool* object maintains a collection of *Reusable* objects. It uses the collection of *Reusable* objects to contain a pool of *Reusable* objects that are not currently in use.

If there are any *Reusable* objects in the pool when the `acquireReusable` method is called, it removes a *Reusable* object from the pool and returns it. If the pool is empty, then the `acquireReusable` method creates a *Reusable* object if it can. If the `acquireReusable` method cannot create a new *Reusable* object, then it waits until a *Reusable* object is returned to the collection.

Client objects pass a *Reusable* object to a *ReusablePool* object's `releaseReusable` method when they are finished with the object. The `releaseReusable` method returns a *Reusable* object to the pool of *Reusable* objects that are not in use.

In many applications of the Thread Pool pattern, there are reasons for limiting the total number of *Reusable* objects that may exist. In such cases, the *ReusablePool* object that creates *Reusable* objects is responsible for not creating more than a specified maximum number of *Reusable* objects. If *ReusablePool* objects are responsible for limiting the number of objects they will create, then the *ReusablePool* class will have a method for specifying the maximum number of objects to be created.

3.3. XML Grammar

For the implementation of the openSF parameter management system a simple XML grammar is used to define the rules used as base for the consistency checking.

The definition of a rules grammar in XML language is a standard used in other software fields such as voice recognition software, translation software, syntax converters, etc. See [RD XML-Grammar] for a detailed description of these items.

The XML grammar used in the rules definition for consistency checking is depicted in section 6.2.1.

4. DESIGN OVERVIEW

This section gives a description of the design solution for this project. It starts with a brief definition of the openSF system plus some different views of it:

- OpenSF architecture evolution.
- Functional requirements view – emphasizing the functional requirements of the system from the user's point of view, including some use cases diagrams.
- Process view – highlighting data flow and data processing in the different execution scenarios.
- Deployment view – defining the physical decomposition of the system on the target platform.

The system is decomposed in a hierarchical structure, abstracting different components in levels. This section continues giving a description of those components, browsing them by levels.

Definition

In the frame of concept and feasibility studies for the Earth Observation (EO) activities, mission performance in terms of final data products needs to be predicted by means of so-called end-to-end (E2E) simulators.

A specific mission E2E simulator is able to reproduce all significant processes and steps that impact the mission performance and gets simulated final data products.

OpenSF is a generic simulation framework product aimed to cope with these major goals. It provides end-to-end simulation capabilities that allow assessment of the science and engineering goals with respect to the mission requirements.

Scientific models and product exploitation tools can be plugged in the system platform with ease using a well-defined integration process.

OpenSF provides a user-friendly framework that allows scientific users to integrate mathematical algorithms and satellite products within a complete simulation chain.

Figure 4-1 shows a high level diagram of the openSF system and the applications associated to it, openSF integration libraries and openSF Parameter Editor.

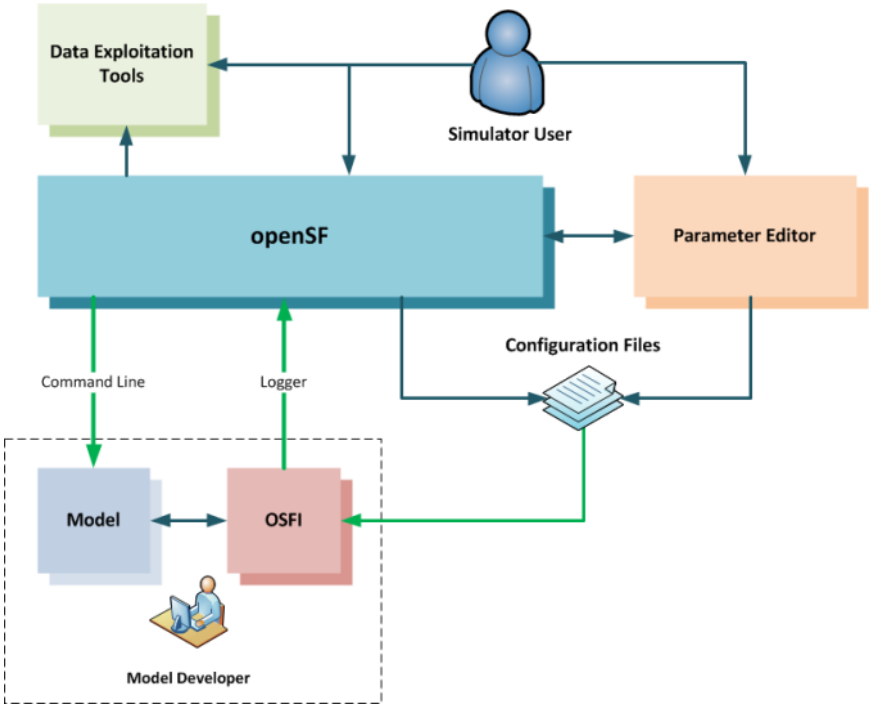

Figure 4-1: openSF High Level Architecture

4.1. Transition from the former to the openSF architecture

The openSF project is "descendent" of a previous ESA-funded project, called ECSIM, intended to provide an End-to-end simulator for the EarthCARE mission. OpenSF project must be fully compatible with the previous ECSIM project, so openSF interfaces are compatible with the ones defined in the ECSIM project ([RD ECSIM ICD]).

The main issue in openSF was to extract the abstract and re-usable simulation functionalities that ECSIM simulator had in common with other E2E simulator projects. Beyond this point and after months of requirements study and functionalities testing, the first operational version of OpenSF was released.

Since the original OpenSF development, it has been (and continue to be) used by other ESA projects (for example GERSI, AIPC, SEPSO, S3-OGPP/OSPS). For each of those projects OpenSF is adapted in order to fulfil the project requirements and some of those adaptations and ideas are now part of the OpenSF "*core*". New features are included in OpenSF after a deep examination (consequences, drawbacks, backward compatibility etc…) performed by the management and development team.

Regarding to the system architecture basically is similar to the approach in the former ECSIM framework but as mentioned before some aspects have been refined and re-designed in order to make the framework as much generic as possible. An example of this is the stage concept re-definition, third party tools manager etc…

4.1.1. Architecture Evolution

This section contains the evolution and changes of openSF from an architectural point of view, redefinition of concepts, new capabilities, constraints removal, etc...

The [RD ECSIM ICD] describes the initial architecture used as baseline for the first openSF version, from that point the following sections contain the different changes introduced by each new framework version.openSF 1.0 evolutions

- Introduction of **Stage concept**: in ECSIM simulation steps were fixed to Scene, Platform, Forward, Instrument and Retrieval.
- Introduction of a **global configuration file**: ECSIM only allowed to specify one single configuration file per Model. Thanks to AIPC project contribution, a global configuration file was added, containing the common parameters applicable and available to all models within a session.
- Removal of ECSIM specific items: Data viewers and post-processing tools.
- **OSFI libraries**: in the frame of CASPER project a set of libraries were developed for easing the integration of Models within openSF framework.

4.1.1.1. openSF 2.0 evolutions

- **Multiple databases**: in this version the possibility of handling multiple openSF databases, coping with the possibility of managing different simulators with the same openSF instance.
- **ParameterEditor**: coming from S3-OSPS project, a new GUI for editing XML configuration files was introduced.
- Support to new programming languages: initially only Fortran 90, C++ and C were supported. In version 2.0 the openSF Integration Libraries (OSFI) were migrated also to **Fortran77, IDL and MATLAB** programming languages, allowing models developed in those languages to be easily integrated within openSF.

4.1.1.2. openSF 2.2 evolutions

- **Parameter Perturbation**: in openSF v2.2 the main evolution was the addition to the openSF sensitivity analysis capabilities a set of statistical functions from SEPSO project.
- Related to ParameterPerturbation a new execution mode was introduced, allowing the execution of one single model a configurable number of times.

4.1.1.3. openSF 3.0 evolutions

- **Parallel Processing**: using the Java tools for thread scheduling and synchronization a new capability was added for executing models in parallel (in different CPU cores).
- **Enhanced framework flexibility**: coming from the feedback of openSF users a need for enhancing the flexibility was detected. In previous version there were strong constraints in the Stage, Simulation and Session definition. A simulation should be a concatenation of models strictly following the stage order and a session should be formed by one or more simulations. In openSF v3.0 these restrictions were removed and a session can be formed by simulations following the stage order (as in previous versions) but also can be an arbitrary concatenation of of models.
- Removal of Logs from the database: in order to improve the framework performance, log messages were removed from the openSF database and now are stored in a plain-text file within the session folder.

4.2. Functional Requirements

This view emphasizes the functional requirements of the system from the user's point of view.

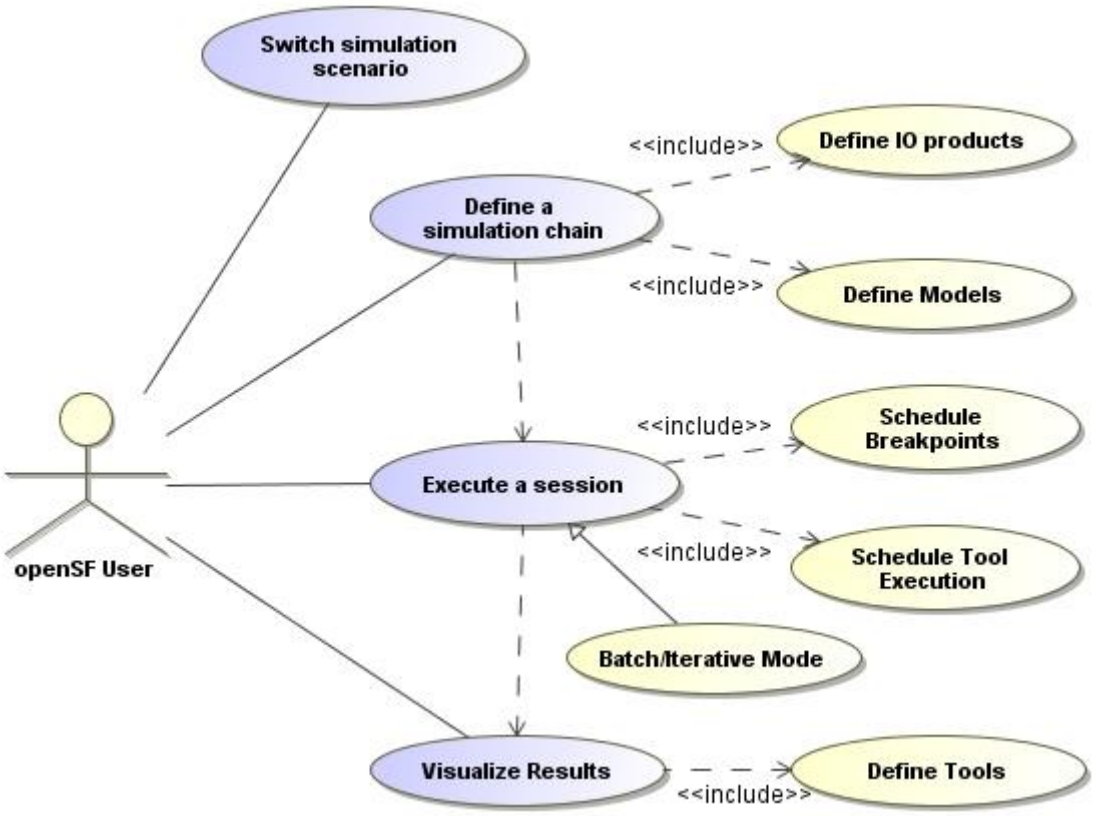

Figure 4-2: Use cases diagram.

Figure 4-2 illustrates the use case diagram for openSF, which presents the context the whole system, and the actor that interacts with it. In this figure only the most relevant use cases have been depicted and those are:

- ***Define a simulation chain**:* case that is referred to the whole simulation creation process including input/output, model and simulation chain definition.
- ***Execute a session**:* this case covers the session definition and its execution.
- ***Visualize results**:* represents the user capabilities for product visualization and result exploitation.
- ***Switch simulation scenario:*** this case represents the user capability to switch between repositories retrieving different simulation scenarios.

Please notice the "include" dependencies (denoted by a dashed line) between two use-cases means that the former contains some steps defined in the latter one. For example, in case of the "Execute a session" and "Schedule Breakpoints" cases, a session can be defined scheduling breakpoints before it is executed.

4.3. Process View

Process view highlights the data processing approach used in openSF.

4.3.1. Models and simulations

As described in the [AD-ICD] openSF work with models and simulations as its main significant concepts.

- Models
- Simulations

A **model** is an independent program that performs a scientific function that is appropriate within openSF accepting input files and configurations and generating output files.

OpenSF models are integrated into processing chains or **simulations**, which means that several models are linked in a logical sequence of stages (or model types) following a defined order.

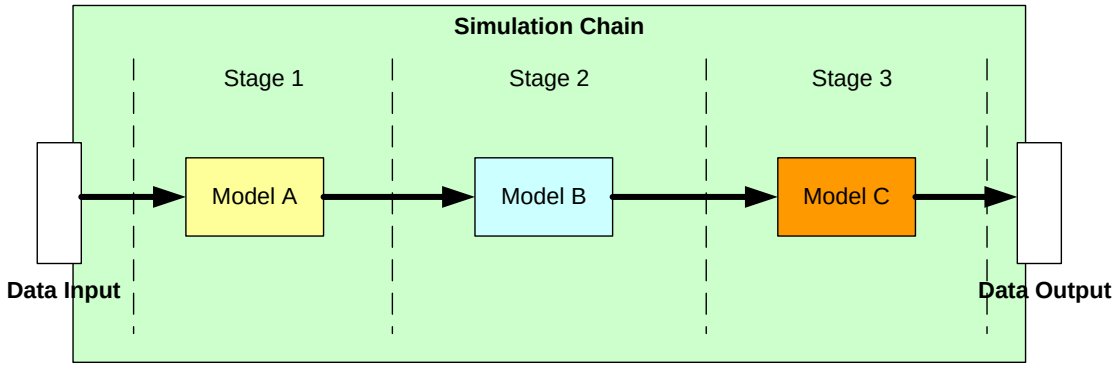

Figure 4-3: Sequence of simulation stages

Note that a user can choose the starting and ending stages as demands. The case above shows a three stages simulation, but it can happen that a user would like to try models of only a part of this string even a single model simulation. With this solution a user has only to choose the start and end stages conveniently as long as fulfils the dependencies between them.

During the current design phase the design team found useful to introduce two new concepts: sessions and products tools. These concepts cover in simpler way requirements as sequence of simulations, simulations in batch and iterative simulations and plotting, comparing or viewing products. They are further described in the next sections.

4.3.2. Session

As defined earlier, a simulation is an ordered sequence of models. Similarly to this concept, a session is an execution of an ordered set of simulations, with certain parameters, inputs, outputs, log messages and results. There are no restrictions on how to concatenate these simulations, they do not have to be compatible between them but, if necessary, the final output files of an execution of a simulation can be used by the following simulation as inputs since they are executed sequentially.

This concept will define precisely these situations:

- A simulation is a sequence of models grouped for families (stages) and connected with a network of dependencies. See Figure 4-3.

- A sequence of simulations is just that, a session including some simulations being executed one after another. There are no restrictions on how to combine them, so it is the responsibility of the user on providing them with correct inputs and configurations.

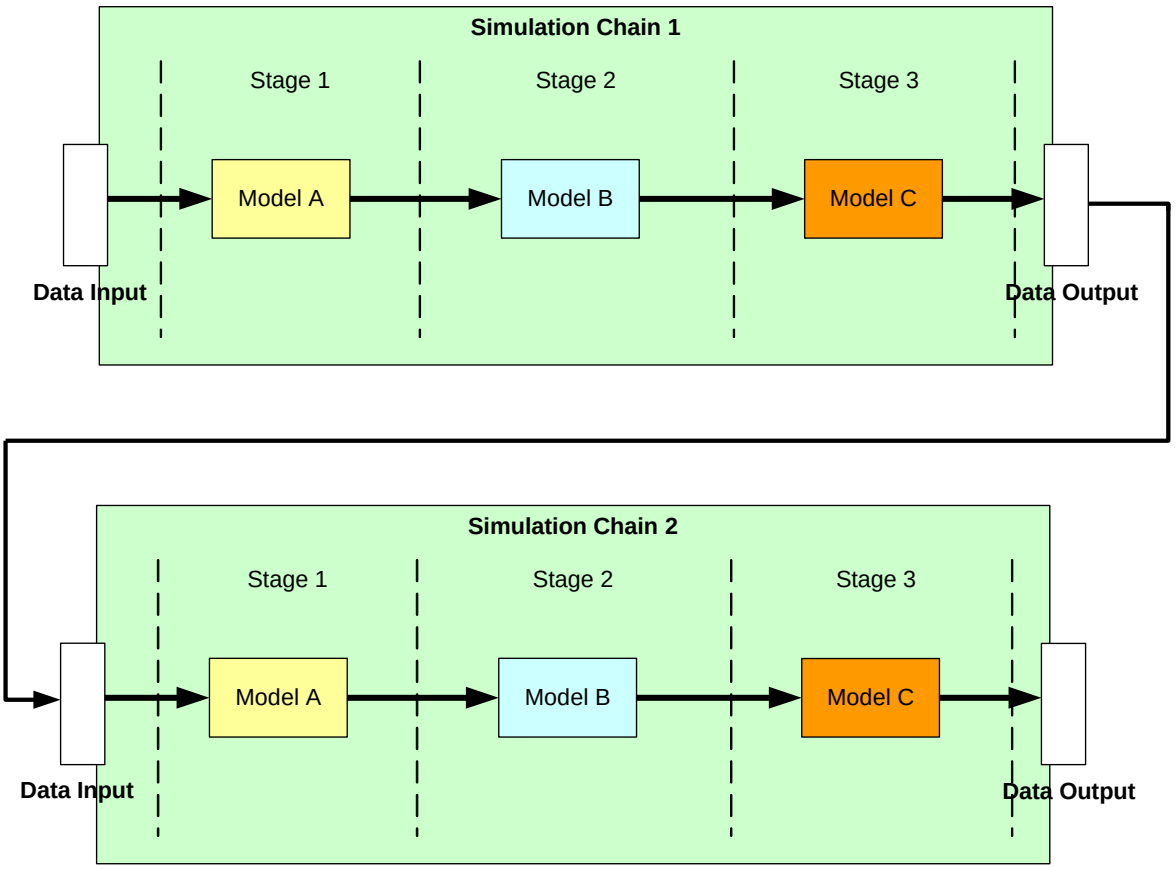

Figure 4-4: Session describing a sequence of simulations

- A simulation in batch mode is a session execution with a sequence of simulations iterated with different configurations (parameter values) for their models.

4.3.3. Product tool

Due to the necessity to cover the requirements related to plotting, viewing and editing product files, the concept **product tool** has been introduced.

A product tool is an external program, an executable that can be called by the openSF system to perform some operations upon product files. A good example of this is an XML editor tool. Users can associate the action "edit" for XML configuration files to a program like GEdit or such, and then enjoy the capabilities of a well-known and powerful third-party solution to common operations outside the scope of the openSF system.

Another plausible application is the plotting and post-processing functionalities.

4.3.4. Multiple Simulation Repository

A processing or simulation chain is a set of processing steps where each step represents a decisive processing stage in the chain. An example of a processing chain with 4 stages is depicted hereafter.

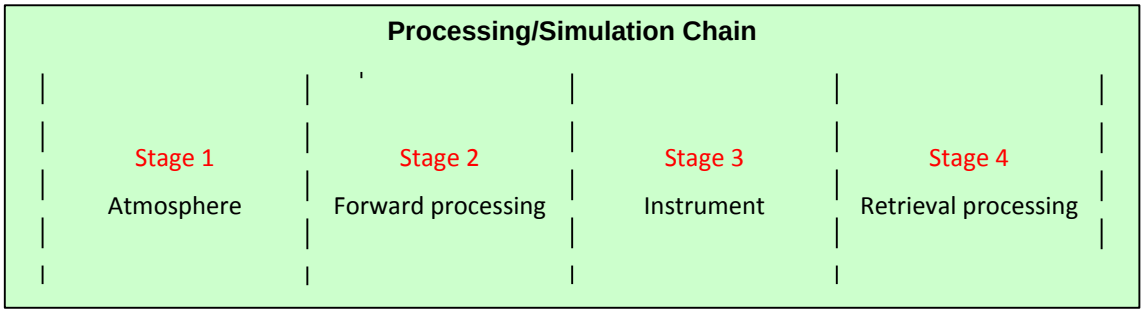

Figure 4-5: Definition of a processing chain

A simulation is a subset of stages from the simulation chain, composed by one or more of the processing stages from the simulation chain.

Upon building a simulation, models are associated to all stages defined for that simulation. Thus, the simulation shown in the example below is constituted by three processing stages. In this case, each one is covered by one model.

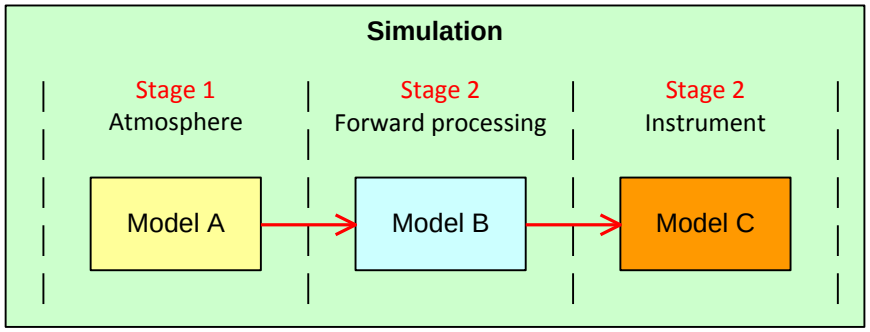

Figure 4-6: Definition of a simulation

For openSF version 2.0 the capability to define more than one processing chain within the same openSF instance has been added. Thus it would be possible to hold simulations for more than one mission, or to define variations of the processing chain for one mission.

This change implies the development of a simple database manager that handles the different simulation scenarios for every mission. This database manager will create different tables for every mission allowing users to switch between missions at the openSF startup or when it is demanded.

4.4. Deployment View

This view emphasizes the physical decomposition of the system on the target platform using nodes, artefacts, components, and relationships.

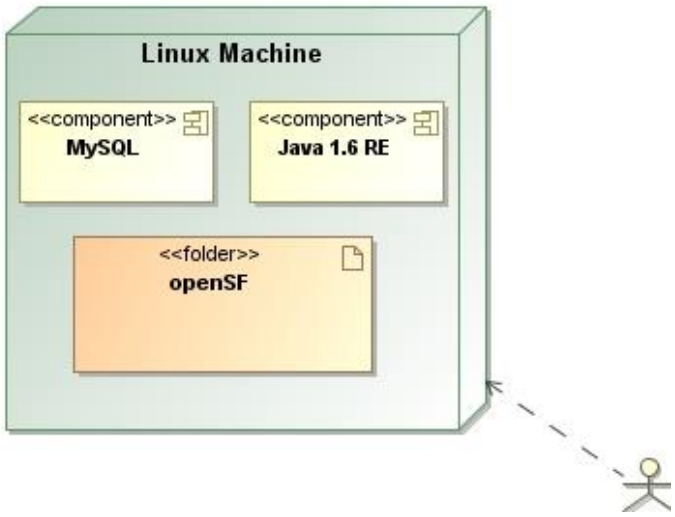

Figure 4-7 OpenSF deployment diagram

This is a diagram showing the different alternatives for deploying openSF. A deployment diagram serves to model the hardware used in system implementations, the components deployed on the hardware, and the associations between those components.

The default deployment configuration, used for acceptation tests, is based on Linux machines. It will consist of a binary installer that will guide the user in the framework deployment. As result a folder with all necessary sources, executable and library binaries will be created in the target machine. In openSF there are third party applications (pre-requisites) that are not included in the distribution and it remains the user's responsibility to download them in purpose (following the instructions of the Software Manual).

Documentation is also distributed.

Composition

- MySQL - MySQL is a relational database management system. The program runs as a server providing multi-user access to a number of databases.
- Java 1.6 JRE – Java 1.6 runtime environment implementing the virtual machine used by the openSF application.

5. SYSTEM DESIGN

This section gives a brief description of the method used for the architectural design, the Unified Modeling Language used for formal diagrams and the system background and context.

5.1. Design Method

The application is distributed in packages used to organize the namespace for packages, classes and interfaces.

In the design process of the system the following conventional guideline to name Java components has been used throughout the whole document:

- **Classes:** class names should be nouns, with the first letter of each class capitalized, such as *Command* and *TreeTable* classes.
- **Packages**: package names should be also nouns, with the first letter in lowercase, and the first letter of each internal word in capitalized, such as *application* or *treeTable* packages.
- **Interfaces**: interfaces names should be capitalized like class names and must end with the suffix "IF", such as *PresentationIF* and *DatabaseIF*.

5.2. System Decomposition

OpenSF is decomposed in four high level packages called *mmi, domain, database and application*. The first three are a direct consequence of the 3-tier architecture approach mentioned earlier and the last one is created for initialization purposes and to give some useful services to every package in the system. In the following diagram the high-level hierarchical structure of these packages is shown:

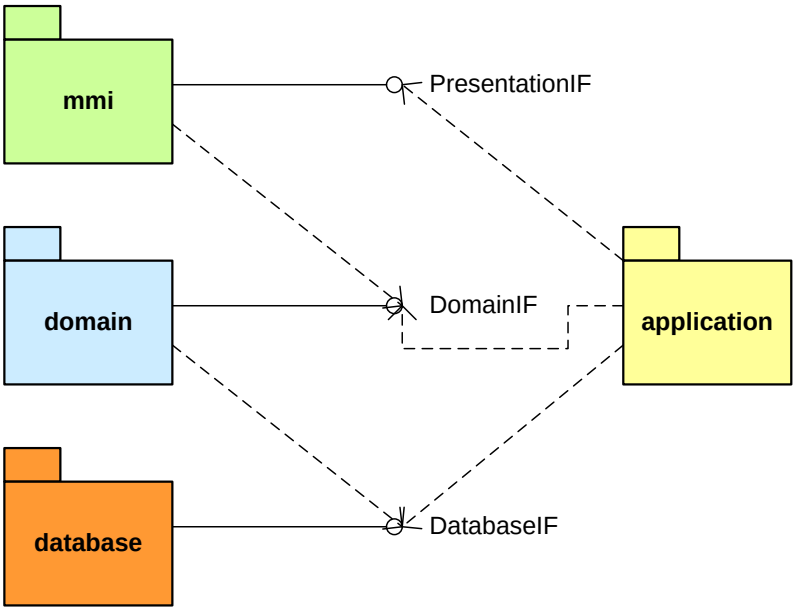

Figure 5-1: High-level package diagram

- **mmi**: Contains packages and classes related to the appearance and behaviour of all visible components of the graphic user interface (windows, frames, visual components and other widgets). This package shares and implements a public interface that could be accessed by the *application* and *controller* layers. It has a strong binding with the *controller* package.
- **domain**: This is the core of the system. It is responsible for all activities related with the specific domain purpose of openSF. This involves the capabilities to carry out the definition and management of the necessary elements identified in the domain: models, simulations, sessions, logs, etc, as well as the execution of the sessions and the visualisation of the results produced.
- **database**: Contains classes designated to control the connection to an external *MySQL* database server and to perform queries and updates against it. This package is being used by the *domain* package and is initialized by the *application* package. This package represents the lower tier in the three-tier paradigm and does not use any other package throughout the system.
- **application**: Contains all classes related to the system initialization and execution, accessing to external resources and some utilities to be used under the whole system scope. The *application* package shall be accessible to the remaining of classes and packages. It makes use of every other package as long as it creates the principal components of the system.

The *mmi* package conforms to the *presentation* layer described in section 3.2.4. Inside this package there is an adaptation of the Model-View-Controller paradigm that implements the JAVA SWING package.

The *domain* and the *database* packages correspond to the *domain* and *database* layers respectively, and are coloured in blue and orange.

The base namespace of the system and of these packages is *openSF*.

5.2.1. application

Classes related to the system initialization and common functionalities are grouped in this package.

This package contains the main class of the system, *openSF*, in charge of creating the database, domain, presentation and controller modules consecutively in order to begin the interaction with the user. This interaction can be seen in Figure 5-2 below.

Note an important aspect of this class. Since openSF is the main class of the system, the final executable shall be also named openSF. This executable is designed in such a way that it can admit a number of parameters that allow users to configure the particular execution of the application. One of the input parameters foreseen regards the batch execution, which shall be explained further ahead. Other input parameters are relevant for the interaction with openSF database (user, password, database name and network address).

The *Resources* class implements access to external resources need by different modules throughout the system, such as images, icons, field titles, tooltip texts, etc. This functionality is related with the system configuration (see section 4.9.1.2 [AD-ICD]).

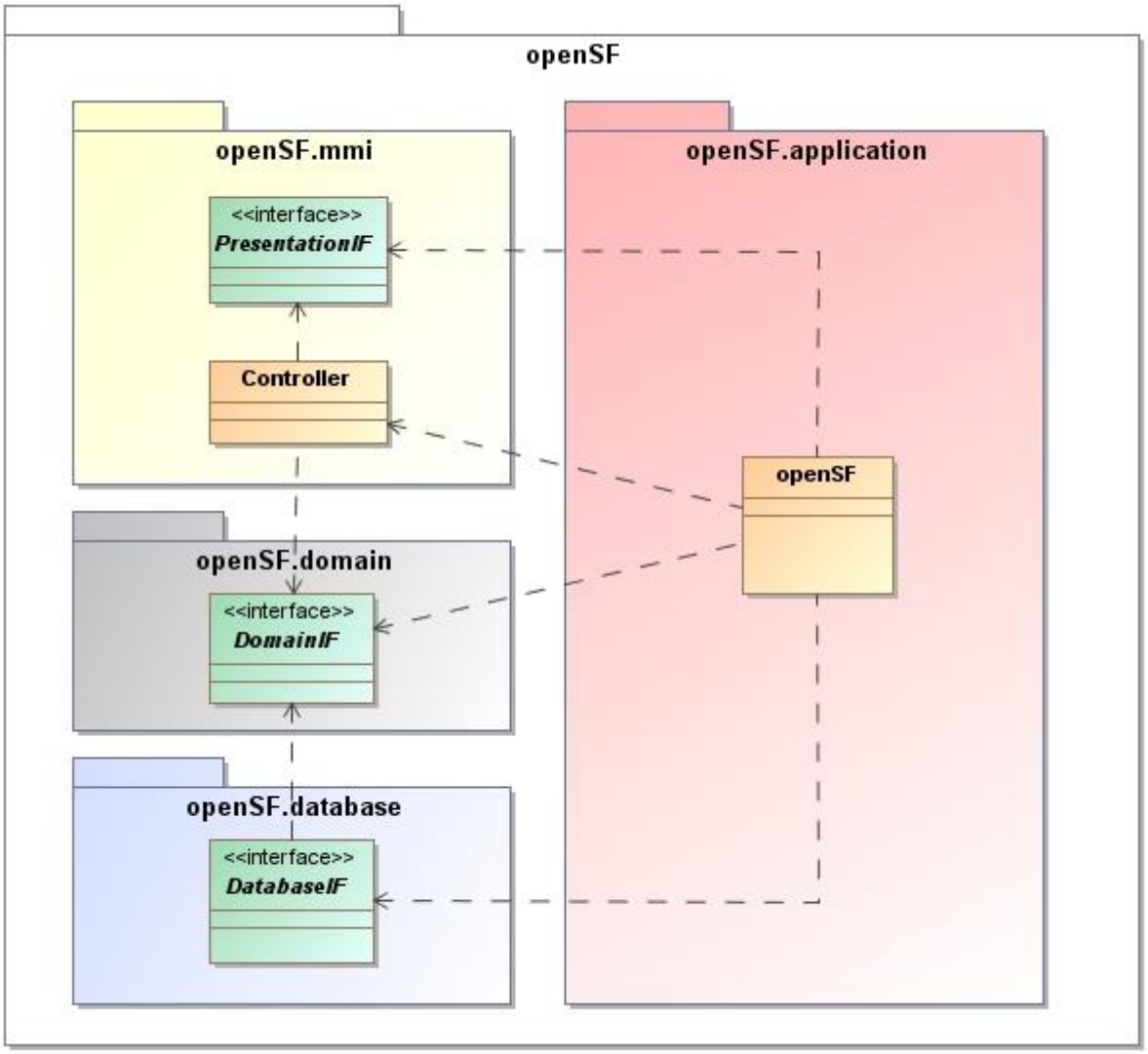

Figure 5-2: openSF.application class diagram

5.2.2. mmi

The *mmi* package is just a global package that is further decomposed in two packages: *presentation* and *controller* to adjust in a clearer way to the three-tier paradigm.

Classes inside the scope of this package use others from the *domain* package and some utility classes from the *application* package. In turn, the *application* package depends on this one because it initializes the principal classes of it.

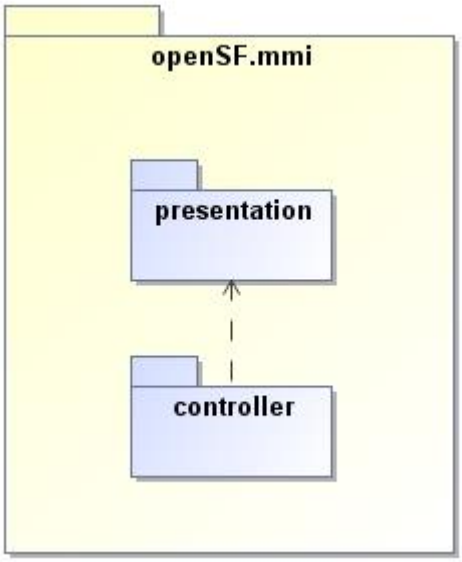

Figure 5-3: openSF.mmi package diagram

5.2.2.1. presentation

This package groups the classes related to the appearance of the user machine interaction.

This *presentation* package is, in turn, composed of four packages:

- ***scrollableDesktop*:** This package contains classes needed to implement a scrollable desktop, a visual container for internal frames. This package does not use any package of the openSF system. This package is used by classes in the *presentation* and *views* packages.

  The complete namespace for this package is: *openSF.mmi.presentation.scrollabledesktop.*

- ***treeTable*:** This package contains all classes used to describe an ad-hoc visual component, the *TreeTable* and its model and listeners. This package does not make use of any package of the system. This package is intensively used by classes inside the *views* package.

  The complete namespace for this package is: *openSF.mmi.presentation.treetable.*

- ***views*:** This package contains all classes required to show internal frames (or views) inside the working area of the GUI, a view factory and other view classes representing the appearance of the openSF modules. This package uses the *treeTable* and the *scrollableDesktop* packages. This class implements the prototype design pattern as it shows a view of a cloned domain object.

  The complete namespace for this package is: *openSF.mmi.presentation.views.*

- ***images*:** It contains all image files needed for the graphical user interface. This does not include any executable component. This package is only accessed by the *Resources* class from the *application* package.

  The complete namespace for this package is: *openSF.mmi.presentation.images.*

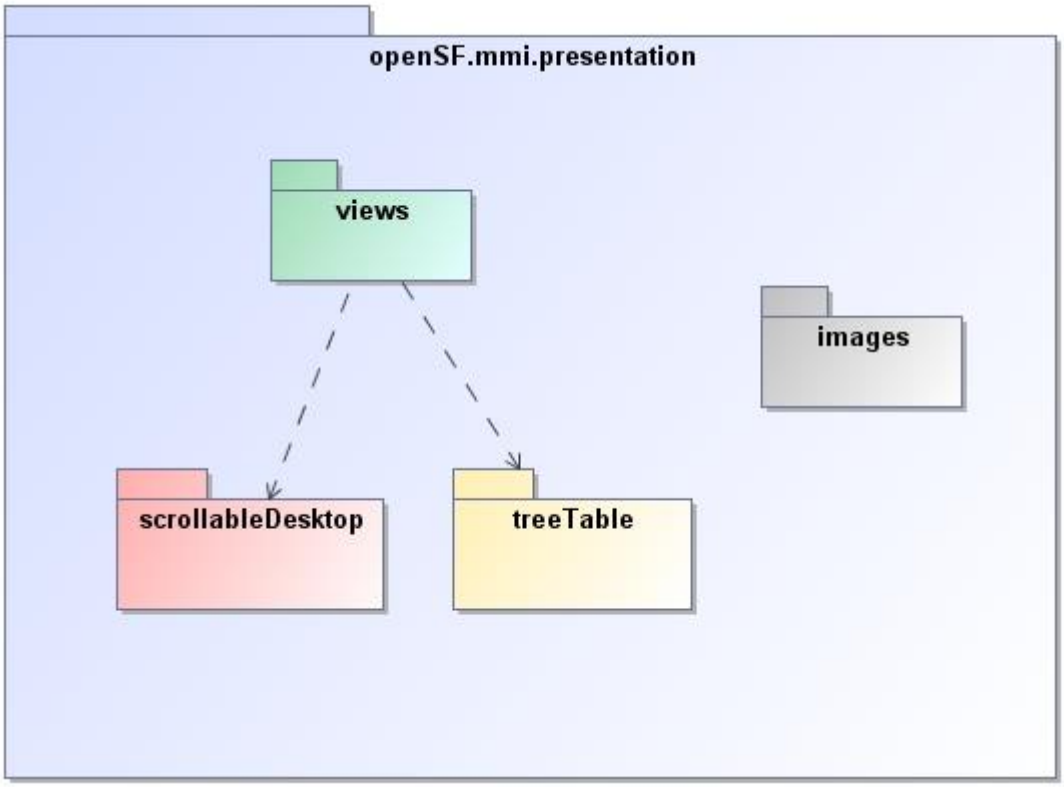

Figure 5-4: openSF.mmi.presentation package diagram

Every package and classes inside the *presentation* package uses intensively the *java.awt* and *javax.swing* packages, whose descriptions are outside the scope of this document but can be found in [RD SWING].

This *presentation* package is accessed by the *application* package and the *controller* package.

This package depends on the functionality inside the *application* package and does not make use of any other part of the openSF system.

Figure 5-5 shows the class diagram of the *presentation* package, a static view of the package where almost the major classes and their relationships can be seen.

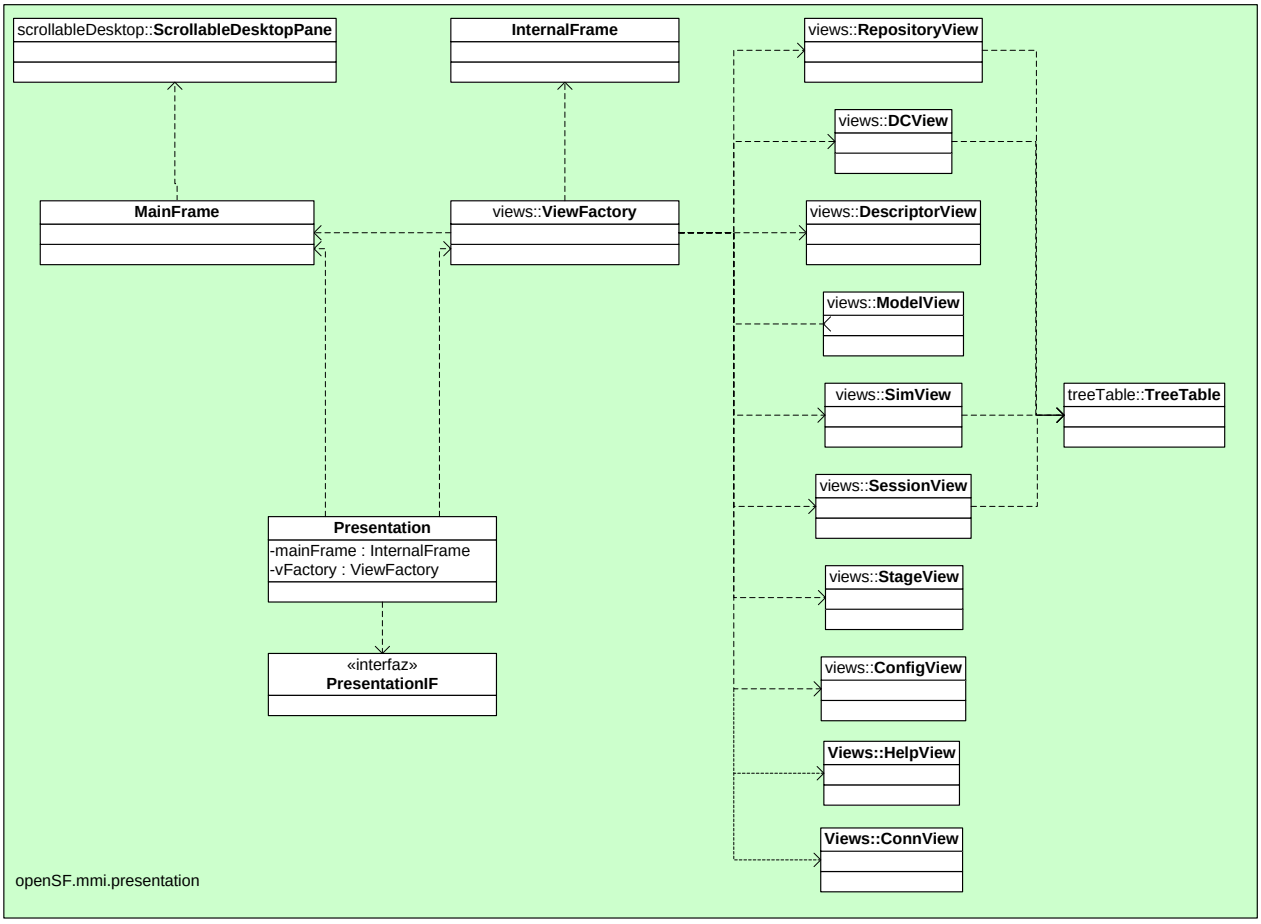

Figure 5-5: openSF.mmi.presentation package class diagram

Inside the *presentation* package lie the following classes:

- ***Presentation***: this class implements the *PresentationIF* interface and creates an instance of *MainFrame* and *ViewFactory* classes.
- ***MainFrame***: this *MainFrame* class contains all the visual components of the main window, that is, the menu bar, the working area, the views and such. It acts as final parent of all the visual components of the GUI. Closing this frame will exit the application.
- ***InternalFrame***: this class implements a wrapper frame to contain all the views produced by the *ViewFactory* class. These frames are shown internally (with an instance of the *ScrollableDesktop Pane* class as parent) in the working area of the *MainFrame*
- ***ModalFrame:*** this class represents a dialog frame that locks the user interaction with other parts of the GUI until obtaining its feedback.
- ***Table,LogTable,IOTreeTable and ParamTreeTable:*** those classes implements the different special views for some of the domain elements (parameters, files and log messages)

5.2.2.1.1. images

This *images* package contains all image files needed for the graphical user interface. This does not include any executable or source component.

5.2.2.2. controller

The *controller* package contains classes and interfaces dedicated to describe the behaviour of the graphical user interface, to provide an access from the presentation layer to the domain layer and therefore, to present the data that must be displayed.

This *controller* package is composed by another three packages:

- ***commands*:** contains the visual representation of the way to access to different functionalities or actions of the system and all its modules. These representations, or commands, can be contained in menus, pop-up menus, buttons and other visual components.

  The complete namespace for this package is: *openSF.mmi.controller.commands*

- ***swingModels*:** This package contains a *swingModelFactory* class that creates different models needed by components of the *presentation* package to present and interact with data. These models describe which kind of data components are going to use. This package is accessed by the *controller* and the *domainConnector* packages. This package makes use of the *domain* package but is not shown in the diagram for the sake of clarity.

- ***domainConnector*:** This package contains classes implementing the presentation of each module of the system. They call *presentation* interfaces to create and show the proper appearance for each action they can make, access the *commands* package and assign a certain *Command* to every action, create a menu with some *Commands* to ease the access, access the *domain* package to get data needed and access the *swingModels* package to create the models to produce the interaction between data and presentation. This package is accessed only by the *controller* package.

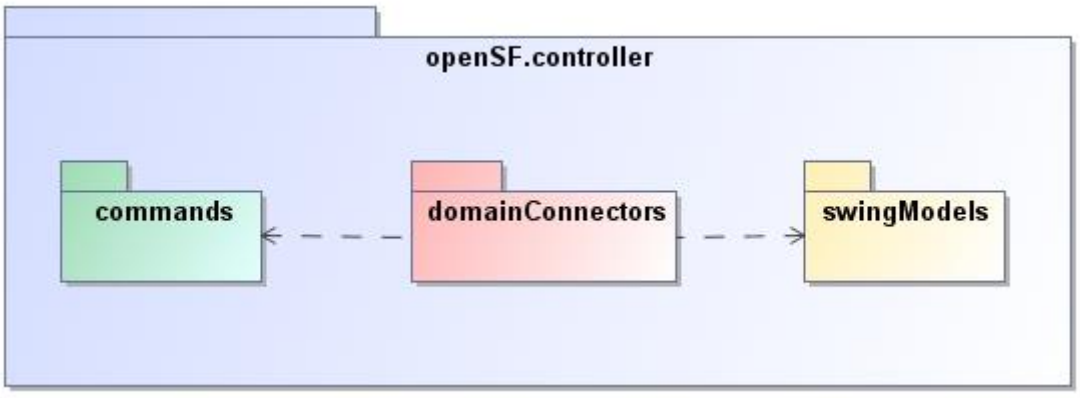

Figure 5-6: openSF.mmi.controller package diagram

This *controller* package makes use of the *PresentationIF* interface from the *presentation* package and the *DatabaseIF* interface from the *domain* package.

This package is initialised and, therefore, accessed by the *application* package. This package is also accessed by the *presentation* package.

The complete namespace for this package is: *openSF.controller*.

5.2.2.2.1. domainConnectors

This package contains classes implementing the presentation of each module of the system. *DomainConnectors* intended:

- To prepare the appearance of every action performed by users, as presenting forms, tables, trees, etc and also menus and buttons to let the user triggers the operations.
- To access the *domain* package to get data needed. This represents the connection to the domain layer presenting or providing the information managed by this part of the application.

The complete namespace for this package is: *openSF.controller.centers.*

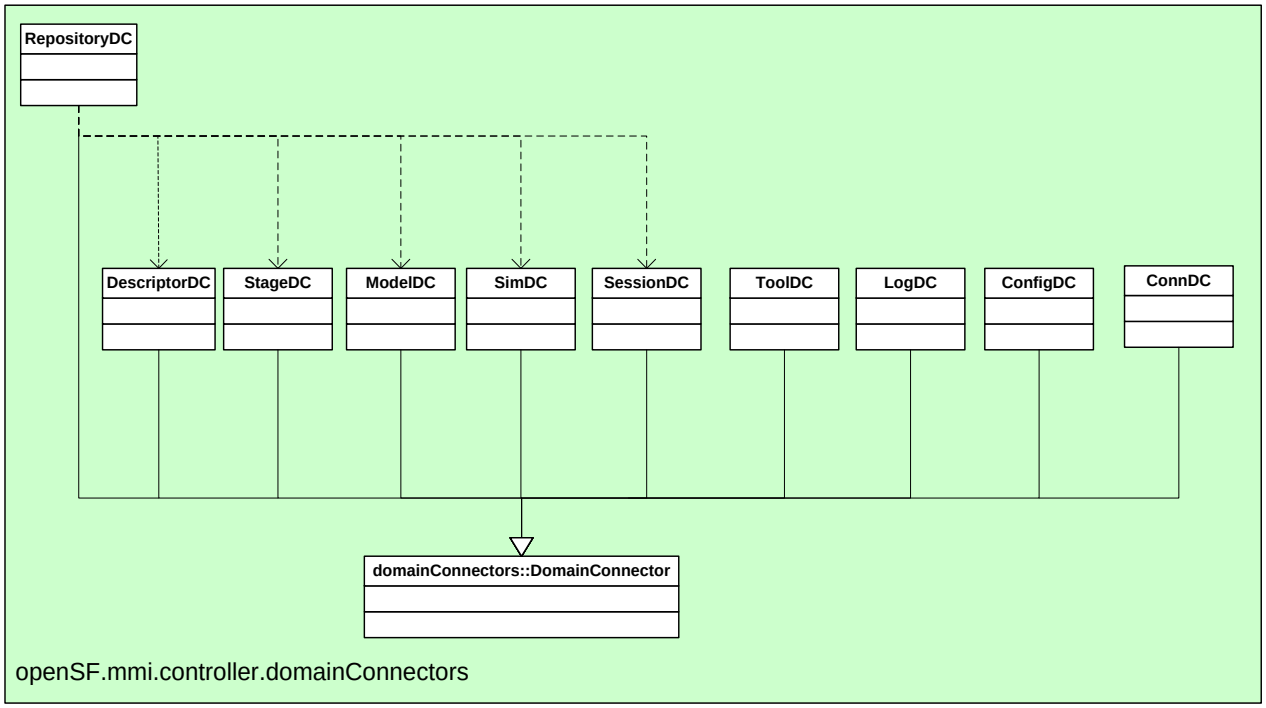

Figure 5-7: openSF.controller.domainConnectors package class diagram

This package contains the following list of classes:

- ***DomainConnector*:** base class of all the others classes in this package. It provides basic functionality available to all classes derived from it.
- ***ConfigDC:*** This class contains the presentation controller for the configuration of openSF. This configuration view can be accessed from the application main menu. From this view users are able to edit some relevant system variables such as environment variables, system folders etc…
- ***DescriptorDC:*** This class contains the presentation aspects for the input output descriptors within openSF system.
- ***RepositoryDC*:** This class contains the presentation issues regarding the complete data repository. It presents a view in the *MainFrame* with two tabs: a system objects view and a file system view. The system objects view contains a *TreeTable* with a hierarchical structure of the data know by the system: models, simulations and executions (as seen in Figure 5-7). The file system view is shown in a tree structure representing files and directories within the application's path. This repository provides quick access to functionalities from the model, simulation and execution modules like showing list of models or creating, editing and deleting simulations or others.

- ***ModelDC***: This class contains the presentation aspects in what refers to models. Similarly to the DescriptorDC, ModelDC presents views for the functionality for listing, creating/editing, deleting models.
- ***ResultDC***: This class contains the presentation aspects related to the session results. It presents a unique view of a executed session showing input, configuration and output files, execution time and whether it failed or not.
- ***SimDC***: This class contains the presentation aspects regarding the simulations available in openSF. It therefore presents views for listing, creating/editing, and deleting simulations.
- ***SessionDC***: This class contains the presentation aspects related to the session in openSF. It presents views for listing, creating/editing, and deleting sessions.
- ***StageDC***: This class contains the presentation aspects related to a stage within openSF system. Users are able to access to a list of the previously defined stages, create a new one or delete other one.
- ***LogDC***: This class contains the presentation aspects relative to the log sessions available in the system. Users can access to the list of log sessions from previously executed sessions, and through the main window to the whole set of previously executed session logs.
- ***ToolDC***: This class contains the presentation aspects relative to the product tools. Users are able to access to a list of the previously defined tools, create a new one or delete other one.
- ***ConnDC***: This class contains the presentation aspects regarding the different simulation repositories that can be plugged in the system.

The *DescriptorDC*, *ModelDC*, *SimDC*, *SessionDC*, *StageDC*, *LogDC* and *ToolDC* components shall create menus and pop-up menus, and define the SWING models (visual components like tables or trees) that will be employed to present to the user the information from their corresponding elements from the *domain* layer.

5.2.3. domain

This package corresponds with the domain layer of the three-tier paradigm.

This package represents the core of the system. All packages and classes related to the simulation process are grouped here. There are two packages inside it as shown in Figure 5-8.

- ***managers*:** this package contains the manager classes, which are meant to manage and control the sets of domain objects (elements). Thus, *managers* shall be composed of classes in charge of managing every module in the system: model, simulations, sessions, etc. The difference with the "*domainConnectors*" classes in the *controller* package is that those give a graphical interface for the user to access these actual operations.

  This package will depend on *elements* and *database* packages. Some classes inside the "*managers"* package will be instantiated by the *domain* package.

- ***elements*:** this package contains classes for the representation of the single elements and objects, that is, temporary data within the scope domain of the system. Classes for describing models, simulations, sessions, tools, descriptors, stages, logs and other auxiliary elements can be found in this package.

  This package is needed by the "*managers"* package.

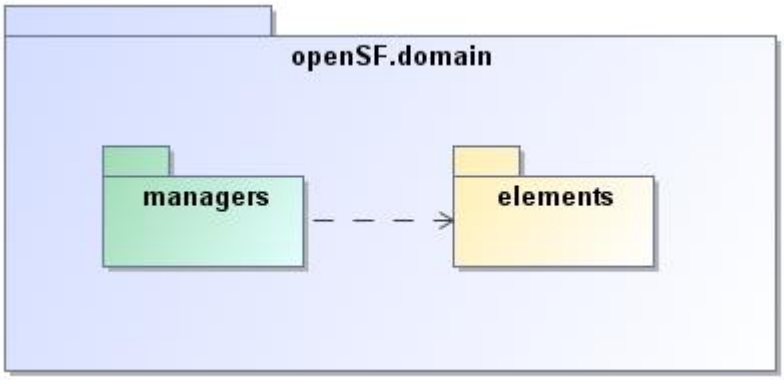

Figure 5-8: openSF.domain package diagram

This class declares the *DomainIF* interface which permits the external packages to access the domain data and its operations.

This *domain* package implements the *domain* class, responsible to implement the *DomainIF* interface. It initializes all the "*managers"* classes to interact with actual data.

This package is accessed by the *controller* and *application* packages and accesses the "*managers"* package.

The complete namespace for this package is: *openSF.domain.*

In the following figure it is detailed more precisely the dependencies between classes in the *domain* and included packages.

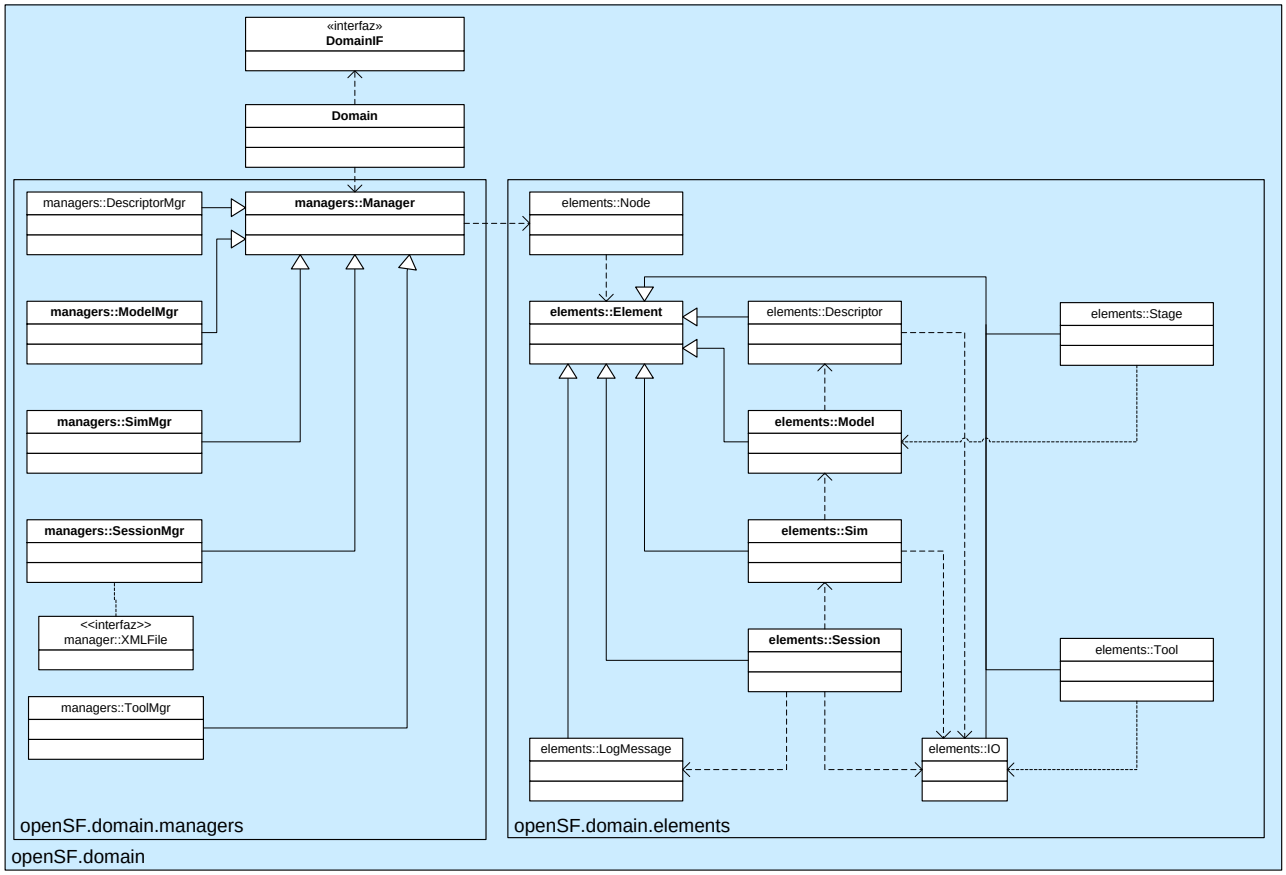

Figure 5-9: openSF.domain class diagram

5.2.3.1. managers

This package contains the manager classes. These classes are meant to manage, and control the sets of domain objects (elements). Thus, they shall be in charge of managing the different kinds of elements in the system: model, simulations, sessions, tools etc.

It may seem that classes of the *managers* package have the same purpose with the *domainConnectors* classes described earlier. The difference lies on the fact that the *domainConnectors* classes provide the graphical interface elements to users for the access of the operations contained *managers*.

The class diagram of this package is shown below and the classes it contains a described in the next paragraphs.

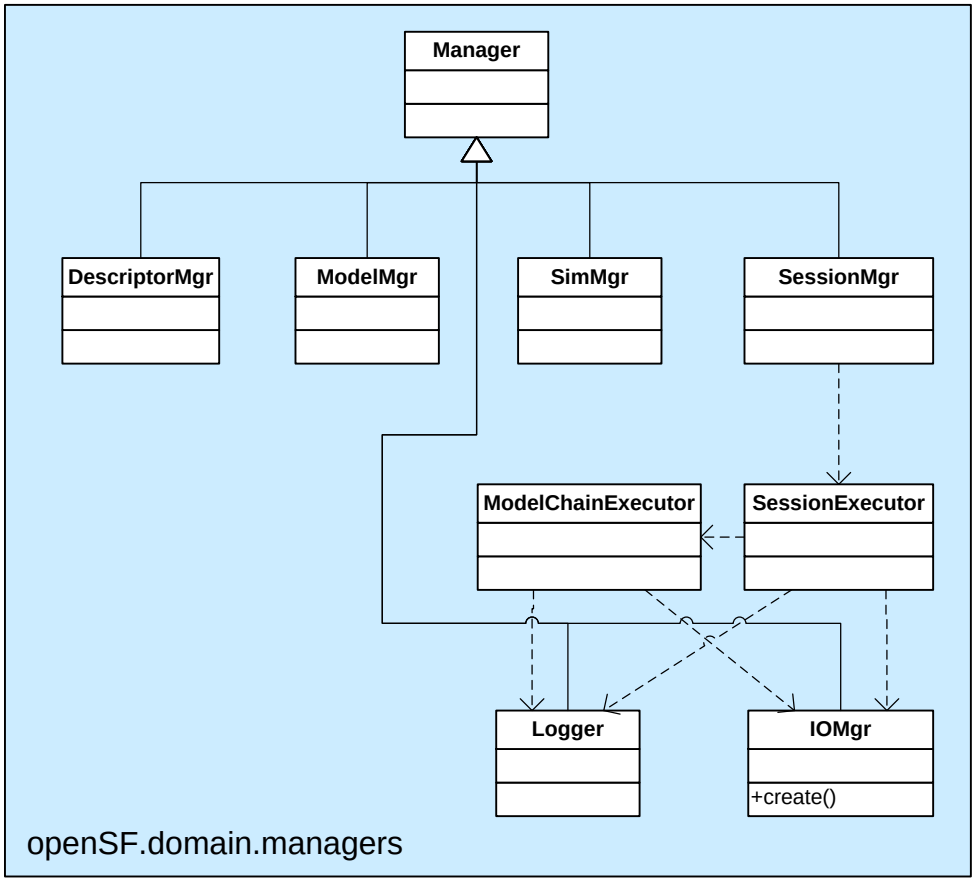

Figure 5-10: openSF.domain.managers class diagram

- ***Manager*:** Base class for all other classes. It provides a set of properties and operations common to all classes derived from it, such as the database access, temporary data storing within a tree structure and so on.

- ***ModelMgr***: class responsible to implement the operations related to model definitions:

Table 4: List of operations of the ModelMgr class

Operation name	Input	Output	Description
`addModel`	model	void	This method creates a model in the database with the information contained in the *Model* instance passed as parameter.
`modifyModel`	model	void	This method modifies an existing model from the information contained in the *Model* instance passed as parameter. The only fields subject to be modified are description and the author fields.
`deleteModel`	modelId	void	This method deletes the model identified by *modelId* from the openSF database.

Operation name	Input	Output	Description
getModelList	void	node	The method returns the list of models currently available from the openSF database.
getDescriptor	oldModels, model	boolean	This method returns the descriptor compatibility between the output generated by the previously executed nodes with the current model.
getStageModel	model	stage	This method gets the stage of the current model.

- ***SimMgr***: class in charge of implementing the operations related to the simulations management, as described below:

Table 5: List of operations of the SimMgr class

Operation name	Input	Output	Description
addSimulation	sim	void	This method creates a simulation in the database with the information contained in the *Simulation* instance passed as parameter.
modifySimulation	sim	void	This method modifies an existing model from the information contained in the Simulation instance passed as parameter. In the simulation modification, openSF shall accept to change the following fields: ❑ Description ❑ Author ❑ List of models.
deleteSimulation	simId	void	This method deletes the simulation identified by *simlId* from the openSF database.
getSimList	void	node	The method returns the list of simulations currently available from the openSF database.

- ***SessionMgr***: This class is responsible for the operations regarding the session management and execution. The operations it implements are listed hereafter.

Table 6: List of operations of the SessionMgr class

Operation name	Input	Output	Description
addSession	session	void	This method creates a session in the database with the information contained in the *Session* instance passed as parameter.

Operation name	Input	Output	Description
modifySession	session	void	This method modifies an existing session model from the information contained in the Session instance passed as parameter. In the session modification, openSF shall accept to change the following fields: ❑ Description ❑ Author ❑ List of sessions.
getSession	sessionId	Session	This method returns a *Session* instance containing the information of the session identified by *sessionId*.
deleteSession	sessionId	void	This method deletes the session identified by *simlId* from the openSF database.
getSessionList	void	node	The method returns the list of sessions currently available from the openSF database.
runSession	sessionId	void	This method initiates the execution of the session identified by *simId*.
generateBatchScript	sessionId	void	This method generates the batch script corresponding to the session identified by *sessionId*. The execution of this script causes the same effect as using the runSession method from the MMI.
getLogSession	logId	node	This method returns the list of log messages pertaining to a previously run session identified by *logId*.
downSimSession	simId	status	This method updates the list of simulations getting backward the input simulation in the execution list. Returns the status of the performed operation.
upSimSession	simId	status	This method updates the list of simulations getting forward the input simulation in the execution list. Returns the status of the performed operation.
addBreakpoints	sessionId, breakList	void	This method adds to the input session the list of breakpoints. The list of breakpoints consists on an integer list specifying the step where the execution shall be stopped.
getBreakpoints	sessionId	ArrayList <Integer>	This method returns the steps where execution shall stop.

One of the major operations from this class is the one in charge of the execution of the session (this is, the implementation of the `runSession` method), which may contain one or more simulations. Section 5.3 contains a description of how the system is organised to successfully execute sessions.

5.2.3.2. elements

This package contains classes for the representation of the single elements and objects, that is, temporary data within the scope domain of the system. Classes for describing models, simulations, sessions, products, logs and other auxiliary elements can be found in this package, as shown in Figure 5-11.

- ***Stage***: this class is the representation of a single stage, including its general properties and the position within simulation chain.
- ***Stages***: this class is the representation of a stages set. This set shall match with the stages covered by a particular simulation.
- ***Model***: represents a single model or algorithm definition with its general properties and input, configuration and output files.
- ***Simulation***: implementation of a single simulation definition, that is, an ordered sequence of models with its given input, configuration and output files.
- ***Session***: description of a session definition that is constructed as an ordered sequence of simulation definitions. Therefore, this session will also contains necessary input and configuration files, will extract the parameters susceptible of being used in batch mode, will store a log session with information about incidents and other events happened during the execution of the session and will store the location of generated output files.
- ***LogMessage***: this package is meant to contain a representation of each message generated by the execution of simulations and those generated by the process of a model.
- ***IO***: this class is implemented for the representation of all the file products: input, outputs, and auxiliary and configuration files. Each subtype of file product will have an editor associated, a program that can performs operations to it (as plotting, editing of extract quantities). This package is needed by the model, simulation and execution packages. It does not need any other package of the system.
- ***Element***: This class provides a base class to all classes that represent a domain item. Basically is a list of Attributes that describes a single instance of an item.
- ***Node***: Internally, elements are stored following a hierarchical structure, a tree in which each node stores a list of other Node instances to increase the level of abstraction in the structure. Each Node wraps an *element* instance to store its data.

This package is needed by the "*managers"* package and does not use another package of the openSF system.

The namespace for this package is: *openSF.domain.elements*.

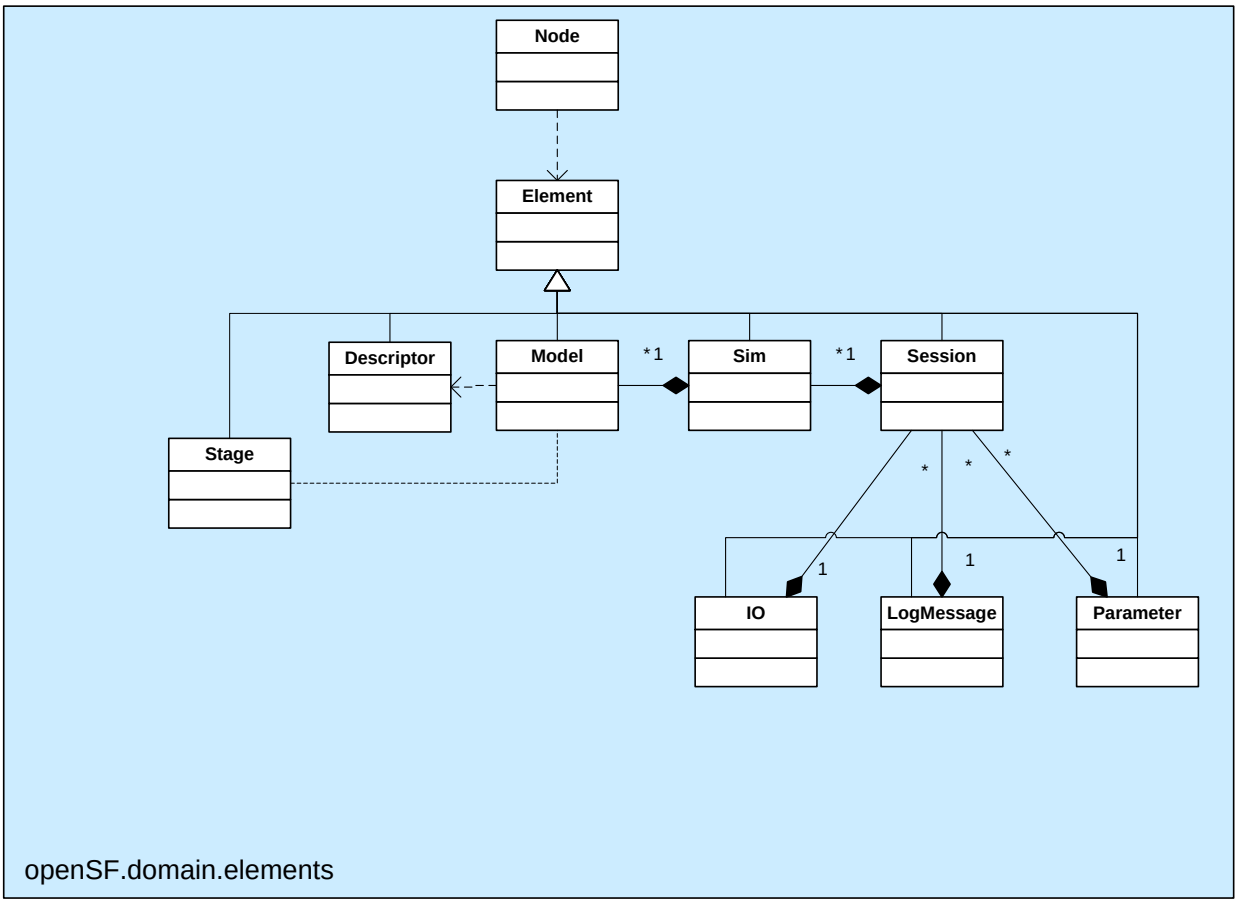

Figure 5-11: openSF.domain.elements Class Diagram

5.2.4. database

This package contains classes designated to control the connection to an external database server and to perform queries and updates against it. It implements the database layer in the three-tier paradigm.

The *database* package contains some classes and interfaces represented in the figure below:

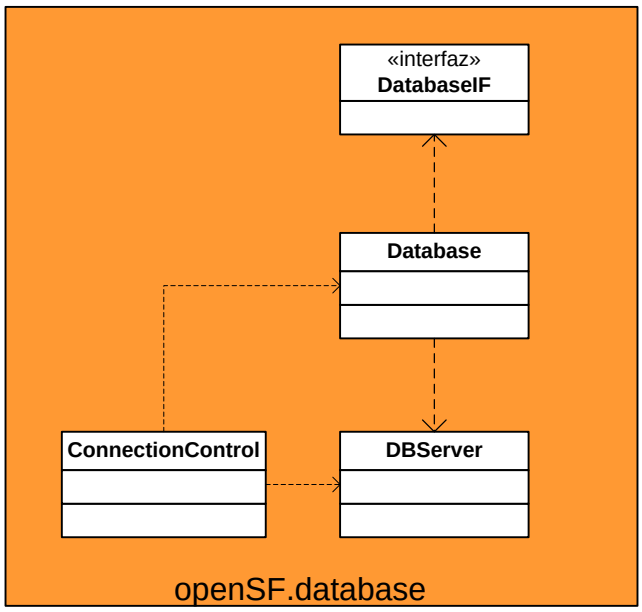

Figure 5-12: openSF.database class diagram

The *DatabaseIF* interface declares methods to share with the domain layer of the system and which calls for the persistence of domain data in a database. This *DatabaseIF* interface is implemented by the *Database* class through a *MySQLServer* instance. This class has the actual implementation of all the SQL statements required to perform the management of all openSF elements (models, simulations, etc) that ensure the correct behaviour of the system.

This *DBServer* class is the responsible to create a connection with the MySQL database server and perform operations of retrieving (queries) and updating (insert, update and delete) through the connection and against the server. This connection is made by the MySQL Connector J 5.0.4, which is a OBDC implementation for Java.

The *ConnectionControl* class is the responsible of managing the multi-repository capability. It stores the connection information for each repository and is in charge of handle the on-the-fly switching between the different mission repositories.

The *ConnectionControl* object has also an interface for *openSF.manager* and *openSF.view* packages in order to be controlled by a view element.

The complete namespace for this package is: *openSF.database*.

Table 7: List of operations of the Database class

Operation name	Input	Output	Description
`addLogMessage`	logMessage	void	This method assembles the SQL sentence needed to add a log message to the database from a *LogMessage* object instance passed as parameter. *DBServer* class instance will execute the sentence and performs the adding operation.

Operation name	Input	Output	Description
createModel	model	void	This method formats the SQL sentence to create a new model in the database from memory data given as parameter. Then the sentence is passed to the *DBServer* that shall execute it.
createSession	session	void	This method formats the SQL sentence to create a new session in the database from memory data given as parameter. Then the sentence is passed to the *DBServer* that shall execute it.
createSim	sim	void	This method formats the SQL sentence to create a new simulation in the database from memory data given as parameter. Then the sentence is passed to the *DBServer* that shall execute it.
createTool	tool	void	This method formats the SQL sentence to create a new tool in the database from memory data given as parameter. Then the sentence is passed to the *DBServer* that shall execute it.
deleteModel	modelId	void	This method makes the SQL sentence to delete a certain model specified by the input parameter. The *DBServer* instance is called later to execute the sentence and actually delete the model.
deleteSim	simId	void	This method makes the SQL sentence to delete a certain simulation specified by the input parameter. The *DBServer* instance is called later to execute the sentence and actually delete the simulation.

Operation name	Input	Output	Description
getLogMessageList	void	ArrayList <LogMessage>	This method describes the SQL sentence needed to recover the complete list of log messages stored in the database. The *DBServer* instance is then called to execute the query and retrieve data. The method returns a structure with the data set.
getModelList	void	ArrayList <Model>	This method contains the SQL sentence needed to recover the complete list of models stored in the database. The *DBServer* instance is then called to execute the query and retrieve data. The method returns a structure with the data set.
getModelsofSim		ArrayList<Models>	This method builds the SQL statement needed to get the Model stored in the database whose simulation is the one passed as input.
getParamsOfSession	session	ArrayList <Parameter>	This method describes the SQL sentence needed to recover the complete list of parameters stored in the database whose session identifier matches with the one passed as input. The *DBServer* instance is then called to execute the query and retrieve data. The method returns a structure with the data set.
getSessionList	void	ArrayList <Session>	This method contains the SQL sentence needed to recover the complete list of sessions stored in the database. The *DBServer* instance is then called to execute the query and retrieve data. The method returns a structure with the data set.

Operation name	Input	Output	Description
getSimList	void	ArrayList <Sim>	This method contains the SQL sentence needed to recover the complete list of simulations stored in the database. The *DBServer* instance is then called to execute the query and retrieve data. The method returns a structure with the data set.
getSimofSession	session	ArrayList<Sim>	This method builds the SQL statement needed to get the Simulations stored in the database whose session is the one passed as input.
getStagesList	void	ArrayList <Stage>	This method builds the SQL statement needed to recover the complete list of stages stored in the database.
getToolsOfSession	session	ArrayList <Tool>	This method builds the SQL statement needed to get the tools stored in the database whose session is the one passed as input.
modifyModel	model	void	This method formats the SQL sentence to alter some model attributes in the database. The model is specified by the id string in the model input parameter. Then the sentence is passed to the *DBServer* that shall execute it.
modifySim	sim	void	This method formats the SQL sentence to alter some simulation attributes in the database. The simulation is specified by the id string in the simulation input parameter. Then the sentence is passed to the *DBServer* that shall execute it.
removeTool	tool	void	This method makes the SQL sentence to delete a certain tool specified by the input parameter. The *DBServer* instance is called later to execute the sentence and actually delete the simulation.

Table 8: List of operations of the ConnectionControl class

Operation name	Input	Output	Description
createNewRep	Repository	void	This method creates a new *Repository*. The *Repository* object stores the connection information for the correspondent database.
deleteRep	Repository	void	This method deletes a *Repository* from the system.
switchConnection	Repository	void	This method switches the current loaded *Repository* to the one passed as argument. This method is in charge of cleaning the current loaded *Database* and create a new one.
setDefaultRep	Repository	void	This method set the selected *Repository* as the default one. This will be the one loaded when openSF starts.
dumpRepository	Repository	String/File	This method performs a dump of the correspondent SQL database into a text file. It is usually used for database maintenance activities.
getSystemReps	void	ArrayList<Repository>	This method retrieves the whole list of simulation repositories within the system.

Figure 5-13 shows the memory representation of the *elements* package, which can be the initial step to the Entity-Relationship diagram, which shall indicate the physical representation of its storage in a MySQL server database.

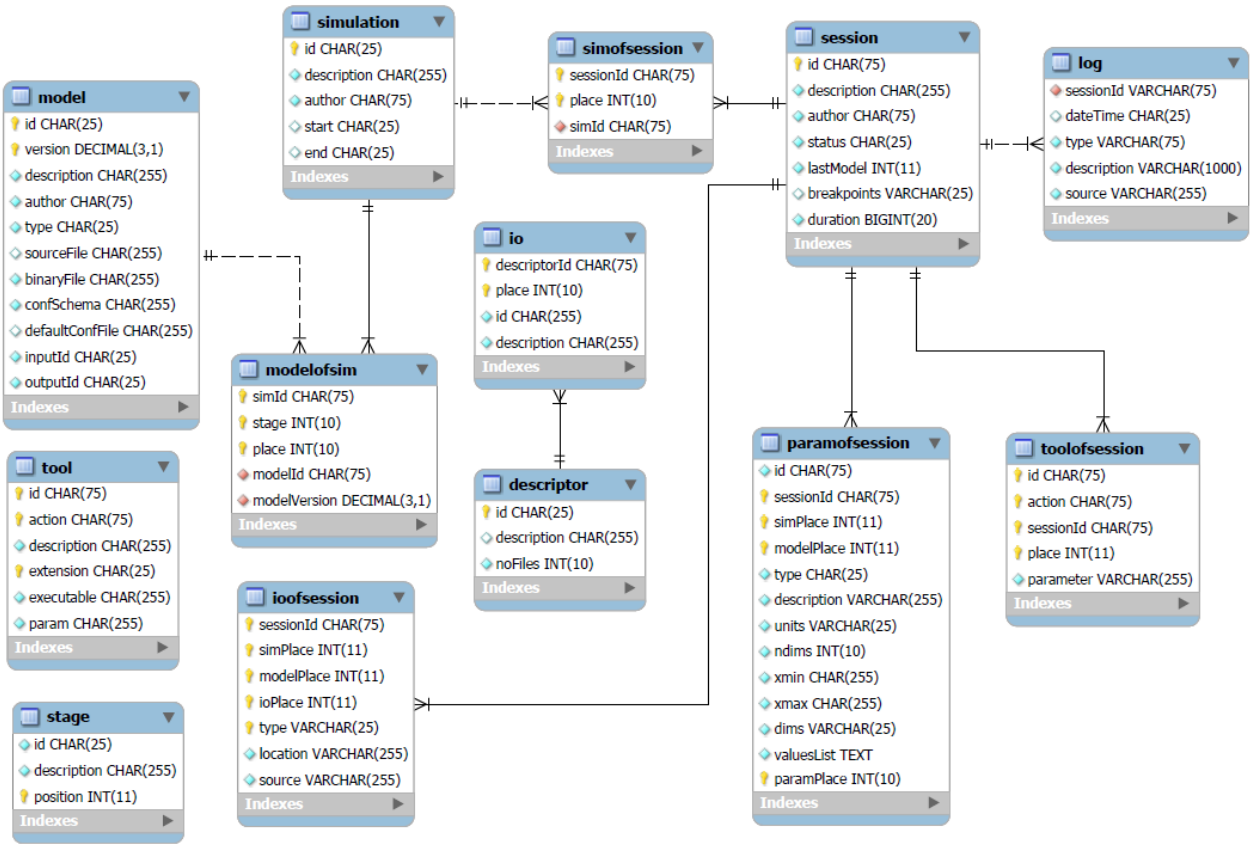

Figure 5-13: Database diagram[1]

5.3. Session Execution Components Description

As mentioned earlier, this section gives more details of the *SessionMgr* class in order to clarify how the execution of sessions is handled by the system.

For this a further decomposition of this class is needed and is provided in Figure 5-14. *SessionMgr* and the classes it is composed of are described in depth in the following sub-sections.

[1] Note: The "log" table shall be dropped from the database schema as a consequence of CCN1 REQ-2 (refer to section 5.6.2 on the Removal of logs from database). Simofsession table will be changed by a modelofsession one, relating directly sessions and models.

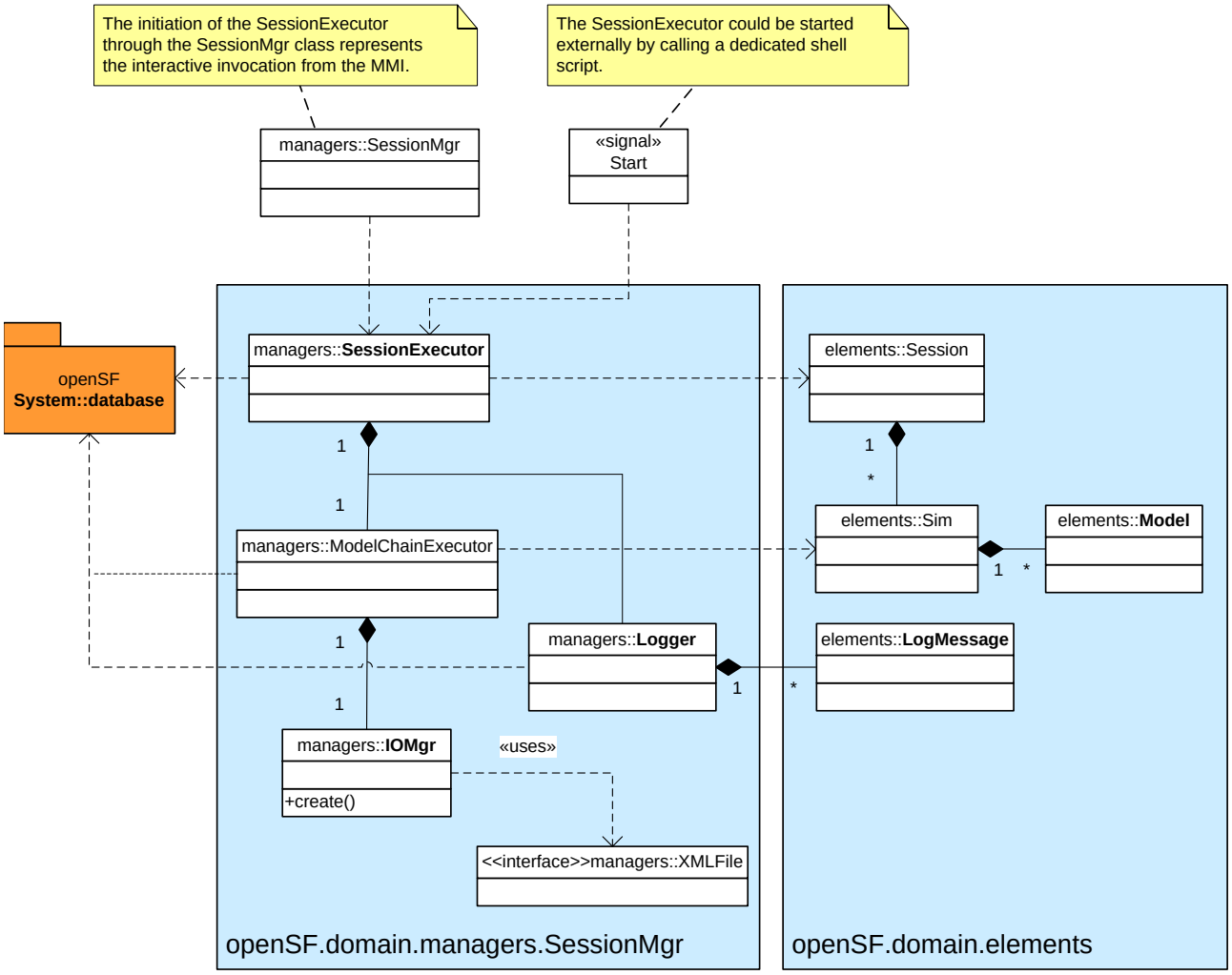

Figure 5-14: SessionMgr class diagram

5.3.1. SessionMgr

5.3.1.1. Type

This component is a class.

5.3.1.2. Purpose

This class is responsible for the operations regarding the management of sessions (addition, modification and deletion) as well as the provision of an execution environment that enables users to launch, monitor and maintain a log of simulation sessions.

5.3.1.3. Function

For the session execution capability, as soon as a session has been invoked to run, either manually or in batch mode, the *SessionMgr* shall create an instance of the *SessionExecutor* class that shall be in charge of launching the execution of the simulations contained in the session. This instance shall be active during the life of the session and therefore shall be responsible to properly finalise the execution in a consistent and correct manner.

5.3.1.4. Super-class

This class inherits from the *Manager* class so it has access to the temporal data structure held in memory and the persistence database over a database server.

5.3.1.5. Dependencies

This class depends on the definition of classes like *Managers*, *Database*, *SessionExecutor*, and *Session*.

5.3.1.6. Interfaces

The public interface of this class is constituted by the operations listed in Table 6.

5.3.2. SessionExecutor

5.3.2.1. Type

This component is a class.

5.3.2.2. Purpose

This class is responsible for the initiating and monitoring the execution of a session, which is composed by one or more models. This capability represents the major challenge of openSF as it must ensure the correct and consistent execution of the different models/algorithms belonging to each specific part of the session.

5.3.2.3. Function

An instance of the *SessionExecutor* class is invoked whenever a session is to be executed. For that the instance shall read from the database all the information relative to the session, whose identifier is passed as an input parameter. This is done via the *Database* instance, which is the connection to openSF's database.

In addition, the *SessionExecutor* instance shall also initiate the corresponding log session (by invoking the appropriate methods of the *Logger* instance) so that all log messages that are generated during the set of simulations are attached to the same session.

Only one session can be run at a single time. This is achieved by using an internal dedicated file that, when exists, shall indicate that there is a session currently being executed avoiding therefore the execution of any other session.

When all conditions to launch the execution are met, the *SessionExecutor* shall create an instance of the *ModelChainExecutor* for each simulation belonging to the session. Apart from providing the set of configuration and the input data files for each simulation, the *SessionExecutor* must provide the logSessionId for the log message storage purposes.

5.3.2.4. Super-class

This class does not inherit from another class.

5.3.2.5. Dependencies

The *SessionExecutor* instance depends on the definitions of these classes: *Database*, *ModelChainExecutor*, and *Logger*.

5.3.2.6. Interfaces

This class provides this list of operations:

Table 9: List of SessionExecutor class operations

Operation name	Input	Output	Description
`start`	sessionId	void	Initiates the execution of the session. For this a new *ModelChainExecutor* instance is created per each simulation belonging to the session, which is responsible for the execution of the particular simulation.
`abort`	void	void	This method will abort the execution of a running session, deleting every output file that could have been generated. This operation is meant to cancel all the operations that the execution has made until the moment.
`executeTools`	void	void	Executes the scheduled product tools into a proper execution environment.

5.3.3. ModelChainExecutor

5.3.3.1. Type

This component is a class.

5.3.3.2. Purpose

The purpose of the *ModelChainExecutor* class is to start and control the execution of a set of models, taking care to feed each participating model with the required input data and configuration files and to collect all log messages generated as part of the simulation (produced by the models or the *ModelChainExecutor* itself) into the database for future queries.

5.3.3.3. Function

This class represents the execution of a model chain by running each of the models it is composed of, providing them with inputs and configuration files and storing and managing the output file products they produce. It also monitors the progress of the execution.

The *ModelChainExecutor* shall be created providing as input parameter the following data:

- The identifier of the simulation to be run
- List of configuration files for the models belonging to the simulation
- List of input data required to start the execution
- The identifier of the log session

The instance is therefore responsible for retrieving all the information of the simulation from the database that needs to be known in order to proceed with its execution. This shall be done through the use of the *Database* instance. This information includes the list of models, and the expected format of their configuration and input files that are passed as input.

The consistency of all input data is checked and if everything is fine, the *ModelChainExecutor* instance shall launch the execution of the first model of the simulation. For this, both the configuration file and the input data files should be available at a specific location. Upon its termination the *ModelChainExecutor* must verify that the input data that must be provided to the next model is available. This is possible if:

- The data files were provided as input by the user, i.e. existed prior to the simulation execution, or
- They were generated by a previous model.

In any case, the models should verify the correctness of the input data file and report an error through the *Logger* class in case of problems.

Once every pre-condition of correctness is checked the actual execution of the model/algorithm is called. This execution is done in a way that allows establishing values for environmental variables and, most importantly, **permits the interception of the standard output messages produced by the models to record them in the log session**. Algorithm developers just need to continue printing their event messages in the standard output file but, if they want them to be logged by the openSF, they need to be written according to the following format described in [AD-ICD]. Then, well-formatted messages will be sent to the *Logger* instance to be stored later in the database.

Note that other events generated by the *ModelChainExecutor* component (this is, not from the algorithms) shall also use the *Logger* instance to store whatever messages in the log session and consequently in the openSF database.

An important advantage provided by the *ModelChainExecutor* is that it overrides the execution of a certain model if it was already executed with the same configuration parameters and inputs in the current session. This is meant to realize an optimization process to avoid unnecessary executions of time-costly algorithms.

5.3.3.4. Super-class

This class does not have a parent super-class.

5.3.3.5. Dependencies

This class depends on some classes from the *elements* package (such as *Sim* and *Model*), *Logger* and *IOMgr* classes from the "*managers*" package, and the physical existence of the executables and associated files needed for their proper execution (configuration and input data files).

5.3.3.6. Interfaces

There is only one operation that can be performed with a *ModelChainExecutor* class instance: *run*. Other operations to control the overall session executions are provided by the *SessionExecutor* instance.

Table 10: List of operations in ModelChainExecutor class interface

Operation name	Input	Output	Description
`stop`	void	void	Stops the current executed simulation.
`start`	void	void	Starts the simulation execution.
`pause`	void	void	Pauses the current simulation
`modelExecution`	model	status (boolean)	Executes a certain model via command line.

5.3.4. Logger

5.3.4.1. Type

This component is a class following the singleton pattern described in section 3.2.1.

5.3.4.2. Purpose

This class aims at recording all events or incidents happening during the execution of a certain session in order to provide the user with some feedback information about the process.

5.3.4.3. Function

This class is responsible for creating a log session and storing all messages produced during the execution of the session. Log session messages must be produced by the *SessionExecutor* when initiating and ending the session, the *ModelChainExecutor* when each model of the simulation is invoked and by the models themselves in order to inform about the actions performed by the specific algorithm.

In [AD-ICD] document, there exists a section describing the formatting style that every message should follow to match with this *Logger* class requirements.

5.3.4.4. Super-class

This class does not inherit from another class.

5.3.4.5. Dependencies

This class has a direct dependency on the *LogMessage* class and also uses information written in the standard output.

5.3.4.6. Interfaces

This class implements the visible interface comprised of the operations in Table 11.

Table 11: List of operations of the Logger class

Operation name	Input	Output	Description
`getInstance`	void	Logger	Accessor method from the singleton design pattern. It returns an instance of the Logger class not allowing more than one copy to be accessed.
`addLogMessage`	logMessage type (Error, Warning, Info)	void	This method will store the incoming message into the database and shall be associated to the log session created through the first invocation of the `getInstance` method.

5.3.5. IOMgr

5.3.5.1. Type

This component is a class.

5.3.5.2. Purpose

This class provides a bridge between the openSF system and the different external file formats used in its context.

5.3.5.3. Function

This class is in charge of translating any specific internal format stored in the openSF system into the files that are needed by the models to run properly and vice versa.

Another important use of this class is to provide a correspondence between file formats and their editors, external programs or product tools. Users can assign to any product file in the system a list of actions, each of them associated with an external tool and its command line parameters that is able to read and manage the referred product file. For example, an XML file shall have an action to edit the file. The Gedit tool (with the appropriate parameters) can be associated to this action meaning that each time the option to edit the XML file is called, the GEdit tool shall be automatically be invoked.

5.3.5.4. Super-class

This class is a descendent of the *Manager* super-class, so inherits its capability to access both temporary and persistent data structures.

5.3.5.5. Dependencies

This class depends on the *Database* as every descendent of the *Manager* class and also has a relation with the operating system over the openSF is running.

5.3.5.6. Interfaces

The public interface declared by this class is as follows:

Table 12: list of IOMgr class public operations

Operation name	Input	Output	Description
`read`	fileName	ArrayList <Element>	Reads a certain file specified by the input parameter and returns a data structure with elements of the system. It makes use of one of the XMLFile or NonXMLFile interfaces that implements.
`write`	filename, ArrayList <Element>	void	Take a list of internal elements and writes in the desired location and the specified format (XML or non XML).

5.3.6. ToolMgr

5.3.6.1. Type

This component is a class.

5.3.6.2. Purpose

This class provides a bridge between the openSF system and the different external used to visualize and post-process the files involved in a simulation chain.

5.3.6.3. Function

The use of this class is to provide a correspondence between file formats and their editors, external programs or product tools. Users can assign to any product file in the system a list of actions, each of them associated with an external tool and its command line parameters that is able to read and manage the referred product file. For example, an XML file shall have an action to edit the file. The Gedit tool (with the appropriate parameters) can be associated to this action meaning that each time the option to edit the XML file is called, the GEdit tool shall be automatically be invoked.

5.3.6.4. Super-class

This class is a descendent of the *Manager* super-class, so inherits its capability to access both temporary and persistent data structures.

5.3.6.5. Dependencies

This class depends on the *Database* as every descendent of the *Manager* class and also has a relation with the operating system over the openSF is running.

5.3.6.6. Interfaces

The public interface declared by this class is as follows:

Table 13: list of ToolMgr class public operations

Operation name	Input	Output	Description
`addTool`	fileExtension, externalProgram, action	void	Associates an external program with some parameters to a given action name applied to every file with the given extension.
`removeTool`	toolID	void	Deletes the association marked as input parameter.
`executeTool`	fileName, toolID	void	Calls for the external execution of a product tool over a certain file.

5.4. Parallel Processing

There are several parallel execution technologies within the software community. Among them, and considering the man cost and implementation complexity, **multicore programming** was selected for the parallel computing in openSF.

A **multi-core processor** is a single computing component with two or more independent actual processors (called "cores"), which are the units that read and execute program instructions. The improvement in performance gained by the use of a multi-core processor depends very much on the software algorithms used and their implementation. In particular, possible gains are limited by the fraction of the software that can be parallelized to run on multiple cores simultaneously (Amdahl's law).

5.4.1. OpenSF Multicore Adaptation

OpenSF target system is typically a computer or server. The popularization (cost reduction) of multicore processors makes it common that almost every target computer nowadays has two or more cores. Furthermore Java provides a set of tools for thread scheduling and synchronization allowing openSF to launch different simulation models in different cores.

The openSF multicore plug-in will consist in the following added entities, carrying out the desired parallelization tasks:

1. *ParallelScheduler*: this will be the core component of the parallel plug-in, orchestrating the execution of the simulation models.
2. *ThreadPool*: managing the available threads
3. *ParallelEventManager*: entity responsible of reporting the scheduler about the model execution status (paused, failed, finished, waiting etc…)

A detailed design description of these entities can be found in the subsequent sub-sections. Other existing openSF components will be affected by this change, leading to appropriate software update. This is the case of the *ModelChainExecutor, SessionExecutor, Logger* components, to name a few.

Regarding the parallel execution the following approaches were analysed:

1. Parallelization at model chain level: each session would acquire a core resource thread and use it to execute all models of the model chain;

2. Parallelization at model level: each model would acquire a core resource thread and use it, then release it when finished.

The selected approach is approach 2. It is foreseen that approach 2 is generic enough to cover parallelization at model chain level as well. This approach also ensures that parallelization is always transparent to the user (i.e. the parallelization is performed without requiring feedback from the user) both at model chain level and model level.

DISCLAIMER: The evolution of openSF to allow parallel execution brings also added responsibility to model developers. It should be clarified that model developers must ensure that the models/algorithms developed are in fact parallelizable, e.g., that the implementation has the proper precautions regarding access to common resources. OpenSF can only go so far in assuring synchronization of model execution and must rely on models being "well behaved" with respect to parallel execution.

5.4.1.1. Precautions to ensure safe model parallelization

In order to ensure safe model parallelization model developers should ensure that models are either:

- Thread safe: implementation is guaranteed to be free of race conditions when accessed by multiple threads simultaneously, or;
- Conditionally safe: different threads can access different objects simultaneously, and access to shared data is protected from race conditions.

The use of software libraries can provide certain thread-safety guarantees. For example, concurrent reads are typically guaranteed to be thread-safe, but concurrent writes might not be. Whether or not a program using such a library is thread-safe depends on whether it uses the library in a manner consistent with those guarantees. Thread safety guarantees imply some design steps to prevent or limit the risk of different forms of deadlocks, as well as optimizations to maximize concurrent performance.

There are several approaches for avoiding race conditions to achieve thread safety. The first class of approaches focuses on avoiding shared state, and includes:

- Re-entrancy: writing code in such a way that it can be partially executed by a thread, re-executed by the same thread or simultaneously executed by another thread and still correctly complete the original execution. This requires the saving of state information in variables local to each execution, usually on a stack, instead of in static or global variables or other non-local state. All non-local state must be accessed through atomic operations and the data-structures must also be re-entrant;
- Thread-local storage: variables are localized so that each thread has its own private copy. These variables retain their values across subroutine and other code boundaries, and are thread-safe since they are local to each thread, even though the code which accesses them might be executed simultaneously by another thread.

The second class of approaches are synchronization-related, and are used in situations where shared state cannot be avoided:

- Mutual exclusion: access to shared data is serialized using mechanisms (e.g. semaphores) that ensure only one thread reads or writes to the shared data at any time. Incorporation of mutual exclusion needs to be well thought out, since improper usage can lead to side-effects like deadlocks and resource starvation;
- Atomic operations: shared data are accessed by using atomic operations which cannot be interrupted by other threads. This usually requires using special machine language instructions, which might be available in a runtime library. Since the operations are atomic, the shared data are always kept in a valid state, no matter how other threads access it. Atomic operations form the basis of many thread locking mechanisms, and are used to implement mutual exclusion primitives;
- Immutable objects: the state of an object cannot be changed after construction. This implies that only read-only data is shared and inherent thread safety. Mutable (non-const) operations can then be implemented in such a way that they create new objects instead of modifying existing ones (e.g. this approach is used by the string implementations in Java, C# and python).

Thread safety

Thread safety is a simple concept: is it "safe" to perform operation A on one thread whilst another thread is performing operation B, which may or may not be the same as operation A. This can be extended to cover many threads. In this context, "safe" means:

- No undefined behaviour;
- All invariants of the data structures are guaranteed to be observed by the threads.

The actual operations A and B are important. If two threads both read a plain int variable, then this is fine. However, if any thread may write to that variable, and there is no synchronization to ensure that the read and write cannot happen together, then you have a data race, which is undefined behaviour, and this is not thread safe.

Unless special precautions are taken, then it is not safe to have one thread read from a structure at the same time as another thread writes to it. If you can guarantee that the threads cannot access the data structure at the same time (through some form of synchronization such as a mutex, critical section, semaphore or event) then there should be no problem.

You can use things like mutexes and critical sections to prevent concurrent access to some data, so that the writing thread is the only thread accessing the data when it is writing, and the reading thread is the only thread accessing the data when it is reading, thus providing the thread safety guarantee. This therefore avoids the undefined behaviour mentioned above.

However, you still need to ensure that your code is safe in the wider context: if you need to modify more than one variable then you need to hold the lock on the mutex across the whole operation rather than for each individual access, otherwise you may find that the invariants of your data structure may not be observed by other threads.

It is also possible that a data structure may be thread safe for some operations but not others. For example, a single-producer single-consumer queue will be fine if one thread is pushing items on the queue and another is popping items off the queue, but will break if two threads are pushing items, or two threads are popping items.

Global variables are implicitly shared between all threads, and therefore all accesses must be protected by some form of synchronization (such as a mutex) if any thread can modify them. On the other hand, if you have a separate copy of the data for each thread, then that thread can modify its copy without worrying about concurrent access from any other thread, and no synchronization is required. Of course, you always need synchronization if two or more threads are going to operate on the same data.

5.4.2. Parallel Model Execution Components Description

This section gives more details on the design of the execution of models in parallel. This functionality shall be achieved as an extension of the *ModelChainExecutor* class. To better understand the functionality a further decomposition of this class is needed and is provided in Figure 5-15. The classes used by *ModelChainExecutor* for parallel model execution model are described in depth in the following sub-sections.

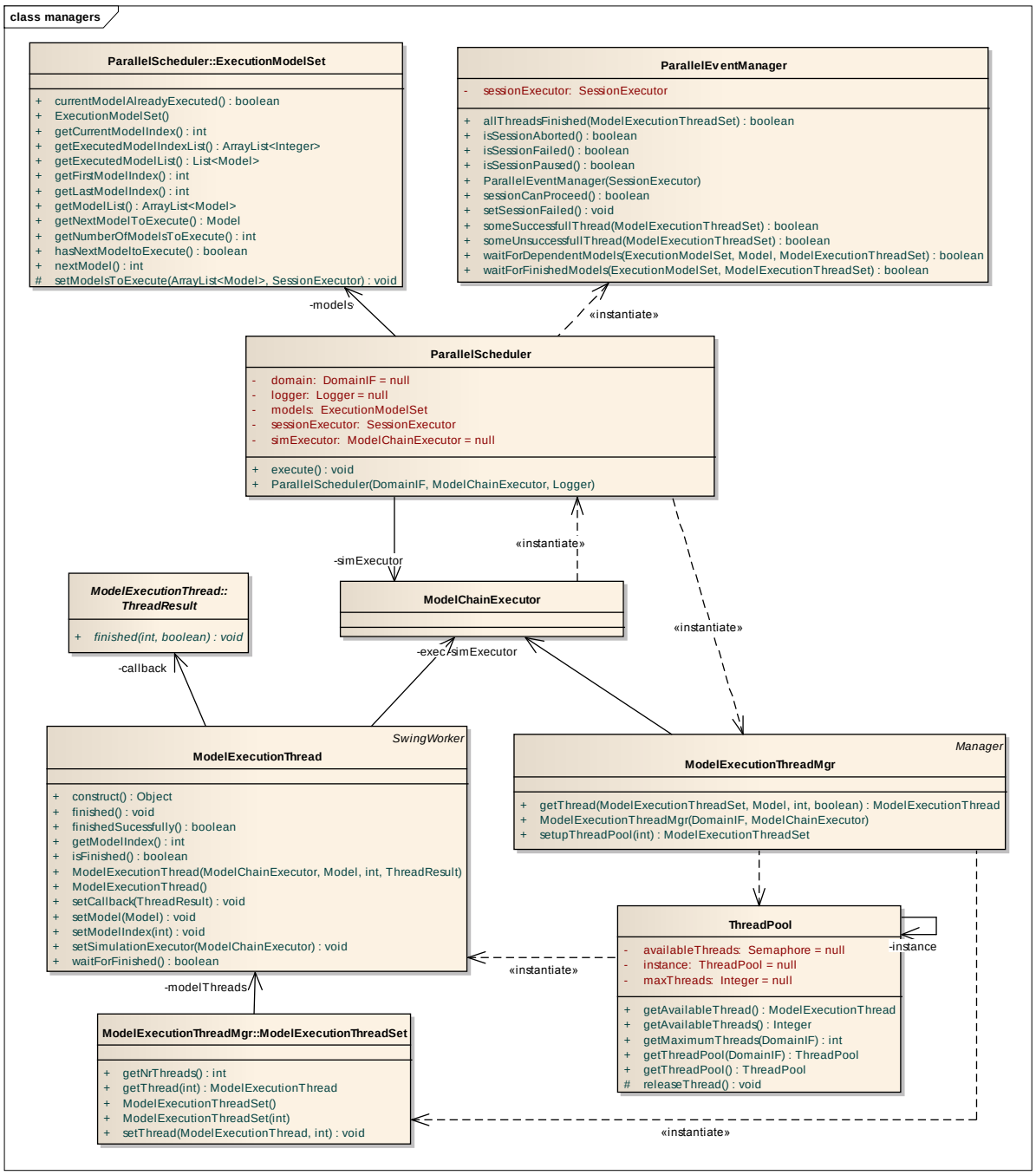

Figure 5-15: ModelChainExecutor class hierarchy for parallel model execution

5.4.3. ParallelScheduler

5.4.3.1. Type

This component is a class.

5.4.3.2. Purpose

This class is responsible for orchestrating the parallel model execution.

5.4.3.3. Function

For the session execution capability, as soon as a simulation has been invoked to run (thru the `start` method) the *ModelChainExecutor* shall create an instance of the *ParalellScheduler* class. This instance shall be active during the life of the session and shall be in charge of the execution of the models contained in the session coordinating the parallelization of models whenever applicable. The following activities are delegated in this class by the *ModelChainExecutor:*

- Analyse dependency graph between the models of a session: this analysis shall be performed at execution time and therefore no dependencies from the database are expected. Any pre-conditions for the correct execution of the simulation models are expected to have been established by the *ModelChainExecutor* class (e.g. availability of input files);
- Management of core resources: each model shall be put waiting for a core thread to be available. Delegates in class *ModelExecutionThreadMgr* the task of managing the access to the several processor core;
- Parallel execution approach.

In the case of parameter perturbation the session shots shall also be parallelized. In this case a dialog should be displayed to the user asking if shots are to be parallelized. It should be noted that the parallel approach 2 mentioned above is also expected to be consistent with the parallelization of parameter perturbation shots.

5.4.3.3.1. Related functional issues

Regarding the production of the log under the scenario of parallel model execution the proposed approach is to (a) produce a single global log containing all the messages for sessions of a simulation and written in the simulation folder, and (b) to produce one separate log per session, stored in the session folder. Note: it is already assumed that the log is being written to file and no longer to the database (refer to section 5.6.2 on the Removal of logs from database).

Regarding the displaying of the log messages with the existing implementation the log messages would be shown all mixed (coming from more than one model being executed simultaneously). The proposed approach shall be to present the logs separately on a per-model basis: each model execution shall be displayed in a separate tab of the execution window. Moreover whenever there is an ERROR log message in a given model the visual focus should be moved to the tab presenting the corresponding log and the tab should be highlighted as well.

5.4.3.4. Super-class

This class does not inherit from another class.

5.4.3.5. Dependencies

This class depends on the definition of classes like *DomainIF*, *ModelChainExecutor*, *SessionExecutor and Logger*. Additionally to support its functionalities it instantiates objects of classes *ParallelEventManager* and *ModelExecutionThreadMng*

5.4.3.6. Interfaces

This class provides the following list of public operations:

Table 14: List of ParallelScheduler class public operations

Operation name	Input	Output	Description
`execute`	void	void	Initiates and manages the parallel scheduling execution of the simulation.

5.4.4. ParallelEventManager

5.4.4.1. Type

This component is a class.

5.4.4.2. Purpose

This class handles the current status of the parallel model execution.

5.4.4.3. Function

An instance of the *ParallelEventManager* class is created for each parallel execution. This instance provides the methods to evaluate the progress of the parallel execution. The class gives feedback to the *ParallelScheduler* on whether the execution is progressing correctly or if some issue as arisen in a given execution thread deeming the simulation execution to halt.

This class also contains the logic to determine dependencies between models. In case of model dependency the execution of a given model must wait for its predecessors to conclude, while in case of model independency the models can be put to execution (possibly in parallel with others).

5.4.4.4. Super-class

This class does not inherit from another class.

5.4.4.5. Dependencies

The *ParallelEventManager* instance depends on the definition of the *SessionExecutor* class.

5.4.4.6. Interfaces

This class provides the following list of public operations:

Table 15: List of ParallelEventManager class public operations

Operation name	Input	Output	Description
isSessionFailed	void	Boolean	Determines if the current session execution has failed.
isSessionAborted	void	boolean	Determines if the current session execution has been aborted
isSessionPaused	void	boolean	Determines if the current session execution has been paused.
sessionCanProceed	void	boolean	Determines if the current session can proceed execution.
setSessionFailed	void	void	Sets the session execution as failed.
waitForDependentModels	ExecutionModelSet (the set of models to execute), Model (the model to check for dependencies), ModelExecutionThreadSet (the set of threads of an execution)	boolean	Waits for execution of dependent models.
waitForFinishedModels	ExecutionModelSet (the set of models to execute), ModelExecutionThreadSet (the set of threads of an execution)	boolean	Awaits the conclusion of the threads corresponding to the models already put in execution.
someUnsuccessfullThread	ModelExecutionThreadSet (the set of threads of an execution)	boolean	Determines if there is at least one unsuccessfully executed thread in a given thread set.
someSuccessfullThread	ModelExecutionThreadSet (the set of threads of an execution)	boolean	Determines if there is at least one successfully executed thread in a given thread set.
allThreadsFinished	ModelExecutionThreadSet (the set of threads of an execution)	boolean	Determines if all threads of an execution have finished successfully.

5.4.5. ExecutionModelSet

5.4.5.1. Type

This component is a class (internal to *ParallelScheduler*).

5.4.5.2. Purpose

Class *ExecutionModelSet* is a data type representing a set of models to be executed.

5.4.5.3. Function

This class is an abstraction representing a set of models to execute regardless of the internal data structures that support the representation of the list of models.

5.4.5.4. Super-class

This class does not have a parent super-class.

5.4.5.5. Dependencies

Instances of this class depend on the definition of the *SessionExecutor* class.

5.4.5.6. Interfaces

This class provides the following list of public operations:

Table 16: List of operations in ExecutionModelSet class public interface

Operation name	Input	Output	Description
`setModelsToExecute`	ArrayList<Model>, SessionExecutor	void	Initializes the internal data structures according to a given set of models to execute.
`hasNextModeltoExecute`	void	Boolean	Determines if there is still a model to be executed.
`getNextModelToExecute`	void	Model	Returns the next model to execute.
`getNumberOfModelsToExecute`	void	int	Determines the total number of models to execute.
`getFirstModelIndex`	void	int	Returns the index of the first model to execute.
`getLastModelIndex`	void	int	Returns the index of the last model to execute
`getCurrentModelIndex`	void	int	Returns the index of the current model being executed.
`getModelList`	void	ArrayList<Model>	Returns the list of models to execute.
`getExecutedModelList`	void	List<Model>	Returns the list of models already in execution
`getExecutedModelIndexList`	void	List<Integer>	Returns the list of model index already executed
`currentModelAlreadyExecuted`	void	boolean	Determines if the current model in execution has already concluded.
`nextModel`	void	int	Iterates to the next model to be executed.

5.4.6. ModelExecutionThreadMgr

5.4.6.1. Type

This component is a class.

5.4.6.2. Purpose

Class *ModelExecutionThreadMgr* manages the definition and access to a new *ModelExecutionThread* object.

5.4.6.3. Function

This class implements a producer-consumer design pattern to manage the use of several processor core threads. Refer to section 3.2.5 for details on the Producer-consumer design pattern used to support the design of this class.

5.4.6.4. Super-class

This class inherits from the *Manager* class so it has access to the domain layer data structure held in memory (note: the access to persistence database over a database server thru the database layer is not applicable, even though it accessible thru the *Manager* public interface).

5.4.6.5. Dependencies

This class instance depends on the definition of the *ModelChainExecutor* class. Additionally to support the functionality the class instance makes use of classes *ModelExecutionThreadSet* and *ThreadPool*.

5.4.6.6. Interfaces

This class provides the following list of public operations:

Table 17: List of operations in ModelExecutionThreadMgr class public interface

Operation name	Input	Output	Description
`setupThreadPool`	int	ModelExecutionThreadSet	Initializes the Thread Set for a given number of models to be executed.
`getThread`	ModelExecutionThreadSet, Model, int, boolean	ModelExecutionThread	Setup and return a model execution thread. Blocks until a thread is available.

5.4.7. ModelExecutionThread

5.4.7.1. Type

This component is a class.

5.4.7.2. Purpose

Objects of class *ModelExecutionThread* execute a single simulation model in a dedicated thread.

5.4.7.3. Function

Objects of this class embody the execution environment of a model. This class is responsible for setting up the thread to execute the model. This instance shall be active during the life of the model and therefore shall be responsible to properly finalise the model execution in a consistent and correct manner.

5.4.7.4. Super-class

This class inherits from the *SwingWorker* class (abstract class already used for sub-classing to perform GUI-related work in a dedicated thread) so it inherits the thread related design pattern already put in place by that abstract class.

5.4.7.5. Dependencies

This class instance depends on the definition available in the *ModelChainExecutor* class. Additionally to support the functionality the class instance makes use of interface class *ThreadResult*.

The *ThreadResult* interface is implemented by a class whose instances are intended to be executed by a thread. The class must define a method called `finished`. This interface is designed to provide a common protocol for objects that wish to execute code when the thread is finished.

5.4.7.6. Interfaces

This class provides the following list of public operations:

Table 18: List of operations in ModelExecutionThread class public interface

Operation name	Input	Output	Description
`construct`	void	Object	Execute the model and compute the value to be returned by the thread execution. Returns null if either the constructing thread or the current thread was interrupted before a value was produced.
`finished`	void	void	Called on the dispatching thread (not on the worker thread) after the `construct` method has returned.
`isFinished`	void	boolean	Determines if the thread execution has finished
`finishedSucessfully`	void	boolean	Determines if the model execution has finished successfully.
`waitForFinished`	void	boolean	Waits for the model to finish. Performs a `join` with the model executing thread thus blocking until it that thread is finished.

5.4.8. ModelExecutionThreadSet

5.4.8.1. Type

This component is a class (internal to *ModelExecutionThreadMgr*).

5.4.8.2. Purpose

Class *ModelExecutionThreadSet* manages a set of threads for module execution.

5.4.8.3. Function

This class is an abstraction representing a set of *ModelExecutionThread* objects regardless of the internal data structures that support the representation of the list of threads.

5.4.8.4. Super-class

This class does not have a parent super-class.

5.4.8.5. Dependencies

This class instance has no dependencies to other classes. Nevertheless to support the functionality the class instance makes use of objects of class *ModelExecutionThread*.

5.4.8.6. Interfaces

This class provides the following list of public operations:

Table 19: List of operations in ModelExecutionThreadSet class public interface

Operation name	Input	Output	Description
`setThread`	ModelExecutionThread, int	Void	Sets the thread of a given model.
`getThread`	int	ModelExecutionThread	Returns the thread for a given model index.
`getNrThreads`	void	int	Returns the number of threads of the object thread Set.

5.4.9. ThreadPool

5.4.9.1. Type

This component is a class.

5.4.9.2. Purpose

Class *ThreadPool* manages the concurrent access to the operating system threads.

5.4.9.3. Function

This class implements a singleton design pattern to manage the low level access to operating system object threads.

A limit to the number of cores to be used is set thru a global configuration parameter. The parameter value has the following semantics: 0 – no limit to the number of threads accessible; 1 – no parallelization (single thread architecture); N - maximum number of core threads to be used.

To synchronize the available resources a semaphore is used in controlling the access to the low level thread objects.

Refer to section 3.2.5.3 for details on the Thread Pool design pattern used to support the design of this class.

5.4.9.4. Super-class

This class does not have a parent super-class.

5.4.9.5. Dependencies

This class instance has no dependencies to other classes. Nevertheless to support the functionality the class instance makes use of objects of class *ModelExecutionThread* as well as accessing the domain layer data structure held in a *DomainIF* instance.

5.4.9.6. Interfaces

This class provides the following list of public operations:

Table 20: List of operations in ThreadPool class public interface

Operation name	Input	Output	Description
`getThreadPool`	DomainIF	ThreadPool	Accesses the singleton ThreadPool instance.
`getMaximumThreads`	DomainIF	int	Gets the maximum allowed execution threads.
`getAvailableThreads`	void	int	Gets the number of available threads.
`getThread`	void	ModelExecutionThread	Returns a thread. Blocks until a thread is available.

5.5. Graphical User Interface Design

This section provides a description of the design features common to the openSF GUI (in advance, the MMI, Man Machine Interaction).

The MMI presented accepts input via devices such as computer keyboard and mouse and provide articulated graphical output on the computer monitor. This certain MMI implements also the OOUI (Object Oriented Interface) paradigm because it is constructed from different pieces, or objects with several properties and operations.

The openSF MMI also follows the Multiple Document Interface (MDI) pattern. This approach has been chosen because of its flexibility, as it let users to organize the layout of the information as desired, showing only relevant windows and in the way users want.

The MDI pattern consists of a “parent” container that can host inside several “internal frames”. These internal frames are intended to present independent modules of the simulator. For example, each time the user wishes to perform operations with the list of models of the system, a “model manager” frame will pop-up inside the bounds of the main window listing the list of models currently available within openSF. This is applicable to other system elements such as simulations, sessions, stages etc…

The design of the openSF MMI has been based in the use of Java SWING and AWT technologies. These Java graphic libraries are widely used to make applications because of its ease of use and the portability advantages they bring to developers.

Figure 5-16 shows a screenshot of the openSF main window. This is the first window users will see anytime they launch the application.

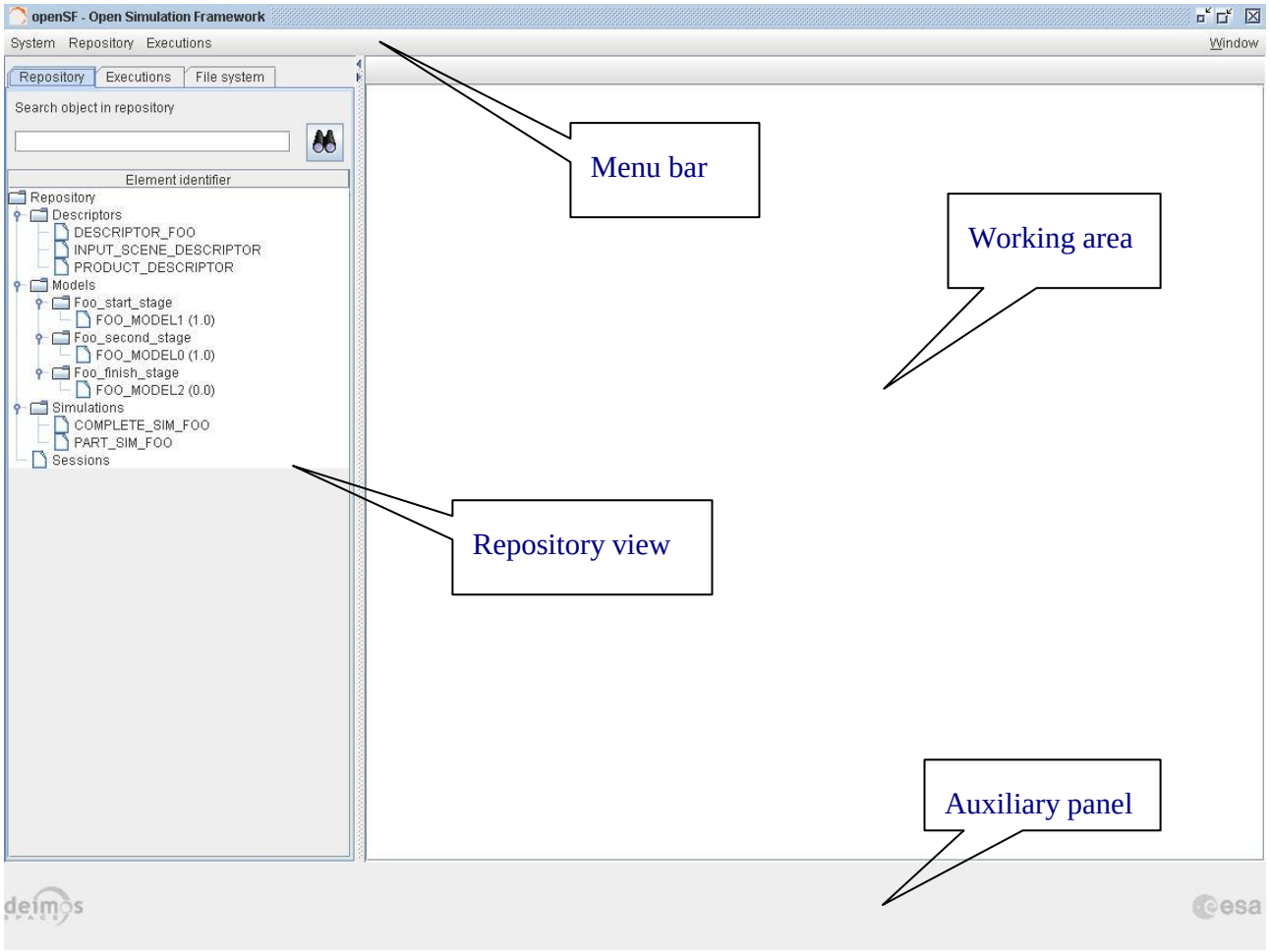

Figure 5-16: Main window appearance

All the windows have commons operations to help their usability: main window, internal frames or dialogs can be closed, resized, maximized or minimized to fit the user's needs.

This main window (Figure 5-16) includes a menu bar to provide keyboard and mouse access to the simulator main functions as well as functions regarding frames management and application basis.

5.5.1. Window Design

This section describes the windows main classes and modes.

The MDI pattern provides some useful capabilities to arrange internal frames (e.g. Figure 5-17) appropriately like cascading or tiling them. Also internal frames can be "iconized" to give more available space. When a user iconizes a frame it can be restored by clicking the button with its name in the "available frames" toolbar or the corresponding menu item at the "windows" menu.

Occupying the central and main region exist a working area. This area is where all internal frames are going to be created and main interaction is hold. Besides that, this working area implements a "scrollable" panel in order to easily navigate through frames surpassing its bounds.

At the left side of the working area there is a system objects navigator, a "repository view" which function is to provide a quick access method to every item known by the system (models, simulations,

sessions, etc). There is also a file system browser to navigate through the contents of the application's directory.

The main window's footer area shows some corporate logos and is meant to provide information to assist the user.

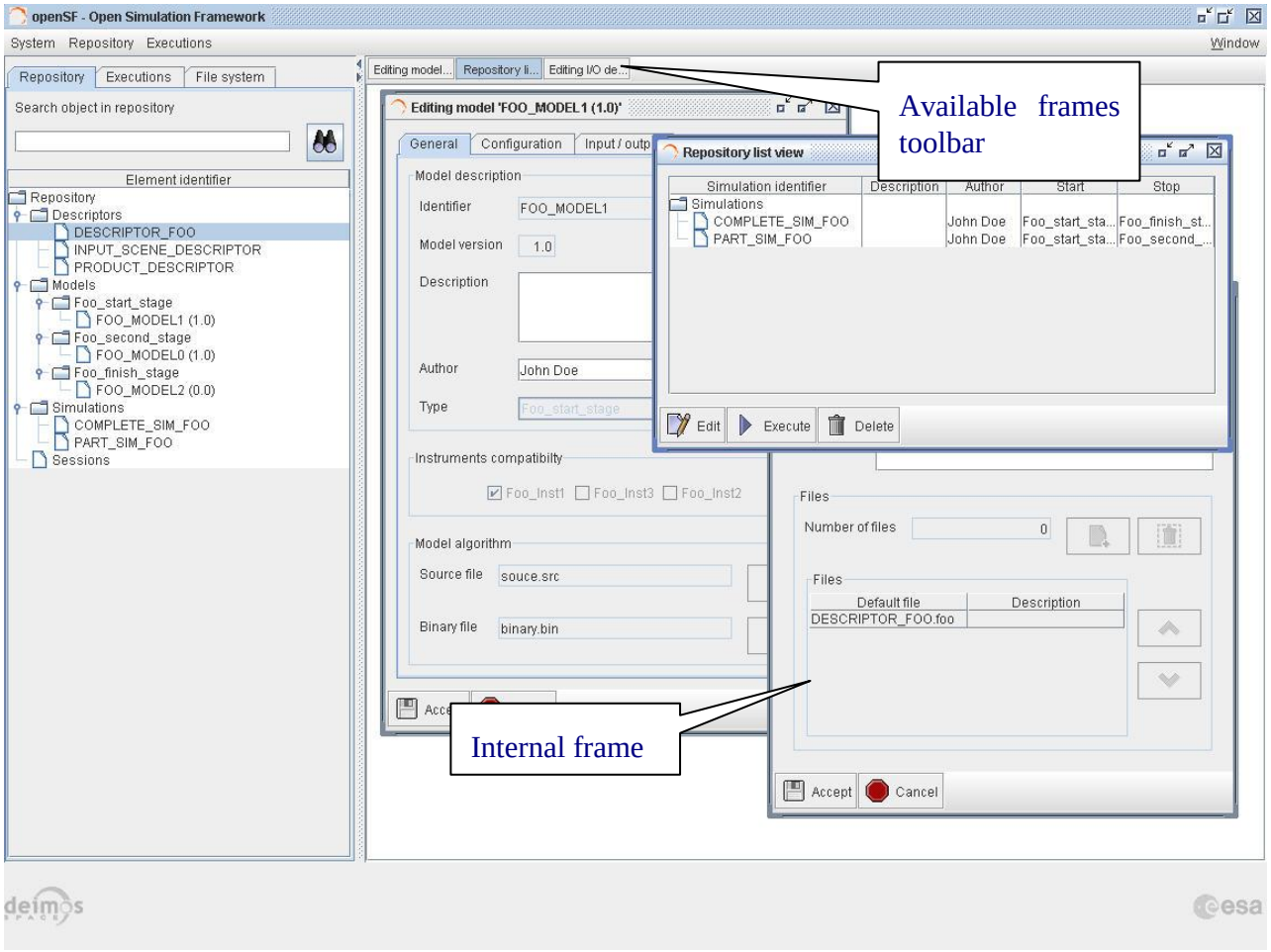

Figure 5-17: Main window appearance showing internal frames and scroll panel

The MMI provides a menu bar (Figure 5-18) at the upper side of the main frame to show some capabilities of the system.

Figure 5-18: Detail of main menu bar

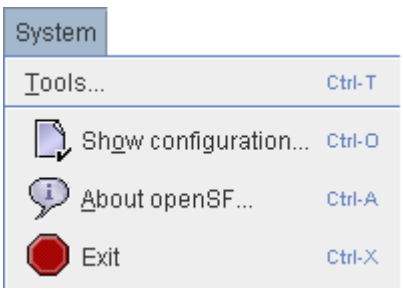

Figure 5-19: Detail of a menu, showing menu items

It is shown in Figure 5-19 that a menu item is an icon with graphically describes the function, the name of the function and a "quick access" key combination. Users can quickly access this functionality pressing this key combination or the letter underscored in the function name while the menu is rolled down.

There are also some contextual or "pop-up" menus that users can access by clicking the right button of the mouse while over certain controls. These "pop-up" menus have the same appearance as menus rolling down the menu bar, but coming from some other component.

Figure 5-20: Detail of a contextual menu

It can be seen in Figure 5-20 that a pop-up menu acts exactly like a menu at the main frame. They also provide mouse and keyboard access to certain capabilities.

5.5.2. GUI Standards

There is no applicable standard to this GUI. However we have followed some guidelines described in [RD GAL MMI] to approximate this GUI solution to other well-known and common solution in the aerospace sector.

5.5.3. Components, Libraries and Tools

This section describes the libraries and, standard widgets used for the GUI.

This MMI is being totally developed in JAVA code with runtime environment version #1.5.0_09.

SWING and AWT libraries are being intensively used throughout the MMI modules. A description of the SWING widgets that comprises the MMI can be found at [RD SWING] and the API documentation at [RD SWING API].

Every single components of the MMI set will be shown in the Java look-and-feel (cross-platform), i.e. visual aspect will be the same independently on which operating system this interface is running on.

Some images have been extracted from the Java look-and-feel repository and some others (mainly corporate logos) are taken from its public web sites.

5.5.4. Generic Functions, Dialogues and Displays

This section is meant to describe the design of generic functions, dialogues and displays used at the GUI.

There are some functionalities of the MMI that show a "file chooser" dialogue as shown in Figure 5-21.

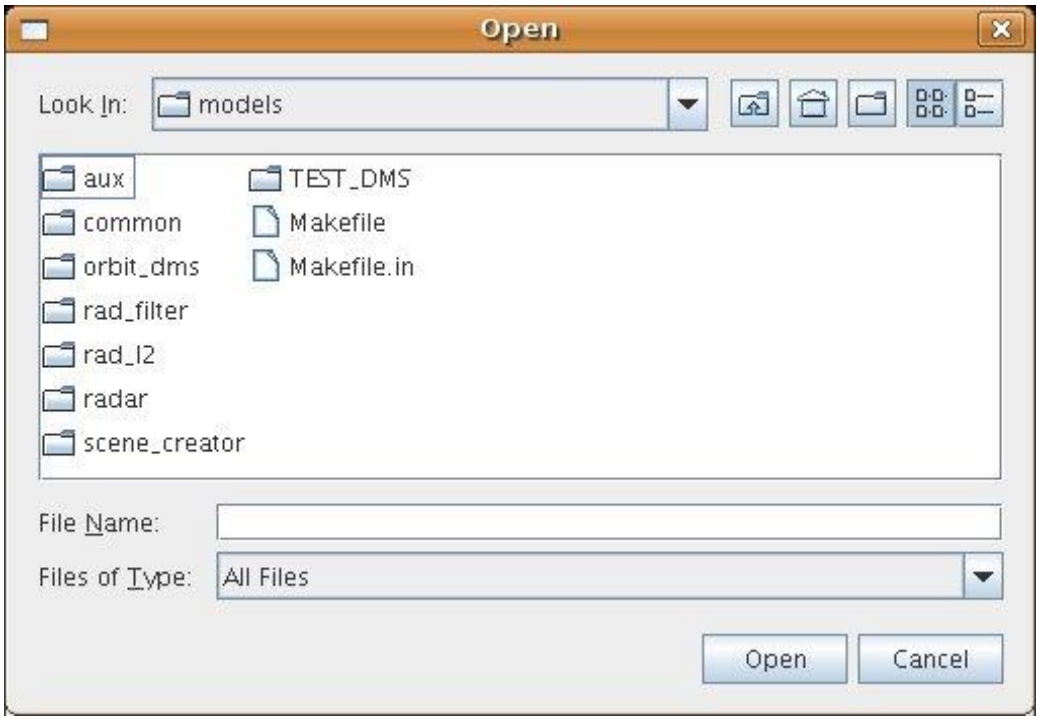

Figure 5-21: File chooser dialogue

This dialogue helps the user to browse the system directory to select a certain file or list of files. It provides some sorting, filtering and file operations very common and already known for the majority of the users.

Throughout the openSF MMI some functionality could show information to the user and could ask for some input in response to an answer. The MMI will present some "modal" dialogues that will get the system focus until the user provides an answer. These modal dialogues will block the input to other areas of the application until a response is given.

Figure 5-22: Dialogue example

These dialogues will typically provide a warning message with an "OK" button or give a yes-or-no question or another question with different options. The dialogues will provide information with a clear description of the event.

5.6. Design approach for openSF V3 additional functionalities

This section presents the design approach for the additional functionalities requested by [AD-CCN1].

5.6.1. Framework revision for flexible session management

The feature requests underlying REQ-3, REQ-4, REQ-7 and REQ-9 of [AD-CCN1] are considered to be interrelated functionality to be tackled in a common approach targeting openSF flexibility. The proposed approach implies reformulation of the openSF framework to allow more flexibility on session management. This shall imply revisiting the openSF architecture with impact on both application and database organization as it shall imply the reformulation of the simulation concept.

5.6.1.1. Simplification of the management of the model chains

The purpose of simplifying the management of simulations is to provide flexibility in the execution of a given chain.

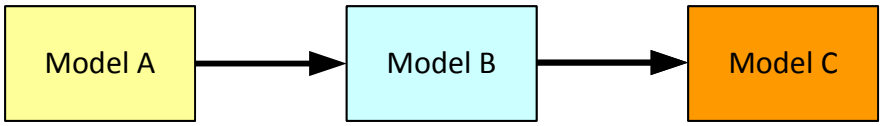

Figure 5-23 Simple model chain

Taking as example the model chain in Figure 5-23 consider that the first executable (Model A) is a very time-consuming model that can be executed once to feed subsequent models. The idea is therefore to permit users to create a sub-chain with models B and C using the same simulation definition. The execution will take as input (for Model B) the outputs of model A, previously stored from another execution.

5.6.1.2. Select model versions for a simulation execution

Similarly to the above capability, and in order to make more flexible the definition of a given simulation, this functionality shall allow selecting a specific version of a model for a simulation execution (as depicted in Figure 5-24 below).

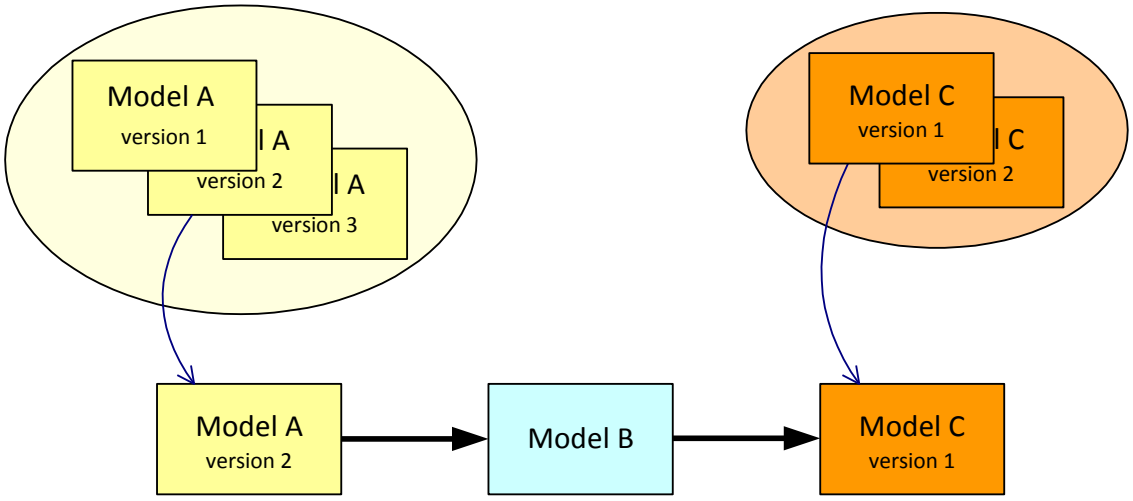

Figure 5-24 Model chain with different model versions

5.6.1.3. Bypass/switch-off models

This functionality enables users to switch off certain models when running simulations. As a result, openSF will inform the user of the data files needed to be provided due to the omission of models and their corresponding outputs.

5.6.1.4. Rerun a session from a previous point

The idea is to allow users to skip models at the beginning of the simulations, and therefore start sessions from a certain point. However, the data from non-executed models is needed for the re-run. Before executing the simulations the user needs to define the data files needed for the run.

The figure below shows a simple example. Models A, B and C constitute the simulation. If we want to rerun it but starting from B, we need to provide the output of Model A, from a previous run.

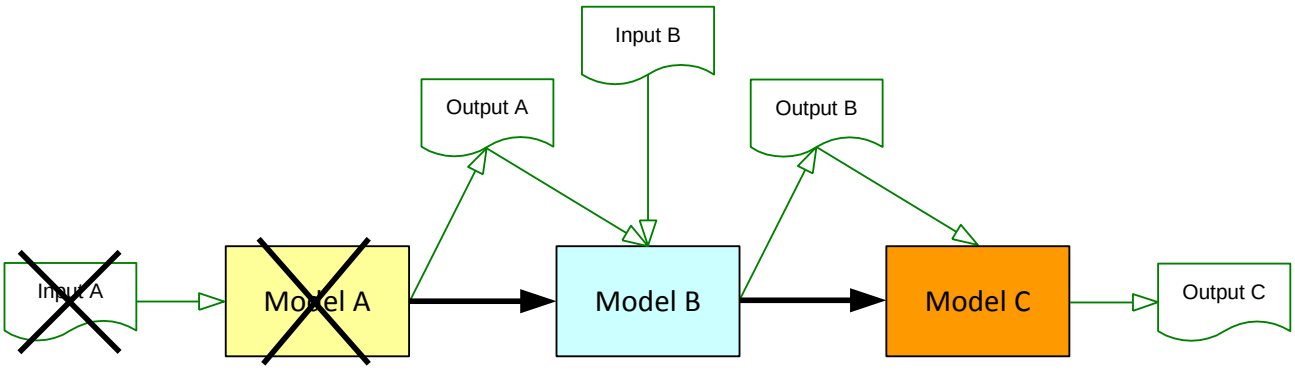

Figure 5-25 Run simulation from Model B

5.6.1.5. Flexible session management design approach

Currently the capabilities mentioned in the above sub-sections are not available due to architectural constraints implemented both in the openSF controller layer (designed in section 5.2.2.2 controller) as well as in the database layer - data storage (designed in section 5.2.4 database). Therefore it is necessary to remove these data storage constraints and adapt openSF controller layer to support these new functionalities.

In order to cover all the requirements regarding framework flexibility the following design decisions have been taken:

- Remove the strict dependency between *Session* and *Simulation* entities.
- Session will be a chain of models instead of a set of simulations, this will allow to easily switch off/bypass models or execute different model versions. Note that *Simulation* entity will not be removed and users will have the capability of adding a Simulation (pre-defined chain of models) to a session.
- This change makes the new framework not compatible with previous versions, change in the database structure, but impact has been minimized from simulation to a set of models during session creation/edition. For more details about database compatibility see section 5.7.

The design components affected by these new features are:

- Database: the table relating sessions and simulations will be changed by a table relating sessions and models to be executed.

- *ModelChainExecutor* will be updated with a model chain executor, not related with *Simulation* entity.
- Session folder structure will be grouped by simulation if possible.
- The user interface for Session creation and edition will be updated providing the user mechanisms for performing the new capabilities. A contextual menu for switching between model versions . Addition/deletion of a simulation will be updated, allowing also the addition of a single model.

5.6.2. Removal of logs from database

It is requested in [AD-CCN1] to remove the storage of logs from the database aiming to prevent the database from being overloaded due to the big size of log messages that a session/simulation execution may involve. Log messages shall be kept as files that are stored as part of the session execution data. They can therefore be consulted at any time by users either from the openSF HMI or directly from the file system.

When displaying logs thru the HMI the system shall limit the size of the parsed data to a maximum number of log entries (typically the most recent ones). Only when the operation for searching messages is triggered shall the parsing carry on the loading of all the log messages. In either case after closing the log view the parsed data should be released from memory.

The proposed implementation approach includes performing a drop of the log table from the openSF database. It should be noted that this implies that openSF V3 cannot be used directly over an existing instance of openSF. Refer to section 5.7 for details on the Migration to openSF V3 procedure.

Log messages shall be dumped to file in a textual format, i.e. containing exactly the messages sent by the models (via OSFI) and intercepted by openSF.

Regarding the production of the log under the scenario of parallel model execution (as discussed in section 5.4.3.3.1) the proposed approach is to (a) produce a single global log file containing all the messages for sessions of a simulation and written in the simulation folder, and (b) to produce one separate log file per session, stored in the session folder.

5.6.3. Removing intermediate data during simulation execution

OpenSF stores all data produced as output of the running models of simulations. The purpose of this capability is to allow the possibility for the user to decide whether they want to remove all the intermediate data generated as part of a simulation execution.

Thus, in the example provided in the figure below, if the user selects the removal of intermediate data, the files coloured in red would be removed once the simulation has been completed.

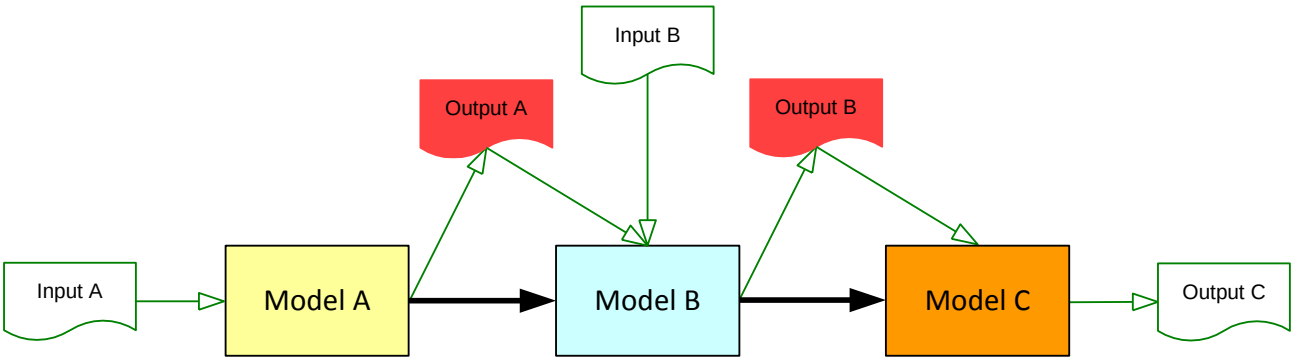

Figure 5-26: Removing intermediate data of a simulation

To implement this functionality the following approaches were analyzed:

Approach 1 - Add a global configuration parameter, activating the functionality with a warning message to the user that intermediate data shall be removed;

Approach 2 - Add a per session configuration parameter set thru a check box at session edition level (located to the left of the "author" field, making this one shorter). Is should be noted that this approach implies changing the database structure to include the additional session setting.

Approach 2 is the proposed implementation. Furthermore intermediate data is to be removed at the end of the session and only upon the successful execution of the session.

5.6.4. Capability to copy elements

Functions to copy elements shall be made available in openSF. This operation shall be made available for each openSF main concept: Descriptor, Model, Simulation, Session and Tool (it shall not be applicable to the Stage concept).

Implementation shall be based on the "clone" method already implemented for every concept. Implementation note: it is important to always ensure that every new attribute added to a given concept class is included in the "clone" method.

Regarding the graphical interface this operation shall be deployed in the contextual menu (right-click) of the given concept instances, presenting a new option. Thus users shall only need to define a new name for the copied element.

5.6.5. Export capability

This capability deals with the possibility of exporting the data associated to a model that has already taken place in a simulation. Thus, the data exported is comprised by the model configuration and input files. The inverse operation shall also be implemented, meaning that a model can be imported into an openSF instance from the data obtained from the export operation. As the contents of the export relate to data files, it is required that the model exists in the target openSF instance.

During the analysis phase there were some doubts raised regarding this functionality since the export/import capability is already developed in openSF V2.2.1. The existing functionality exports an entire session, including: (a) an SQL script that allows rebuilding the simulation structure in the database, and (b) a tarball with all input and configuration files for the several session models.

For now it shall be assumed that the intended export functionality is to export the data related to only one model of a given session. The following functionality shall be implemented:

- open a dialog with the list of models of the session to be selected;

- export a tarball with input and configuration files for only the selected model.

5.6.6. Defining openSF elements externally

This capability is related to defining openSF elements externally, not using the HMI. Thus, elements such as IO descriptors, models, simulations and session can be managed via XML files. Afterwards these definitions can be imported to openSF. In the following sub-sections we discuss the technique for importing data from XML into an openSF instance database.

5.6.6.1. Importing external definitions

The proposed implementation approach is through the use of the "`LOAD XML`" MySQL statement. An implementation of this new SQL statement has been accepted for MySQL 6.0. `LOAD XML` greatly simplifies the task of importing data from an XML file into a MySQL table, without having to use stored procedures. The (partial) syntax for this SQL statement is as shown here:

```
LOAD XML [LOW_PRIORITY | CONCURRENT] [LOCAL] INFILE 'filename'
[REPLACE | IGNORE]
INTO TABLE [db_name.]tbl_name
[ROWS IDENTIFIED BY '<tagname>']
```

It should be noted that in order to support this approach the MySQL database software must be migrated from the current openSF supported version (mysql v5.1) to the most recent version available for the Linux target platform (mysql v5.6 – refer to http://dev.mysql.com/downloads/mysql for version details).

5.6.6.2. Defining elements - XML file format

The `LOAD XML` statement reads data from an XML file into a table. `LOAD XML` supports three different XML formats:

- Attributes are interpreted as column names, and attribute values as interpreted as column values:

  <row column1="value1" column2="value2" .../>

- Tag names are interpreted as column names, and the content of these tags are interpreted as column values:

  <row>

  <column1>value1</column1>

  <column2>value2</column2>

  </row>

- Table column names are derived from the name attributes of <field> tags, and column values are taken from the contents of these tags:

  <row>

```
  <field name='column1'>value1</field>
  <field name='column2'>value2</field>
</row>
```

The selected format to be used for the openSF XML import file corresponds to the third format listed above. This is also the format used by MySQL tools such as mysqldump in XML output mode (that is, using the --xml option). This implies that it is possible to easily extract the information from currently existing openSF instances database in a format that later on can be imported using the new import functionality.

By ensuring the compatibility between the new import file format and the mysql export capabilities minimizes the risks of having to upgrade the MySQL version required for the use of the `LOAD XML` statement. Refer to section 5.7 for details on the Migration to openSF V3 procedure.

Further details regarding the XML file format for defining openSF elements externally can be found in the [AD-ICD].

5.6.7. Simplify session file and directory names

openSF approach for naming session execution directories as well as session execution supporting files involves the use of names with a timestamp. The use of a timestamp is meant to ensure a unique identification of the session folder and files. Nevertheless, upon openSF user's request, it was identified that handling such names with timestamp is not "user-friendly". In order to overcome this inconvenience openSF shall support a more user-friendly naming.

In order to simplify session directory names as well as other session execution supporting files symbolic links shall be used. Each time a session is executed a Linux symbolic link shall be generated in the file system with the name of the session being executed and pointing to the corresponding session execution directory. This means the current organization based on having timestamps in session folders is kept but a simpler way to access such folders is provided. Each time a session is re-run the symbolic link is re-generated pointing to the latest session execution (the one with the latest timestamp).

5.7. Migration to openSF V3

Some of the above design approaches for new capabilities of openSF V3 prevent this new software version to be used directly over an existing database instance. The following restrictions were identified:

- Hard restrictions:
  - Migration of MySQL version from v5.1 to v5.6 (the most recent version available for the Linux target platform);
  - Drop of the log table from the database;
- Soft restrictions:
  - Change the database structure to include an additional session configuration parameter to support the removal of intermediate data capability.

To support the migration procedure to openSF V3 a script shall be developed reflecting the following actions:

- Export the full contents of an existing openSF instance database. The database shall be exported in the same format as specified for the new import capability (refer to section 5.6.6 for details);
- export the contents of the log table of an existing openSF instance database to the corresponding log files;
- Initialize the additional session configuration parameter that supports the removal of intermediate data to a default value (deactivated).
- Remove the database table that link *Simulation* with *Session* and add a new one relating *Model* with *Session*.

A detailed set of instructions to support the porting from previous versions to openSF V3 shall be presented in [AD-SUM].

6. OPENSF PARAMETER MANAGEMENT SYSTEM

This section gives a description of the design solution for the openSF Parameter Editor. It starts with a brief definition of the openSF parameter management system and followed with:

- The description of the Parameter Editor purpose and functionality.
- A brief design overview and the analysis of the application use cases.
- Detailed system design with the most relevant package and class diagrams

6.1. Parameter Editor Overview

DEIMOS is currently developing, in the frame of the Sentinel-3 Optical System Performance Simulator (O-SPS), a parameter management tool used for creating and editing parameters as well as setting relationships between them with the objective to ensure the consistency of the models' configuration.

This tool is born from the necessity of performing a simulation consistency checking before running the simulation chain. Analysing the parameter consistency before executing the models minimizes the potential problems that could arise during the run. This aspect is really relevant when the simulator is related to on-board and operational software where resources and time consumption are huge.

The Sentinel-3 Optical System Performance Simulator is an ESA activity led by Thales Alenia Space France, and DEIMOS is the prime contractor for the activity.

In the frame of openSF project this application has been considered generic enough to be applicable for all simulation projects that use openSF as framework.

The next section presents an overview of the parameter editor functionality and purpose.

6.2. Parameter management system

OpenSF parameter management system is composed of two software modules, a parameter rule editor and a parameter editor. The first is used offline, before the simulation definition etc, and consists in a simple grammar and a graphical editor allowing the user to define a set of rules. These rules will be used to validate the parameters entered by the user in the session definition stage.

6.2.1. Parameter Rules - Grammar Definition

A simple grammar has been designed for defining the rules that govern the parameter editor. This grammar is based in a XML syntax detailed below.

A single rule is composed of:

- A unique rule identifier

  <rule id="ID">

- An operation tag, nested to the identifier. There are 3 operation types:
  - **Condition**: <condition type="ConditionType">. Condition type is a list of the most used logical operators (equals, exists, greater than, etc…)

- **Action**: <action type="ActionType">
- **IF statement**: nested to an "if" tag, a condition and an action.

This grammar allows the user to define rules (conditions, constraints etc…) when setting models parameters during the session definition or edition. The use of XML language for grammar definition is common in the software application world as it is flexible and provides a way to validate the syntax (XSD schemas).

The openSF Parameter Editor also provides a graphical front-end for the creation and edition of this grammar, from this point called openSF Rules Editor.

6.2.2. Parameter Editor

The Parameter Editor is a graphical interface to visualize and edit the configuration files involved in a simulation.

Within the openSF Parameter Editor users are able to:

- ❑ Open a rules file in order to validate a set of parameters with it.
- ❑ Edit the parameter rules file.
- ❑ Check the errors through an information log panel.
- ❑ Visualize all the parameters of a configuration file with an intuitive tree view.
- ❑ Create and delete parameters.
- ❑ Edit the values of a parameter.
- ❑ Save the configuration file.

6.2.3. Road from S3-OSPS to openSF

In order to adapt the Sentinel 3 O-SPS Parameter Editor to openSF, the development team has performed the following tasks:

- ❑ Re-vamping of the rules: Currently the set of rules used in the S3 Parameter Editor is customized for fulfilling the Sentinel 3 project requirements. The set of rules has been studied and, making an abstraction, generalize them to be useful for all projects based on openSF.
- ❑ Seamless integration: the parameter editor currently manages the configuration files related to models within Sentinel 3 simulation chain. This mechanism has been adapted in order to handle an arbitrary number of configuration files suitable for any kind of simulation chains.

6.3. Parameter Editor – Design Overview

The design of the openSF Parameter Editor is intended to meet the following goals:

- ❑ having a simple data model for rules definition
- ❑ seamless integration with current and older openSF versions
- ❑ using an XML-based syntax
- ❑ supporting use of XML schema for validating the syntax

- providing a friendly graphical user interface
- ease the simulation definition stage providing a tool for consistency checking

The design standards that drive the Parameter Editor development are the same as the ones used for openSF application, such as:

- Model View Controller paradigm
- Singleton Pattern for rules message terminal
- Prototype Pattern for the objects visualization
- Factory Pattern for the dynamic rules interpreter

6.3.1. Parameter Editor - Functional Requirements

In this section a diagram is presented showing the use cases for the openSF parameter editor. These have been extracted from the Sentinel 3 OSPS requirements document.

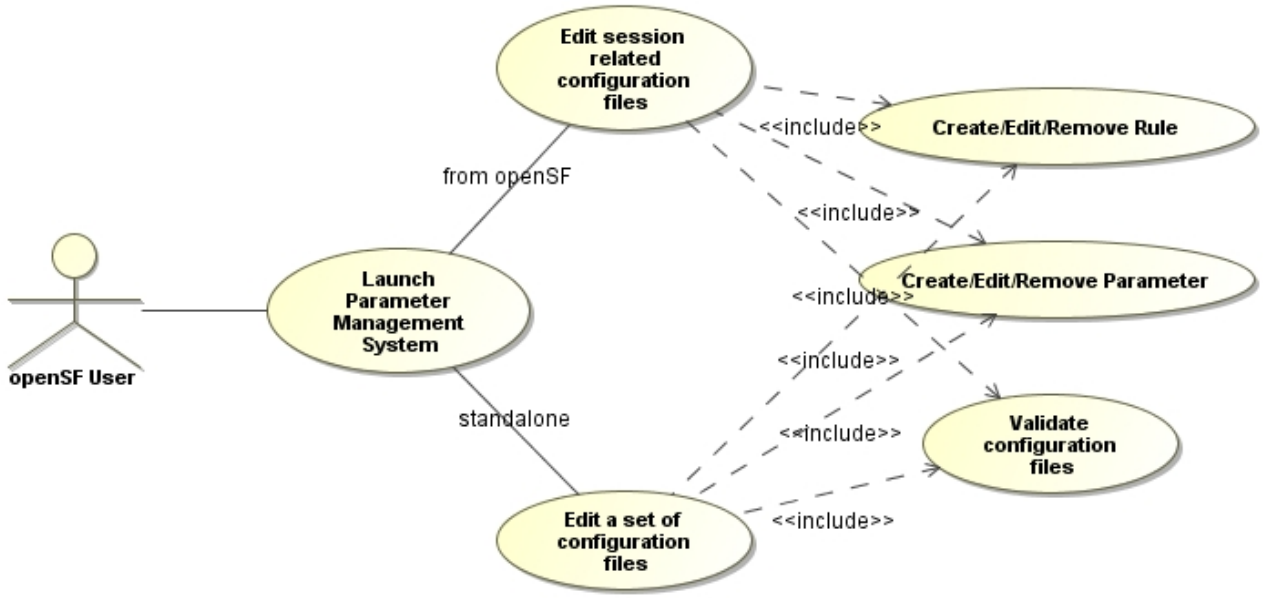

Figure 6-1: Parameter Editor high level use cases

The diagram presented in illustrates the use case diagram for openSF parameter management system, which depicts the context of the whole application, and the actor that interacts with it. In this figure only the most relevant use cases have been depicted and those are:

- ***Create/Edit/Remove Rule**:* this case covers the definition, edition and deletion of a rule within the system.
- ***Create/Edit/Remove Parameter**:* this case covers the definition, edition and deletion of a parameter within the system.
- ***Validate configuration files**:* this action represents the validation of the configuration parameters with the rules defined within the system.

Please notice that there are two different execution modes for the Parameter Editor, one when it is launched from openSF and consequently related with the session definition and the other when it is executed as a standalone application allowing editing a set of configuration files.

6.4. Parameter Editor – System Design

This section gives a brief description of the method used for the architectural design, the Unified Modeling Language used for formal diagrams and the system background and context.

6.4.1. Design Method

The application is distributed in packages used to organize the namespace for packages, classes and interfaces.

In the design process of the system the following conventional guideline to name Java components has been used throughout the whole document:

- **Classes:** class names should be nouns, with the first letter of each class capitalized, such as *Rule* and *Parameter* classes.
- **Packages**: package names should be also nouns, with the first letter in lowercase, and the first letter of each internal word in capitalized, such as *manager* or *rulesEditor* packages.
- **Interfaces**: interfaces names should be capitalized like class names and must end with the suffix "IF" or "Interface", such as *RulesInterface*.

6.4.2. System Decomposition

The openSF parameter management system is decomposed in four high level packages called *view*, *domain*, *manager* and *support*. The first three are a direct consequence of the 3-tier architecture approach from the model-view-controller paradigm and the last one is created for support purposes and to give some useful services to every package in the system such as, system constraints, frequent-used functionalities etc.... In the following diagram the high-level hierarchical structure of these packages is shown:

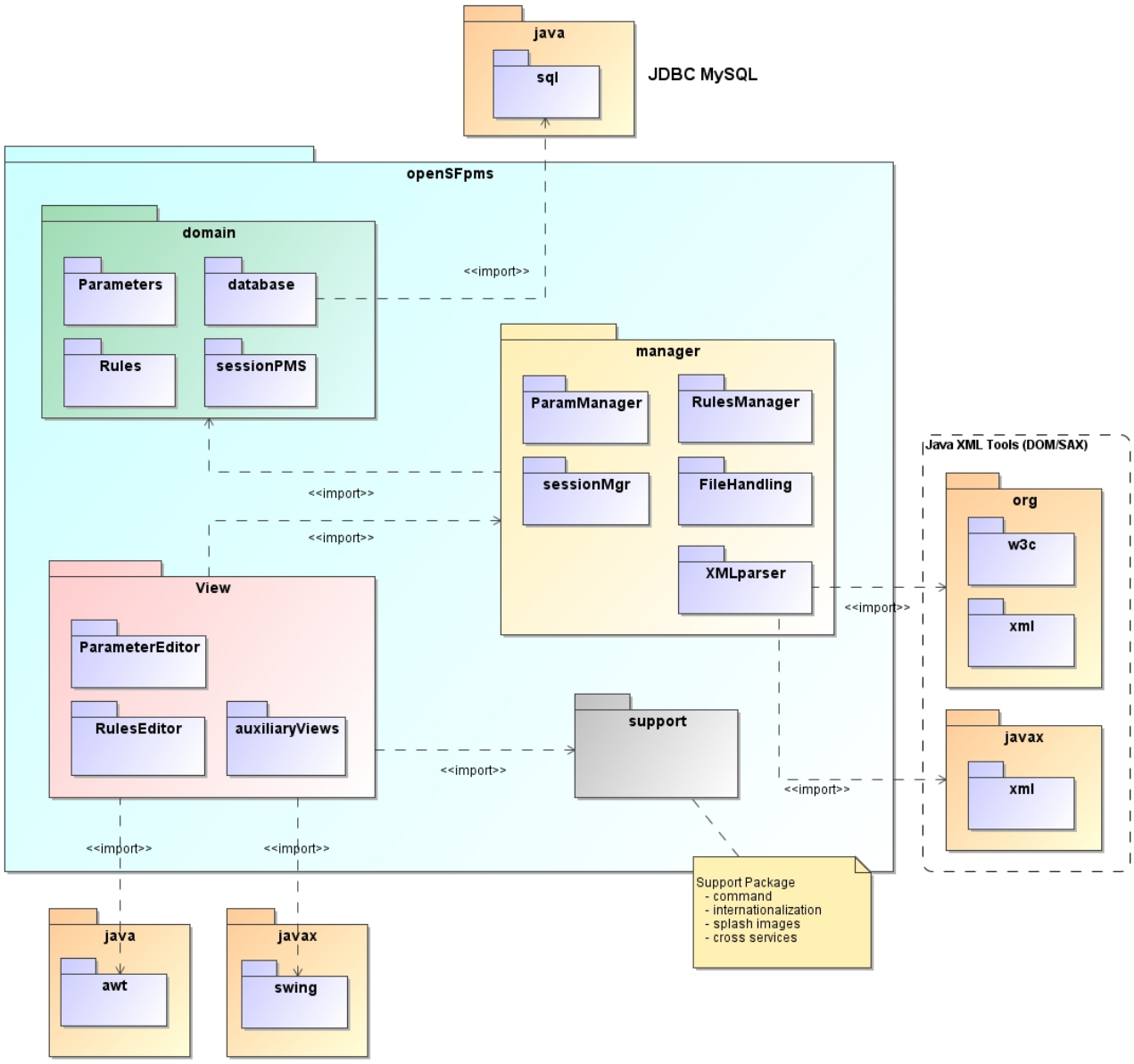

Figure 6-2: Parameter Editor High Level Architecture diagram

- **view**: Contains packages and classes related to the appearance and behaviour of all visible components of the graphic user interface (windows, frames, visual components and other widgets). This package shares and implements a public interface that could be accessed by the *support* and *manager* layers. It has a strong binding with the *manager* package playing the controller role in the MVC design approach.
- **domain**: This is the core of the parameter management system. It is responsible for all activities related with the specific domain purpose such us storage and entity object representation. This involves the capabilities to carry out the definition and management of the necessary elements identified in the domain: parameters, rules, configuration files, etc… This package is also in charge of storage user sessions in a persistent layer (file/database)

- **manager**: These classes are meant to manage, and control the sets of domain objects (elements). Thus, they shall be in charge of managing the different kinds of elements in the system: rules, parameters, namespaces etc…
- **support**: Contains all classes related to the whole system initialization accessing to external resources and some utilities to be used under the whole system scope. The *support* package shall be accessible from all packages within the application as it provides auxiliary functionalities that all layers have in common (OS specifics, filesystem interaction etc…).

The *domain* and the *manager* packages correspond to the *model* and *controller* layers respectively, from the MVC design paradigm. The view package as it names points, corresponds to the view layer from the MVC.

The base namespace of the system and of these packages is *openSFpms*.

6.4.2.1. domain

This package corresponds with the domain layer of the three-tier paradigm.

This package represents the core of the system. All packages and classes related to the parameter consistency checking are grouped here. From this package it is controlled also the persistent layer of the parameter editor. There are four packages inside it as shown in Figure 6-3.

- ***parameters*:** this package contains the classes that represent the different kind of simulation parameters involved in openSF.
- ***sessionPMS*:** this package is in charge of the persistent layer management. It is foreseen two different mechanisms, one using the openSF database and the other storing the session info into a file. This second mechanism is available when the Parameter Editor is in standalone mode.
- ***rules:*** Contains all the classes representing the constraints and relationships between simulation parameters.
- ***database***: This package contains classes designated to control the connection to an external database server and to perform queries and updates against it.

Figure 6-4 and Figure 6-5 show the class diagrams for parameter and rules packages respectively.

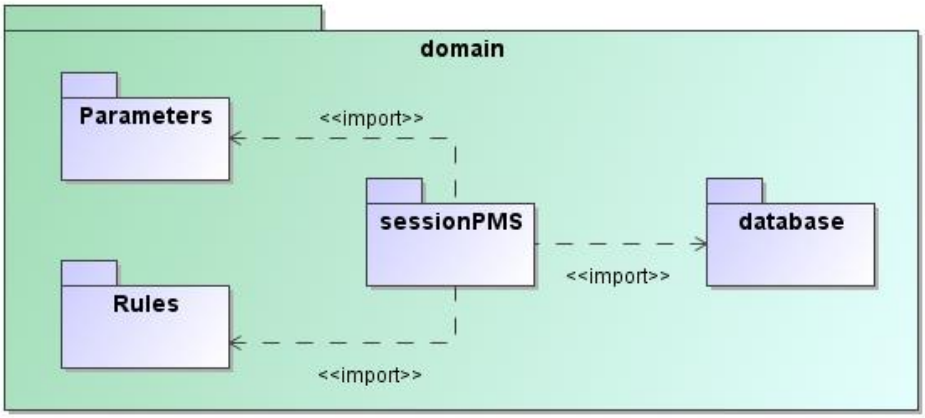

Figure 6-3: openSFpms.domain packages diagram

6.4.2.1.1. Parameter

6.4.2.1.1.1. Type

This component is a class.

6.4.2.1.1.2. Purpose

This class provides a representation of the common attributes for a simulation parameter.

6.4.2.1.1.3. Function

This class gives an interface to retrieve some parameter attributes without taking into account the parameter type (Integer, Double, String, File or Boolean).

The attributes that all parameters have in common are:

- *value*: parameter value in string format
- *description*: string with a brief description
- *name*: string with the unique parameter name
- *dimensions*: an array of integers representing the value dimensions (rows x columns)
- *visibility*: integer whose purpose is to provide a mechanism to hide some parameters to users depending on the user-role (admin, operator, etc…).

6.4.2.1.1.4. Dependencies

This class is related with the XML parsing as it represents an entity extracted from a XML configuration file.

6.4.2.1.1.5. Interfaces

The public interface declared by this class is as follows:

Table 21: list of Parameter class public operations

Operation name	Input	Output	Description
`getParamValue`	paramName	Array of values	This method retrieves the list of parameter values.
`getDefaultValue`	paramName	Array of values	This method retrieves the list of parameter default values. If not default value set returns a void set of Objects.
`checkValidity`	paramName	Boolean	This method checks the validity of a parameter in terms of value consistency (min, max, type)

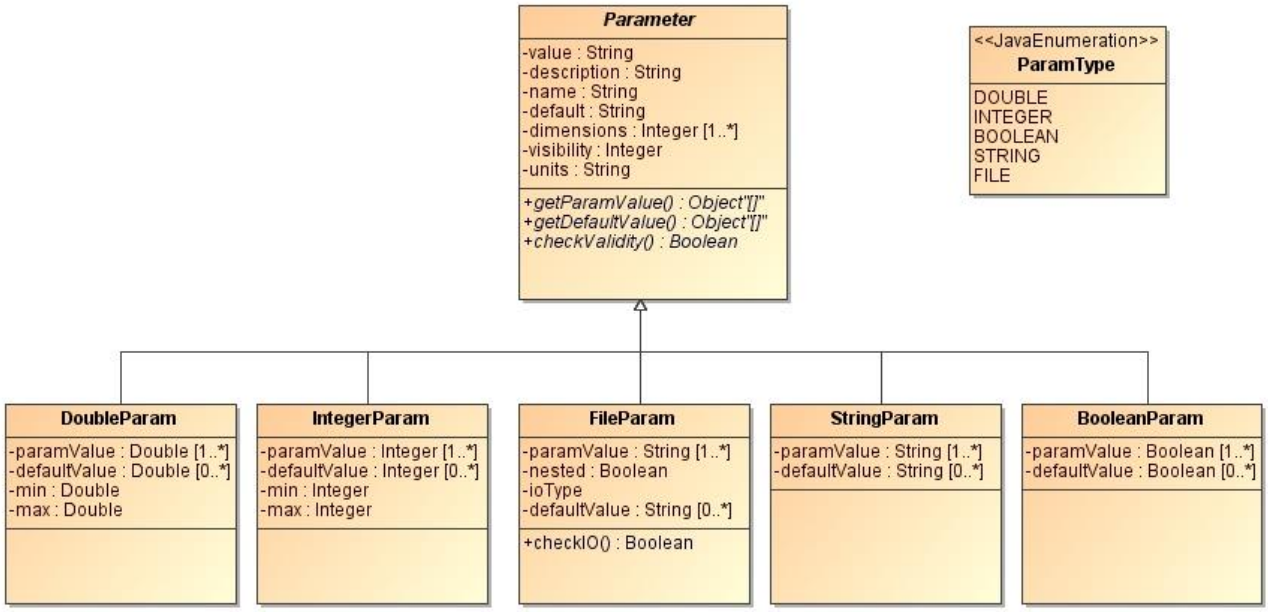

Figure 6-4: openSFpms.parameter class diagram

6.4.2.1.2. RulesInterface

6.4.2.1.2.1. Type

This component is an interface.

6.4.2.1.2.2. Purpose

This interface provides an abstract bridge for the controller package in order to access *Rules* methods.

6.4.2.1.2.3. Function

The functionality of this interface is give a common API for accessing to the different rules present on the system.

6.4.2.1.2.4. Dependencies

This interface depends on *rules* and *parameters* packages as it relates both of them.

6.4.2.1.2.5. Interfaces

The public methods of this interface are the followings:

Table 22: list of RulesInterface public operations

Operation name	Input	Output	Description
`getNamespace`	ruleID	Namespace	Returns the namespace where this rule can be applied.
`checkIntegrity`	ruleID	Boolean	This method checks if the Rule syntax is correct.

Operation name	Input	Output	Description
`getErrorMessage`	ruleID	String	Gets a string with a message in order to be shown in the error terminal.
`applyRule`	ruleID	Boolean	Applies a rule/constraint to a set of parameter values. In case the values are compliant returns *true*, *false* otherwise.

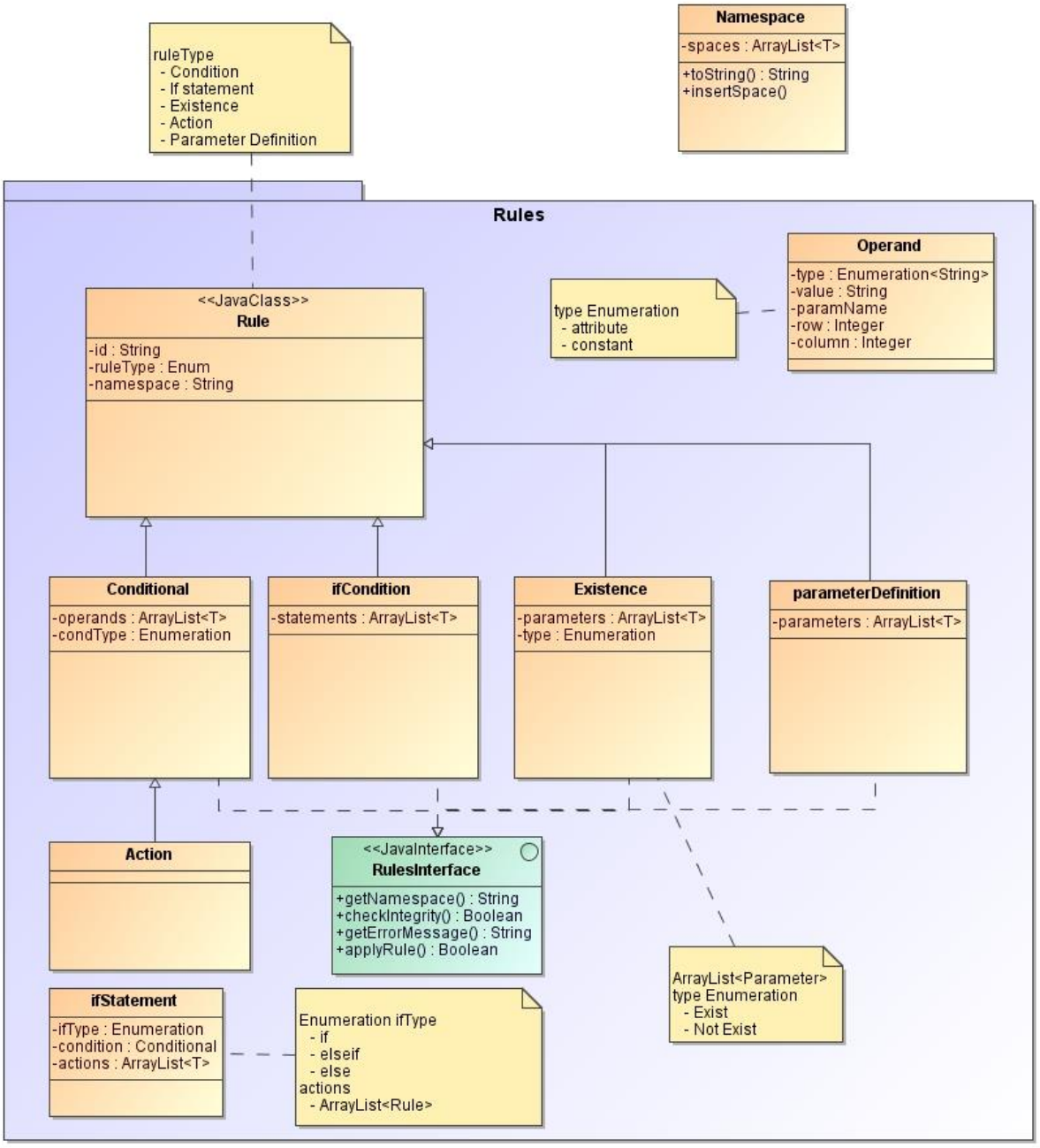

Figure 6-5: openSFpms.domain.rules class diagram

6.4.2.2. manager

This package contains the manager classes. These classes are meant to manage, and control the sets of domain objects (elements). Thus, they shall be in charge of managing the different kinds of elements in the system: namespaces, parameters, rules etc…

This package also contains the classes used for connecting the view and domain packages:

- To prepare the appearance of every action performed by users, as presenting forms, tables, trees, etc and also menus and buttons to let the user triggers the operations.
- To access the *domain* package to get data needed. This represents the connection to the domain layer presenting or providing the information managed by this part of the application.

The manager package is composed of five sub-packages:

- ***sessionMgr*:** this package contains the classes corresponding to the Parameter Editor graphical interface.
- ***rulesManager*:** this package contains the classes corresponding to the Rules Editor graphical interface.
- ***paramManager:*** contains the classes corresponding to views used by the previously mentioned before (file selection, specific renderers etc…)
- ***fileHandling:*** contains the classes corresponding to views used by the previously mentioned before (file selection, specific renderers etc…)
- ***XMLparser:*** contains the classes corresponding to views used by the previously mentioned before (file selection, specific renderers etc…)

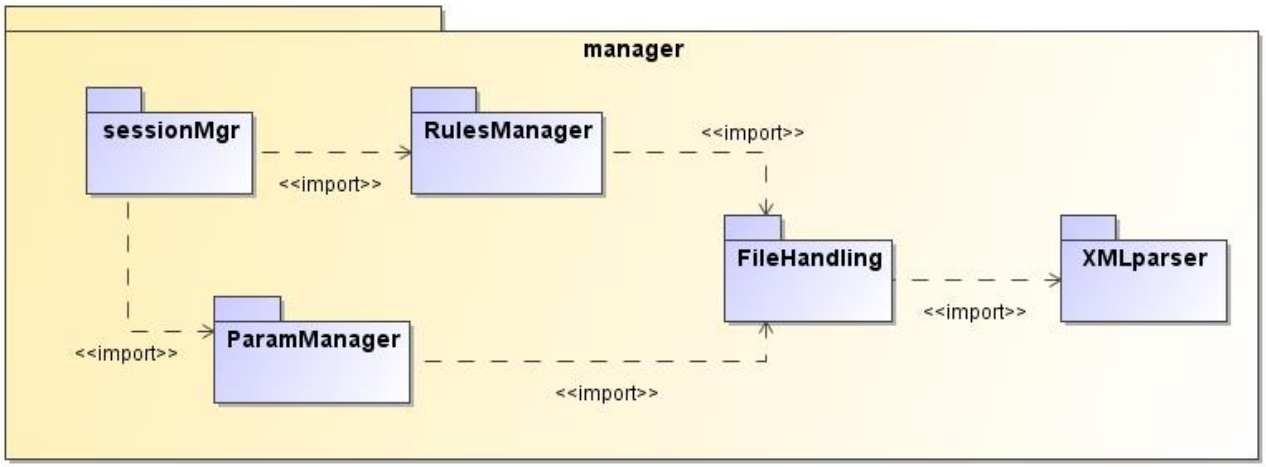

Figure 6-6: openSFpms.manager packages diagram

6.4.2.3. view

This package groups the classes related to the appearance of the user machine interaction.

This *view* package is, in turn, composed of three packages:

- ***parameterEditor*:** this package contains the classes corresponding to the Parameter Editor graphical interface.

- ***rulesEditor*:** this package contains the classes corresponding to the Rules Editor graphical interface.
- ***auxiliaryViews*:** contains the classes corresponding to views used by the previously mentioned before (file selection, specific renderers etc…)

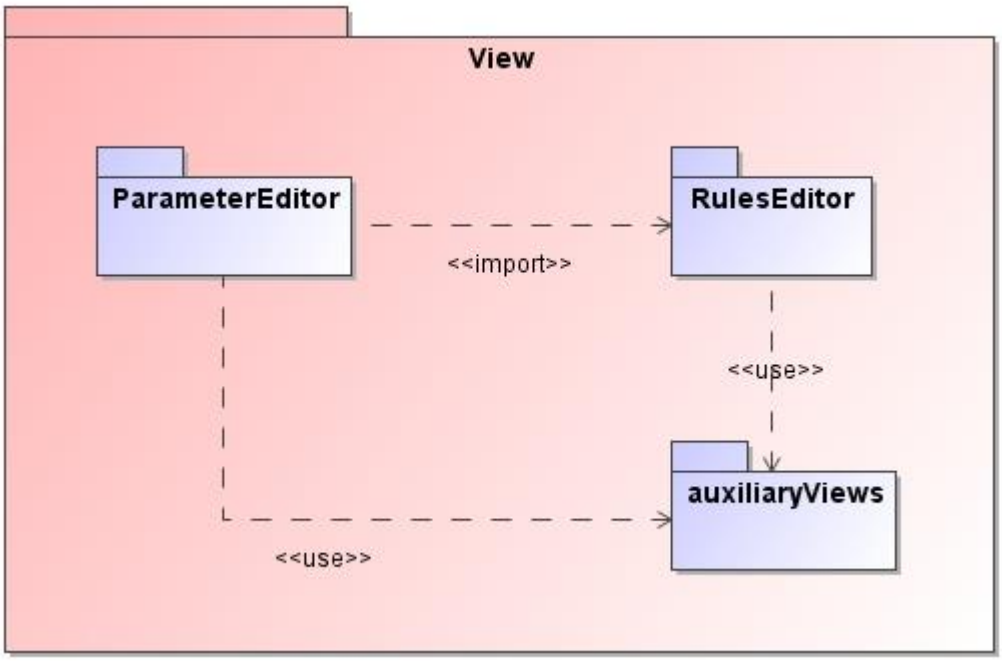

Figure 6-7: openSFpms.view packages diagram

Every package and classes inside the *view* package uses intensively the *java.awt* and *javax.swing* packages, whose descriptions are outside the scope of this document but can be found in [RD SWING].

This *view* package is accessed by the *manager* package in order to access domain elements attributes and interact with them.

6.4.3. GUI design

In this section will be shown some draft interfaces for the openSF parameter management system that as mentioned before is composed of a Rule Editor and a Parameter Editor interface.

The standards used for this application GUI design are the same ones used for the openSF system. Figure 6-8 and Figure 6-9 show a draft of the graphical interfaces for Parameter and Rule editor respectively.

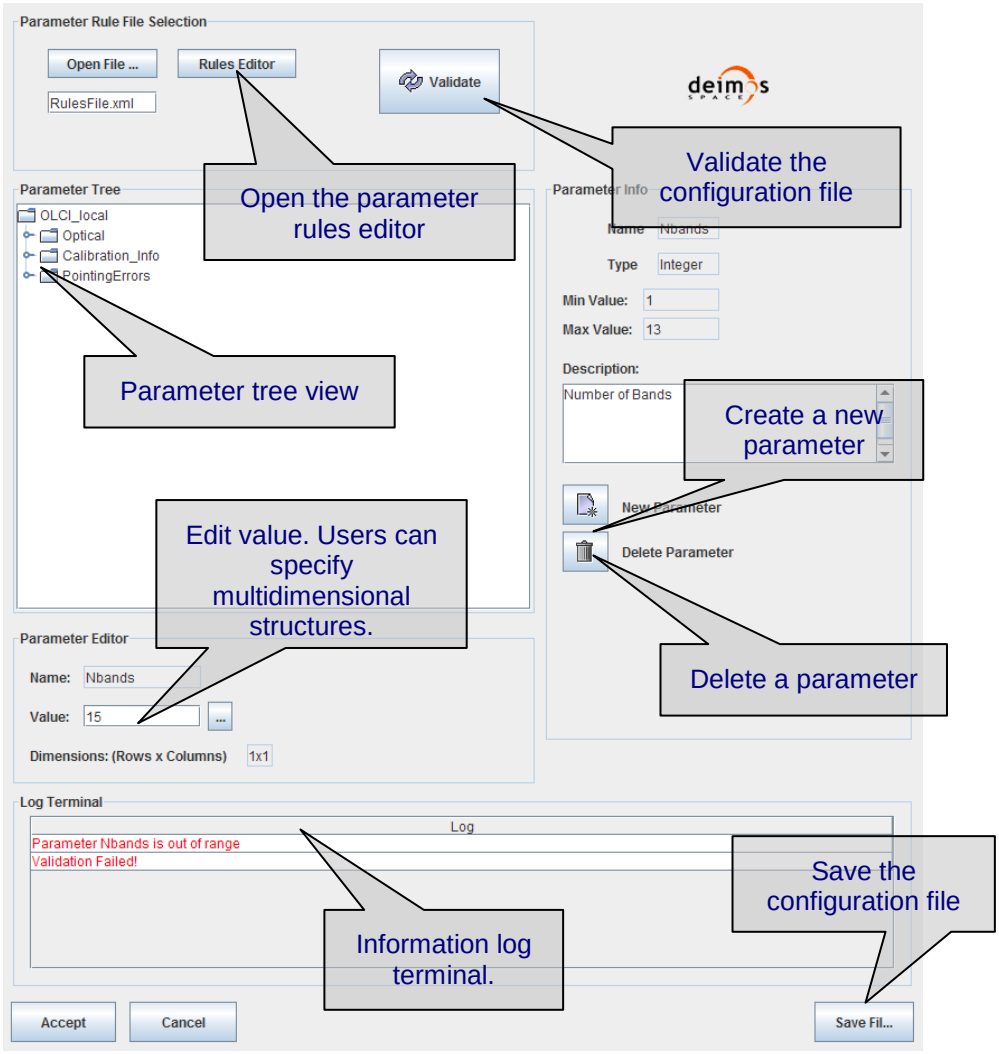

Figure 6-8: Parameter Editor draft interface

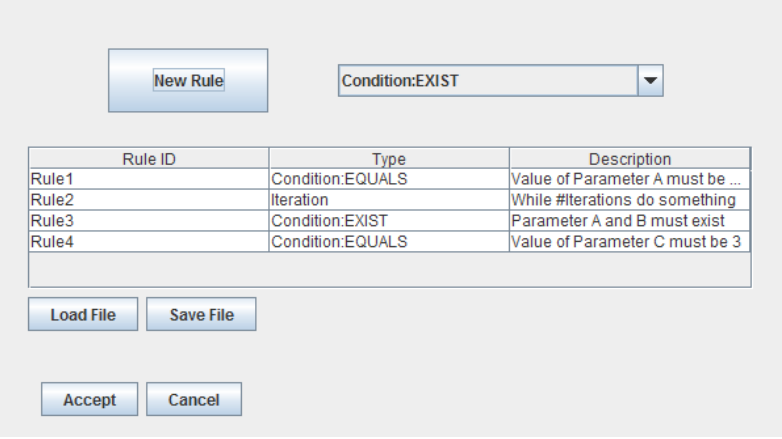

Figure 6-9: Rule Editor draft interface

7. OSFI – OPENSF INTEGRATION LIBRARIES

This section presents the architecture of the openSF integration libraries.

- Overview of the integration libraries
- Integration libraries design
- Architecture of the OSFI adaptation for other programming languages. This section will also show what is the interaction mechanism between OSFI, openSF, Matlab and IDL.

7.1. Introduction

The Open Simulation Framework Integration Libraries (OSFI from now on) will be used to ease the integration of models into the open Simulation Framework.

The main goals of the integration libraries:

- Solve the interfacing issues between models and openSF
- Minimize the model code intrusion, providing native libraries for each of the supported programming languages

The Integration Libraries activity provides model developer with a set of routines with a well-defined public interface hiding the implementation details.

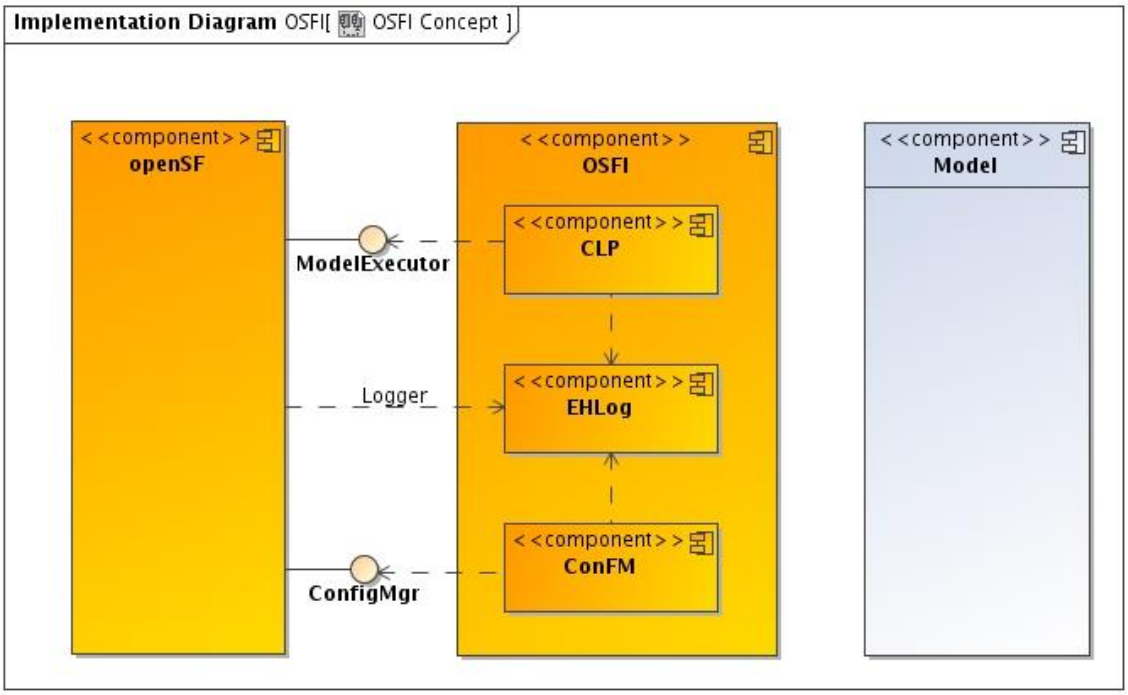

Figure 7-1: OSFI Integration with openSF

7.2. Integration Libraries Design

This section gives a brief description of the method used for the architectural design, the Unified Modeling Language used for formal diagrams and the system background and context.

OSFI libraries are decomposed in the following packages also shown in Figure 7-2:

- **CLP**: containing all classes correspondent to models command line interface
- **EHLog**: logging and openSF communication module
- **ConFM**: responsible of configuration files parsing and simulation parameters retrieval.

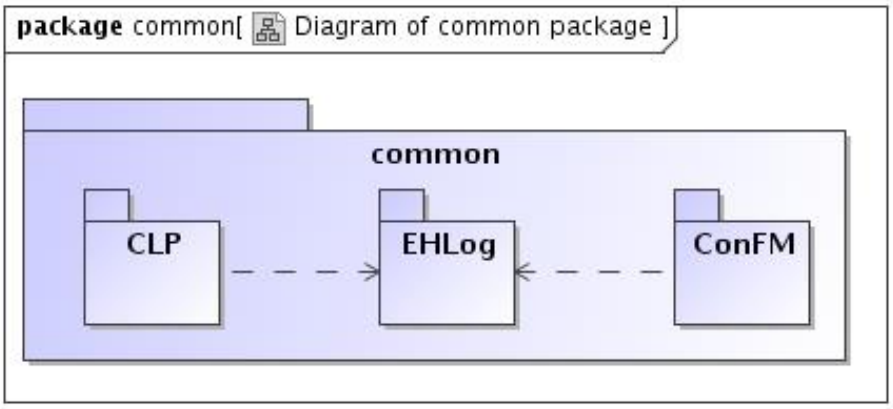

Figure 7-2: OSFI common packages

OSFI design details described below these lines correspond to the C++ implementation; other programming language issues are listed in section 7.3.

7.2.1. CLP

CLP stands for Command Line Parsing and is the software module in charge of taking model command line arguments and parse them providing a set of routines to easily access them. This module also checks that the command line used to invoke the model is compliant with [AD-ICD].

Figure 7-3 shows the CLP class diagram, listing interface methods.

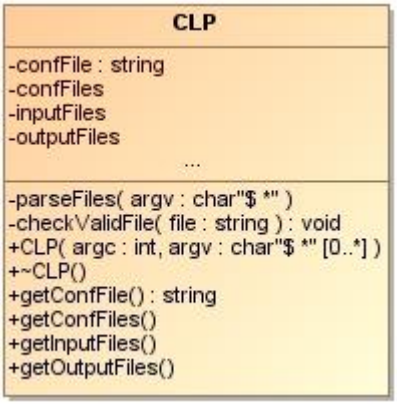

Figure 7-3: CLP class diagram

7.2.2. EHLog

EHLog stands for error handler and logging module, and is the package responsible of formatting log and error messages following the convention specified in [AD-ICD].

This module core class is *Logger* whose diagram is shown in Figure 7-4.

Figure 7-4: Logger class diagram

7.2.3. ConFM

This package is the responsible of parsing/reading the XML configuration files, store parameters and present them to the model.

ConFM module makes use of the Apache XercesC++ that is a validating XML parser for generating, manipulating, and validating XML documents using the DOM, SAX, and SAX2 APIs.

Classes contained in this package are:

- *XMLparser:* XercesC adaptation class that provides the foundations to read and parse an XML file.
- *ParamReader:* class that inherits from XMLparser providing functions to read XML files and retrieve configuration
- *Parameter:* object representation of a simulation parameter, encapsulating all parameter attributes such as, name, value, type, dimensions etc…

Additionally this module keeps track of model configuration parameters in a map like container that identifies a parameter name with the correspondent object.

This package contains also extra classes that support some ConFM functionalities:

- *TreeErrorReporter*: for raising errors coming from the XML DOM parsing.
- *DynamicArray*: implementation of a generic and dynamic bi-dimensional matrix.

Figure 7-5 shows class diagram of the ConFM package.

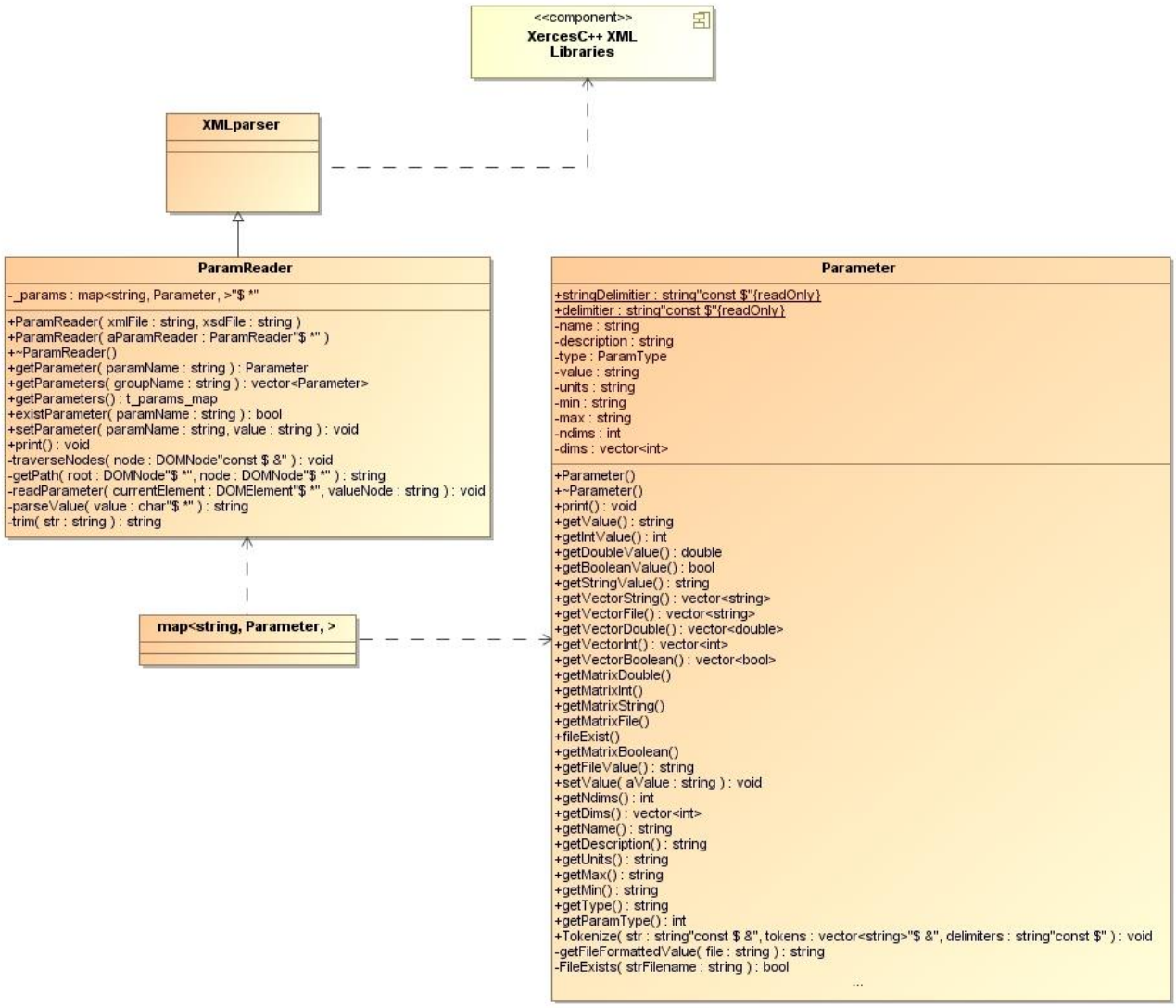

Figure 7-5: ConFM class diagram

7.3. OSFI Other programming languages

This section shows the design of OSFI specific components aimed to support models written in other programming languages.

7.3.1. OSFI wrappers - C, Fortran 90 and Fortran 77

OSFI support for other programming languages is based in the implementation of a wrapper over OSFI C++ core libraries. This design implies the implementation of an adaptation layer that faces the interface issues between C++ and the other programming languages.

Currently the languages supported by OSFI are C, F90 and F77, listed below some of the interface problems found for each programming languages.

- C: C++ string vs. char* C arrays, no native boolean type in C
- Fortran 90 and Fortran 77: array rows and columns, string handling

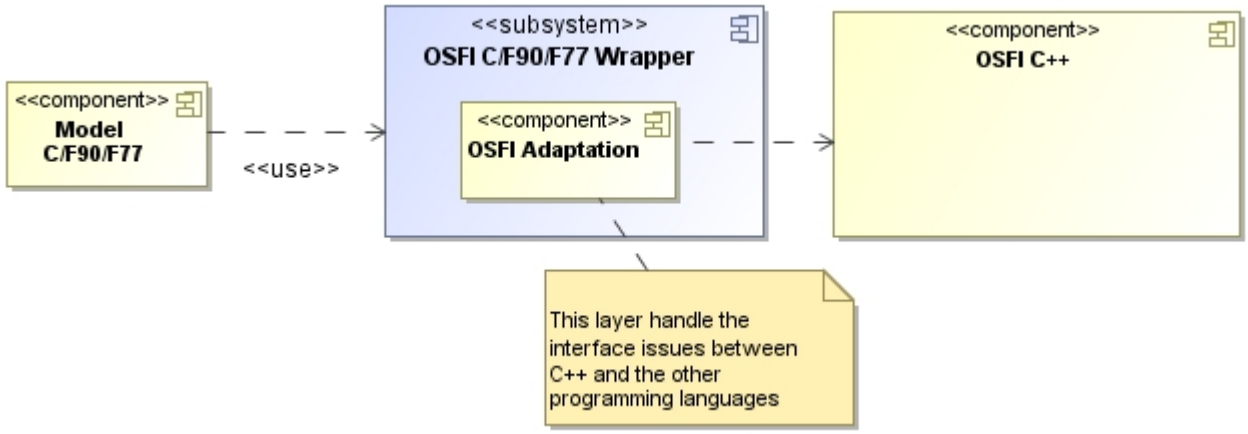

Figure 7-6: OSFI wrapper, implementation diagram

7.3.2. OSFI Matlab

The OSFI Matlab component is composed by a group of Matlab code files (.m) following the same philosophy as the one used for core C++ libraries.

It has been decided to implement Matlab code files instead of a wrapper due to the following reasons:

- Matlab programming requires less coding effort than any other mechanism (mex functions etc…).
- Ease the use of the libraries in the model developer side. See [RD-OSFI-DM]
- Allow model developers to test the code from Matlab Workbench.
- OS architecture independency.

The main drawback of this implementation decision is that the maintenance is tougher as a change in OSFI core libraries implies also a change in OSFI Matlab libraries.

OSFI Matlab uses the object oriented programming approach introduced in Matlab since version R2008a. This fact implies that OSFI libraries do not support previous Matlab versions. Large documentation support about OOP in Matlab can be found in the web.

Classes within OSFI Matlab libraries are:

- *Logger.m*: in charge of openSF log message and communication interface
- *ConFM.m*: XML parsing and parameter retrieval
- *Parameter.m*: encapsulates parameter attributes and functionalities
- *CLP.m*: command line arguments, configuration, input and output files

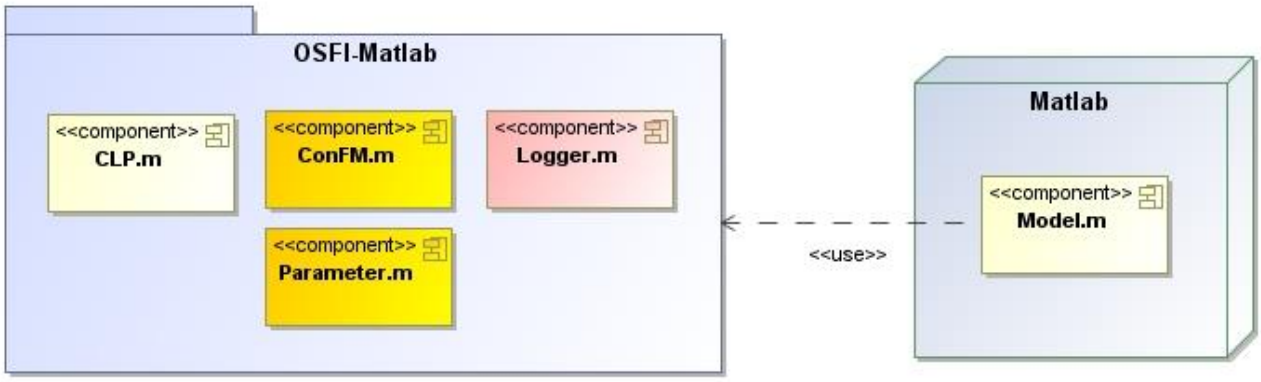

Figure 7-7: OSFI Matlab Implementation diagram

7.3.2.1. OpenSF Integration: Executing Matlab models

The mechanism used to call Matlab models from the openSF HMI (*SessionMgr*) is the one provided by Mathworks to execute scripts in batch mode from the command line.

The openSF and Matlab interaction method is as follows:

1. A Matlab script file is provided as model binary.
2. openSF gets OS runtime.
3. openSF calls Matlab in batch mode through the OS runtime passing as argument the model binary and the required configuration, input and output files

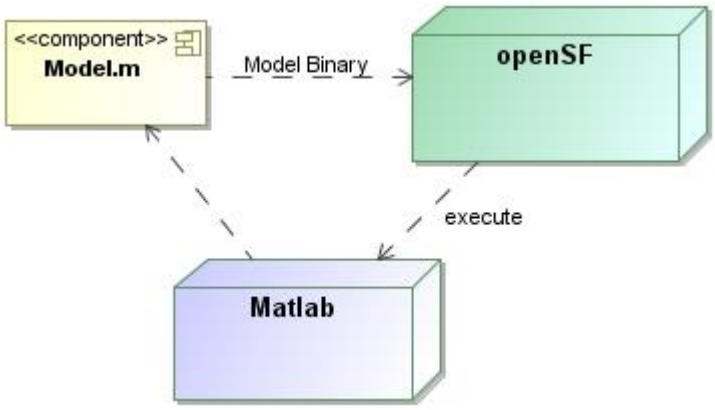

Figure 7-8: Matlab Model Execution

7.3.3. OSFI IDL

The architecture of the OSFI IDL libraries is conceptually the same as the one used for OSFI C++ and Matlab libraries.

The following IDL source code files (.pro) compose OSFI IDL libraries:

- *Logger.pro*: in charge of openSF log message and communication interface
- *ConFM.pro*: XML parsing and parameter retrieval
- *Parameter.pro*: encapsulates parameter attributes and functionalities

- *CLP.pro*: command line arguments, configuration, input and output files

In the IDL case a new component is introduced in order to allow openSF users to directly execute IDL source code files from the framework. This component is the *idl-model* coded in C++ and that makes use of the *Callable IDL* mechanism that basically is an interface to execute IDL commands from a code written in Java, C or C++.

7.3.3.1. idl-model

This software component is a C++ executable that is in charge of compiling IDL source code passed as argument (model) and the OSFI IDL libraries.

As result this executable is a layer that makes transparent to the user the process of compiling IDL code.

A major disadvantage of this mechanism is that it is not OS architecture independent and needs to be re-compiled for each IDL version and target computer platform.

Idl-Model component is not necessary when the IDL source code (OSFI components + model) has been previously compiled and saved in .SAV format.

For more information about *Callable IDL* please visit the ITT IDL homepage and read the IDL External Development Guide.

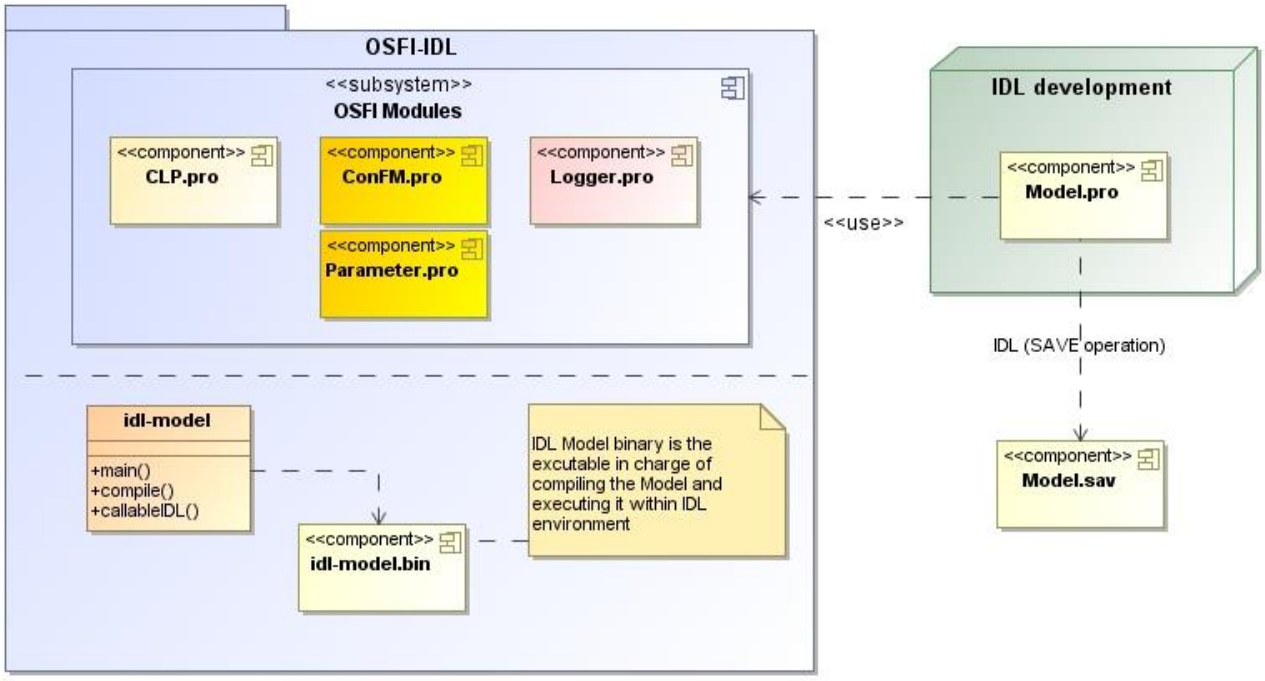

Figure 7-9: OSFI IDL implementation diagram

7.3.3.2. OpenSF Integration: Executing IDL models

- IDL saved file execution (.SAV)

1. IDL saved file is specified as model binary
2. openSF executes IDL runtime through OS runtime
3. IDL saved file is passed as argument to the IDL runtime
4. Configuration, input and output files shall be specified through environment variables as IDL runtime does not allow command line arguments.

- IDL source file execution (.PRO)

1. IDL source file is specified as model binary
2. openSF executes *idl-model.bin* passing as argument the IDL .pro file and configuration, input and output files.
3. *idl-model* compiles model source file and OSFI libraries and execute as it would be done through IDL Workbench.

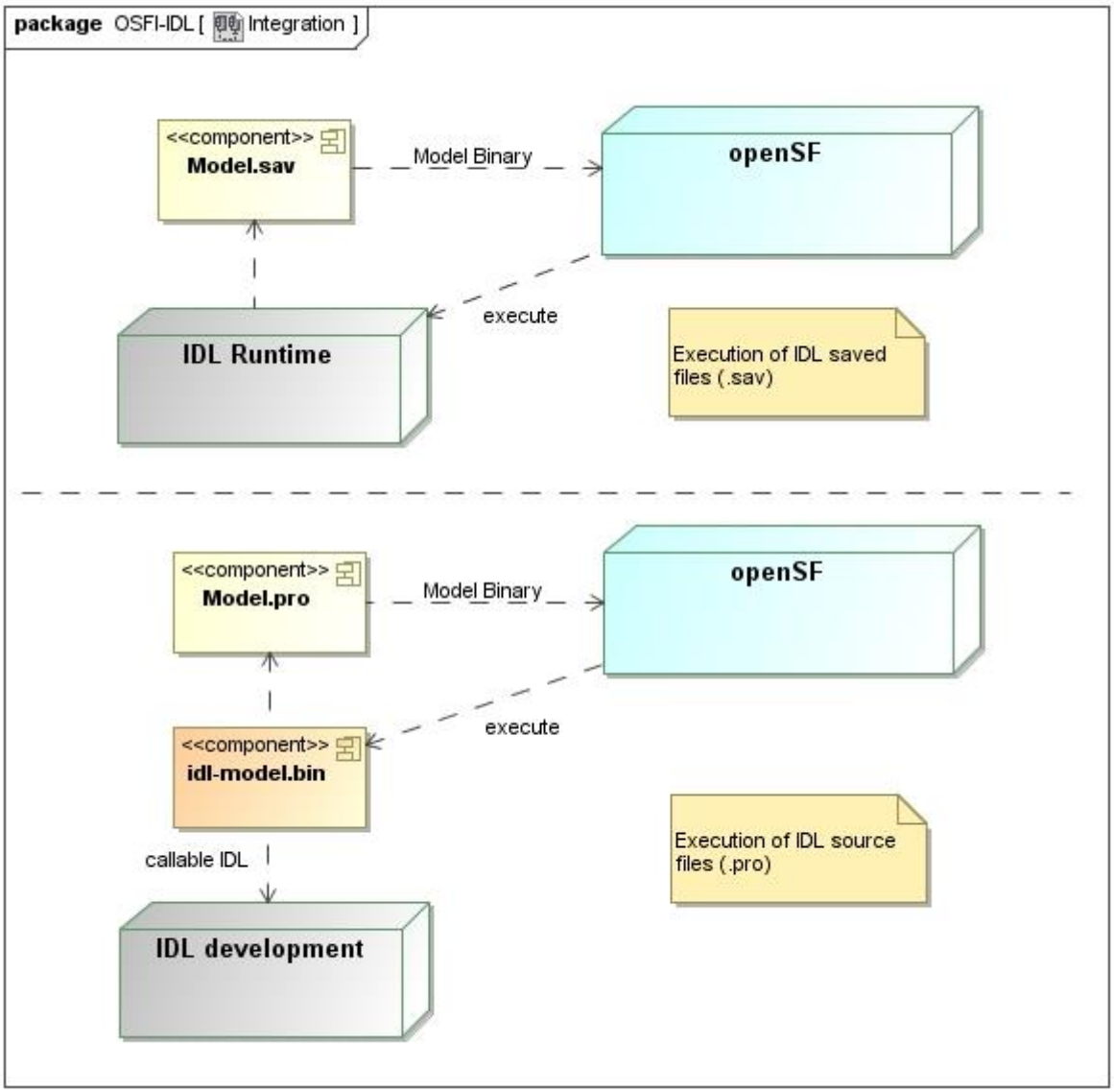

Figure 7-10: IDL model execution modes.

8. OSFEG – OPENSF ERROR GENERATION LIBRARIES

This section presents the architecture of the openSF error generation libraries.

- Overview of the error generation libraries;
- Description of the libraries architecture;
- Specification of the libraries error functions;
- Error generation libraries design.

8.1. Error generation libraries overview

The openSF tool allows users to integrate and execute pieces of code, «models» that form the building blocks of a simulation process. Typically those pieces of code, «models» are handled by openSF as simple executable programs with three interfaces, input, output and configuration.

Under this scenario appears the goal of performing a statistical analysis of the E2E simulator driven by the errors and perturbations present in the parameters involved in a simulation chain.

The Open Simulation Framework Error Generation Libraries (OSFEG from now on) will be used as a tool to ease the mathematical modeling of a perturbation within statistical analysis scenarios. OSFEG offers to developers a well-documented interface to ease the modeling and generation of a perturbation over desired parameters. The libraries provide an error-modeling interface based on a XML file definition and its correspondent implementation in C++.

8.2. Error generation libraries architecture

OSFEG comes in different distributions depending on the needs of the user:

- Source package, including necessary sources in the supported language, for including and compiling with other sources;
- Binary package, including headers and static/dynamic libraries for linking with other sources. There is a version of this package for the supported target machine and Linux Operating System.

Figure 8-1 shows a high-level view of the contents of OSFEG distributions.

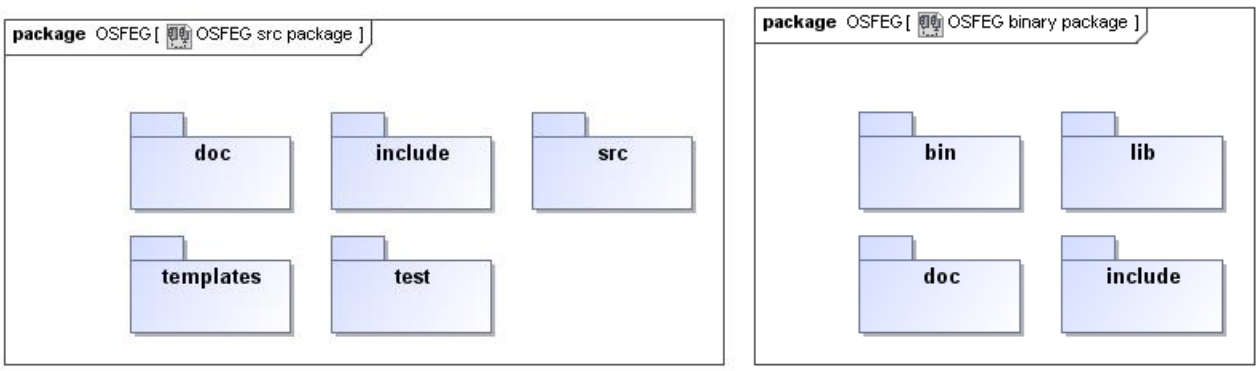

Figure 8-1: OSFEG deployment

8.3. OpenSF error generation libraries specification

This section describes the mathematical functions implemented within the error generation libraries. The libraries include the most used analytical and random functions for parameter perturbation in E2E simulation modeling scenarios.

8.3.1. Error definition files

The parameter perturbation functions are defined through an XML file. Details regarding the format of the XML file can be found in [AD-ICD].

8.3.2. Error Functions

For a detailed description of the error functions and the variables involved in the function definitions please refer to the [AD-SRD].

An example of an error definition file implementing all the binary and some composite operations can be found in the [AD-ICD].

8.4. Error Generation Libraries Design

This section gives a brief description of the method used for the architectural design and the Unified Modeling Language used for formal diagrams.

OSFEG libraries are decomposed in a set of classes shown in Figure 8-2:

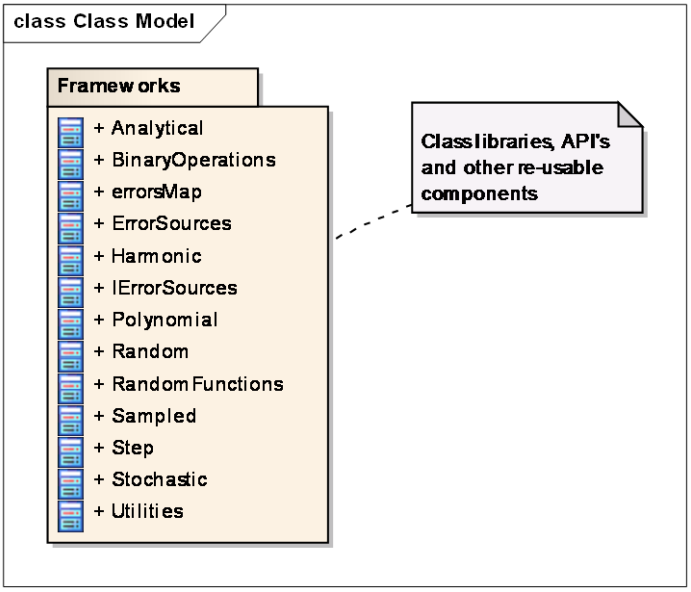

Figure 8-2: OSFEG main packages

OSFEG design details described below corresponds to the C++ implementation of the libraries.

8.4.1. ErrorSources

ErrorSources class is the main entry point for error generation. This class is the responsible of parsing/reading the XML error definition files and present them to the error generation functions.

This module makes use of the Apache XercesC++ that is a validating XML parser for generating, manipulating, and validating XML documents using the DOM, SAX, and SAX2 APIs (refer to [AD-ICD] for details on the XML file format).

Figure 8-3 shows the Error Sources class diagram, listing interface methods.

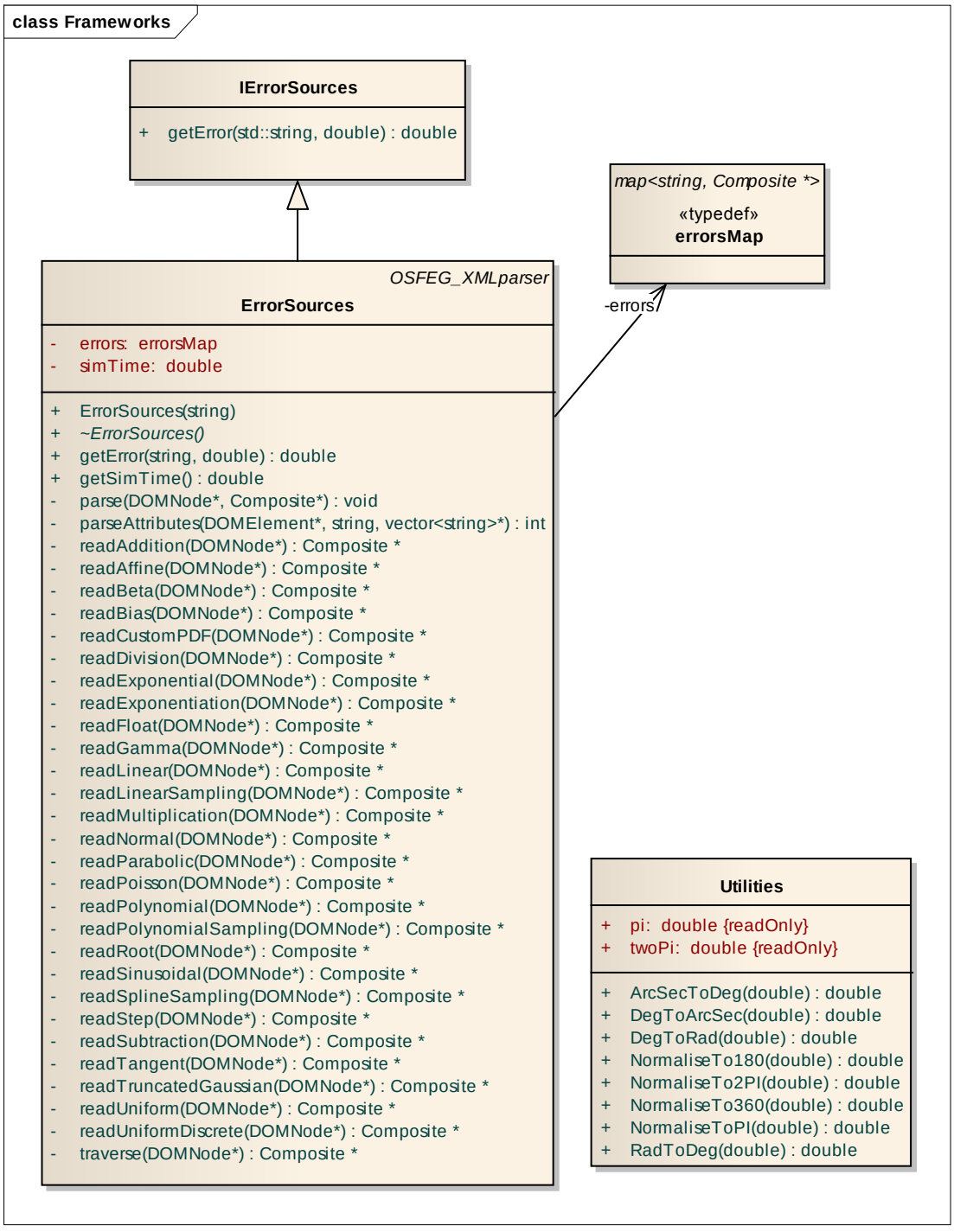

Figure 8-3: ErrorSources class diagram

8.4.2. Error Functions

The several available error functions correspond to a given class which can be used to generate the intended values. There are two types of functions corresponding to two class hierarchies:

- *Analytical:* hierarchy representing functions with a given mathematical formula (see Figure 8-4);
- *Random:* hierarchy representing functions with an associated random factor (see Figure 8-5).

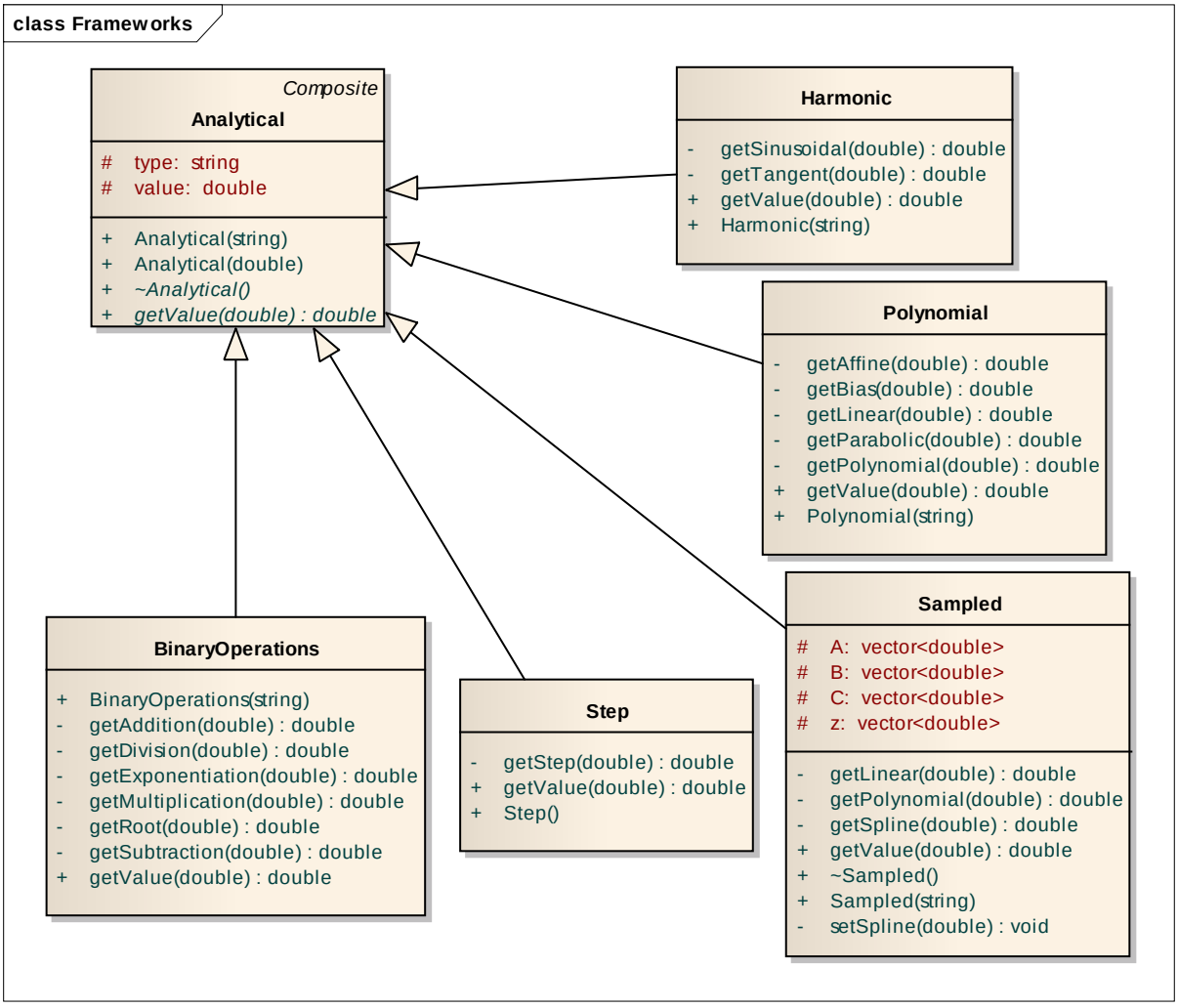

Figure 8-4: Analytical hierarchy class diagram

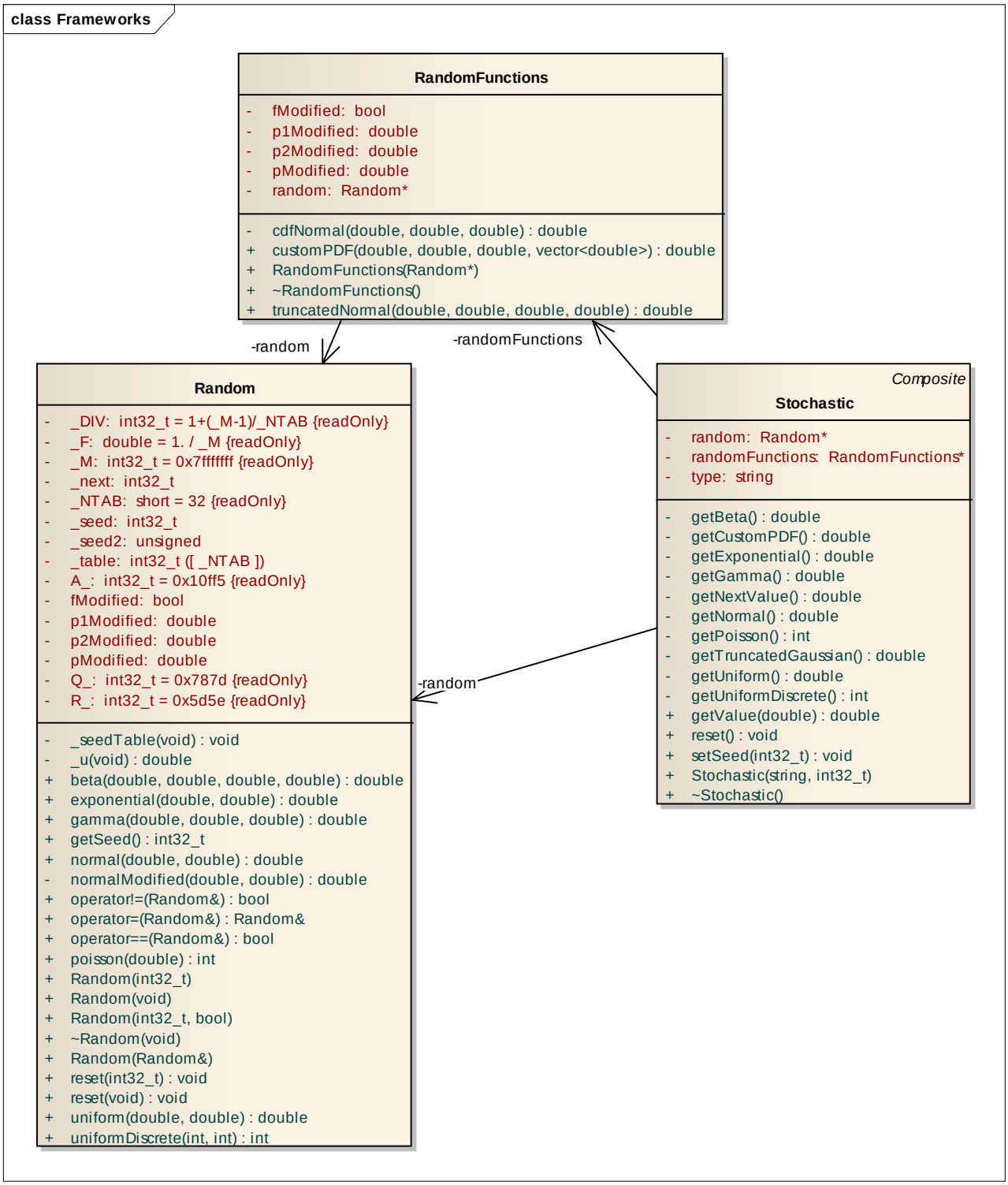

Figure 8-5: RandomFunctions hierarchy class diagram

9. TRACEABILITY MATRIXES

In this section we include traceability matrixes relating *System Requirements to System Components.*

9.1. Direct Traceability

In this section we include a traceability matrix from System Requirements to System Components. Comments are added to further clarify the reasons for the assignments, as well as to explain any limitation or constraint.

Those requirements marked with an "R" at the "Design component" column are validated by review and therefore, do not have to be traced.

Table 23: Direct Traceability Table

System Req.	Design component	Comments
SR-FUN-0010	*Model,Session,Simulation,SessionExecutor, ModelChainExecutor*	
SR-FUN-0020	*Session,SessionMgr, SessionView, SessionDC*	
SR-FUN-0030	*Session,ToolMgr,ToolDC*	
SR-FUN-0040	*HelpView*	
SR-FUN-0050	R	
SR-FUN-0060	*Model,ModelView,ModelDC,ModelMgr*	
SR-FUN-0070	*Model,ModelView,ModelDC,ModelMgr*	
SR-FUN-0080	*Model,ModelView,ModelDC,ModelMgr*	
SR-FUN-0090	*Model,ModelView,ModelDC,ModelMgr*	
SR-FUN-0100	*DELETED*	
SR-FUN-0110	*DELETED*	
SR-FUN-0120	*DELETED*	
SR-FUN-0130	*Descriptor,DescriptorMgr,DescriptorView,DescriptorDC*	
SR-FUN-0140	*Descriptor,DescriptorMgr,DescriptorView,DescriptorDC*	
SR-FUN-0150	*Descriptor,DescriptorMgr,DescriptorView,DescriptorDC,*	
SR-FUN-0160	*Descriptor,DescriptorMgr,DescriptorView DescriptorDC,*	
SR-FUN-0170	*Descriptor,DescriptorMgr,DescriptorDC DescriptorDC,Database,*	
SR-FUN-0180	*Descriptor,DescriptorMgr,DescriptorDC, DescriptorView*	
SR-FUN-0190	*Descriptor,DescriptorMgr,DescriptorDC,Database*	
SR-FUN-0200	*Model,ModelMgr*	
SR-FUN-0210	*Model,ModelMgr,Database*	
SR-FUN-0220	*ModelMgr,ModelDC,ModelView*	

System Req.	Design component	Comments
SR-FUN-0230	*ModelMgr,ModelDC,ModelView*	
SR-FUN-0240	*ModelMgr,ModelDC,ModelView*	
SR-FUN-0250	*ModelMgr,Database*	
SR-FUN-0260	*Model,ModelMgr*	
SR-FUN-0270	*Model,ModelMgr*	
SR-FUN-0280	*Model,ModelMgr,ModelDC,ModelView*	
SR-FUN-0290	*Model,ModelMgr,ModelDC,ModelView, Database*	
SR-FUN-0300	*Model,ModelMgr,Database*	
SR-FUN-0310	*Model,ModelMgr,ModelDC,ModelView*	
SR-FUN-0320	*Model,ModelMgr,ModelDC,ModelView*	
SR-FUN-0330	*Model,ModelMgr,Database*	
SR-FUN-0340	*Stage,StageMgr,StageDC,StageView*	
SR-FUN-0350	*Stage,StageMgr,StageDC,StageView*	
SR-FUN-0360	*Stage,StageMgr,StageDC,StageView*	
SR-FUN-0370	*Stage,StageMgr,StageDC,StageView*	
SR-FUN-0380	*Stage,StageMgr,Database*	
SR-FUN-0390	*Stage,StageMgr,StageDC,StageView*	
SR-FUN-0400	*Stage,StageMgr,Database*	
SR-FUN-0410	*Stage,StageMgr,StageDC,StageView*	
SR-FUN-0420	*Stage,StageMgr,Database*	
SR-FUN-0430	*Sim,SimMgr,SimDC,SimView*	
SR-FUN-0440	*Sim,SimMgr,SimDC,SimView*	
SR-FUN-0450	*Sim,SimMgr,SimDC,SimView*	
SR-FUN-0460	*Sim,SimMgr,SimDC,SimView,DescriptorMgr,ModelMgr*	
SR-FUN-0470	*Sim,SimMgr,SimDC,SimView,StageMgr*	
SR-FUN-0480	*Sim,SimMgr,SimDC,SimView,ModelMgr*	
SR-FUN-0490	*Sim,SimMgr,Database*	
SR-FUN-0500	*Sim,SimMgr,Database*	
SR-FUN-0510	*Sim,SimMgr,DescriptorMgr,ModelMgr,Database*	
SR-FUN-0520	*Sim,SimMgr,SimDC,SimView*	
SR-FUN-0530	*Sim,SimMgr,SimDC,SimView*	
SR-FUN-0540	*Sim,SimMgr,SimDC,SimView*	
SR-FUN-0550	*Sim,SimMgr,Database*	
SR-FUN-0560	*Sim,SimMgr,SimDC,SimView*	
SR-FUN-0570	*Sim,SimMgr,SimDC,SimView*	
SR-FUN-0580	*Sim,SimMgr,Managers,Database*	
SR-FUN-0590	*Session,SessionMgr,SessionDC,SessionView*	
SR-FUN-0600	*Session,SessionMgr,SessionDC,SessionView*	

System Req.	Design component	Comments
SR-FUN-0610	*Session,SessionMgr,SessionDC,SessionView*	
SR-FUN-0620	*Session,SessionMgr,SessionDC,SessionView*	
SR-FUN-0630	*Session,SessionMgr,Database*	
SR-FUN-0640	*Session,SessionMgr*	
SR-FUN-0650	*Session,SessionMgr,Database*	
SR-FUN-0660	*Session,SessionMgr*	
SR-FUN-0670	*Session,SessionMgr,SessionDC,SessionView*	
SR-FUN-0680	*Session,SessionMgr,SessionDC,SessionView*	
SR-FUN-0690	*Session,SessionMgr,SessionDC,SessionView*	
SR-FUN-0700	*Session,SessionMgr,SessionDC,SessionView*	
SR-FUN-0710	*SessionView,Parameter*	
SR-FUN-0720	*Session,SessionMgr,Database*	
SR-FUN-0730	*Session,SessionMgr,SessionDC,SessionView*	
SR-FUN-0740	*Session,SessionMgr,SessionDC,SessionView*	
SR-FUN-0750	*Session,SessionMgr,Managers,Database*	
SR-FUN-0760	*Session,SessionMgr,SessionDC,SessionView,ToolMgr*	
SR-FUN-0770	*Session,SessionMgr,SessionDC,SessionView*	
SR-FUN-0780	*SessionDC,SessionView,ModelMgr,Parameter, pms.Parameter,pms.Rule*	
SR-FUN-0790	*SessionDC,SessionView,ModelMgr,Parameter, pms.Parameter,pms.Rule*	
SR-FUN-0800	*SessionView, pms.ParameterEditor*	
SR-FUN-0810	*SessionView, pms.ParameterEditor*	
SR-FUN-0820	*Session,SessionMgr,SessionDC,SessionView*	
SR-FUN-0830	*SessionMgr,SessionExecutor*	
SR-FUN-0840	*SessionMgr,SessionExecutor,ModelChainExecutor, SessionView*	
SR-FUN-0850	*RepositoryView,RepositoryDC,SessionExecutor*	
SR-FUN-0860	*SessionMgr,SessionExecutor,ModelChainExecutor*	
SR-FUN-0870	*SessionMgr*	
SR-FUN-0880	*Logger,SessionView*	
SR-FUN-0890	*SessionMgr,SessionView,ModelChainExecutor, SessionExecutor*	
SR-FUN-0900	*SessionView,DescriptorMgr,IO*	
SR-FUN-0910	*SessionView*	
SR-FUN-0920	*SessionMgr,ModelChainExecutor,SessionExecutor*	
SR-FUN-0930	*SessionMgr,ModelChainExecutor,SessionExecutor, SessionView*	
SR-FUN-0940	*Tool,ToolMgr,SessionMgr,SessionDC*	

System Req.	Design component	Comments
SR-FUN-0950	*RepositoryDC,SessionMgr,SessionDC, SessionView*	
SR-FUN-0960	*SessionView,SessionDC*	
SR-FUN-0970	*SessionView,Logger,SessionDC*	
SR-FUN-0980	*SessionView*	
SR-FUN-0990	*Tool,ToolMgr,ToolDC,SessionMgr*	
SR-FUN-1000	*Tool,ToolMgr,ToolDC,SessionMgr*	
SR-FUN-1010	*SessionView,Logger*	
SR-FUN-1020	*Logger*	
SR-FUN-1100	*ConnectionControl,Database,DatabaseIF DBServer*	
SR-FUN-1110	*ConnectionControl,Database,DatabaseIF DBServer*	
SR-FUN-1120	*pms.Parameter,pms.Rule*	
SR-FUN-1130	*pms.Parameter, pms.Rule, pms.ParameterEditor pms.RulesEditor*	
SR-FUN-1140	*pms.ParameterEditor*	
SR-FUN-1150	*pms.ParameterEditor*	
SR-FUN-1160	*pms.Parameter, pms.Rule, pms.ParameterEditor, pms.RulesEditor*	
SR-FUN-1170	*pms.ParameterEditor*	
SR-FUN-1345/3.0	*Logger*	
SR-FUN-1450/3.0	*ParallelScheduler {ParallelEventManager, ExecutionModelSet, ModelExecutionThreadMgr, ModelExecutionThread, ModelExecutionThreadSet, ThreadPool}*	
SR-FUN-1460/3.0	*ParallelScheduler {ParallelEventManager, ExecutionModelSet, ModelExecutionThreadMgr, ModelExecutionThread, ModelExecutionThreadSet, ThreadPool}*	
SR-FUN-1470/3.0	*ParallelScheduler, SessionExecutor*	
SR-FUN-1480/3.0	*Controller, ModelChainExecutor*	
SR-FUN-1490/3.0	*Controller, ModelChainExecutor*	
SR-FUN-1500/3.0	*Controller, ModelChainExecutor*	
SR-FUN-1510/3.0	*Controller, ModelChainExecutor*	
SR-FUN-1520/3.0	*Controller, ModelChainExecutor*	
SR-FUN-1530/3.0	*Controller, ModelChainExecutor*	
SR-FUN-1540/3.0	*ModelMgr, ModelDC, ModelView, ConnectionControl, Database, DatabaseIF*	
SR-FUN-1550/3.0	*ModelMgr, ModelDC, ModelView*	
SR-FUN-1560/3.0	*ModelMgr, ModelDC, ModelView*	
SR-FUN-1570/3.0	*ModelMgr, ModelDC*	
SR-FUN-1580/3.0	*ModelMgr, ModelDC, ModelView*	
SR-FUN-1590/3.0	*ModelMgr, ModelDC*	

System Req.	Design component	Comments
SR-FUN-1600/3.0	*ModelMgr*	
SR-FUN-1610/3.0	*ModelMgr, ModelDC*	
SR-FUN-1620/3.0	*Controller, DescriptorDC, ModelDC, SessionDC, SimDC, ToolDC*	
SR-FUN-1630/3.0	*Controller, DescriptorDC, ModelDC, SessionDC, SimDC, ToolDC*	
SR-FUN-1640/3.0	*ConnectionControl, Database, DatabaseIF, DBServer, Controller*	
SR-FUN-1650/3.0	*Managers*	
SR-FUN-1660/3.0	*Controller, Managers*	
SR-FUN-1670/3.0	*ConnectionControl, Database, DatabaseIF, DBServer, DescriptorDC, ModelDC, SessionDC, SimDC, ToolDC, StageDC*	
SR-FUN-1680/3.0	*OSFEG library*	
SR-FUN-1690/3.0	*OSFEG library*	
SR-FUN-1700/3.0	*OSFEG library*	
SR-FUN-1710/3.0	*OSFEG library*	
SR-FUN-1720/3.0	*OSFEG library*	
SR-FUN-1730/3.0	*OSFEG library*	
SR-IMP-0010	*R*	
SR-IMP-0020	*R*	
SR-IMP-0030	*R*	
SR-IMP-0040	*R*	
SR-IMP-0050	*R*	
SR-IMP-0060	*R*	
SR-IMP-0070	*R*	
SR-IMP-0080	*R*	
SR-IMP-0090	*R*	
SR-IMP-0100	*R*	
SR-IMP-0110	*R*	
SR-IMP-0120	*R*	
SR-INS-0010	*R*	
SR-INT-0010	*Model*	
SR-INT-0020	*Model*	
SR-INT-0030	*Logger,SessionExecutor,ModelChainExecutor*	
SR-OPE-0010	*Mmi*	
SR-OPE-0020	*SessionMgr,SessionExecutor,ModelChainExecutor*	
SR-OPE-0030	*R*	
SR-OPE-0040	*R*	

System Req.	Design component	Comments
SR-OPE-0050	*R*	
SR-RES-0010	*Mmi*	
SR-RES-0020	*Database*	
SR-RES-0030	*Model*	
SR-SAF-0080	*R*	

9.2. Inverse Traceability

In this section we include a traceability matrix from system design components to system requirements.

Table 24: Inverse Traceability Table

Design component	System Req.
Database	SR-FUN-0100 - DELETED SR-FUN-0120 - DELETED SR-FUN-0170 SR-FUN-0190 SR-FUN-0210 SR-FUN-0250 SR-FUN-0290 SR-FUN-0300 SR-FUN-0330 SR-FUN-0380 SR-FUN-0400 SR-FUN-0420 SR-FUN-0490 SR-FUN-0500 SR-FUN-0510 SR-FUN-0550 SR-FUN-0580 SR-FUN-0630 SR-FUN-0650 SR-FUN-0720 SR-FUN-0750 SR-RES-0020 SR-FUN-1100 SR-FUN-1110 SR-FUN-1540/3.0 SR-FUN-1640/3.0 SR-FUN-1670/3.0
DatabaseIF	SR-FUN-0100 - DELETED SR-FUN-0120 - DELETED SR-FUN-0170 SR-FUN-0190 SR-FUN-0210

Design component	System Req.
	SR-FUN-0250 SR-FUN-0290 SR-FUN-0300 SR-FUN-0330 SR-FUN-0380 SR-FUN-0400 SR-FUN-0420 SR-FUN-0490 SR-FUN-0500 SR-FUN-0510 SR-FUN-0550 SR-FUN-0580 SR-FUN-0630 SR-FUN-0650 SR-FUN-0720 SR-FUN-0750 SR-RES-0020 SR-FUN-1100 SR-FUN-1110 SR-FUN-1540/3.0 SR-FUN-1640/3.0 SR-FUN-1670/3.0
DBServer	SR-FUN-0100 - DELETED SR-FUN-0120 - DELETED SR-FUN-0170 SR-FUN-0190 SR-FUN-0210 SR-FUN-0250 SR-FUN-0290 SR-FUN-0300 SR-FUN-0330 SR-FUN-0380 SR-FUN-0400 SR-FUN-0420 SR-FUN-0490 SR-FUN-0500 SR-FUN-0510 SR-FUN-0550 SR-FUN-0580 SR-FUN-0630 SR-FUN-0650 SR-FUN-0720 SR-FUN-0750 SR-RES-0020 SR-FUN-1100 SR-FUN-1110 SR-FUN-1640/3.0 SR-FUN-1670/3.0

Design component	System Req.
Managers	SR-FUN-0120 - DELETED SR-FUN-0580 SR-FUN-0750 SR-FUN-1650/3.0 SR-FUN-1660/3.0
Model	SR-FUN-0010 SR-FUN-0060 SR-FUN-0070 SR-FUN-0080 SR-FUN-0090 SR-FUN-0200 SR-FUN-0210 SR-FUN-0260 SR-FUN-0270 SR-FUN-0280 SR-FUN-0290 SR-FUN-0300 SR-FUN-0310 SR-FUN-0320 SR-FUN-0330 SR-INT-0010 SR-INT-0020 SR-RES-0030
ModelMgr	SR-FUN-0060 SR-FUN-0070 SR-FUN-0080 SR-FUN-0090 SR-FUN-0200 SR-FUN-0210 SR-FUN-0220 SR-FUN-0230 SR-FUN-0240 SR-FUN-0250 SR-FUN-0260 SR-FUN-0270 SR-FUN-0280 SR-FUN-0290 SR-FUN-0300 SR-FUN-0310 SR-FUN-0320 SR-FUN-0330 SR-FUN-0460 SR-FUN-0780 SR-FUN-0790 SR-FUN-1540/3.0 SR-FUN-1550/3.0 SR-FUN-1560/3.0 SR-FUN-1570/3.0

Design component	System Req.
	SR-FUN-1580/3.0 SR-FUN-1590/3.0 SR-FUN-1600/3.0 SR-FUN-1610/3.0
ModelDC	SR-FUN-0060 SR-FUN-0070 SR-FUN-0080 SR-FUN-0090 SR-FUN-0220 SR-FUN-0230 SR-FUN-0240 SR-FUN-0280 SR-FUN-0290 SR-FUN-0310 SR-FUN-0320 SR-FUN-1540/3.0 SR-FUN-1550/3.0 SR-FUN-1560/3.0 SR-FUN-1570/3.0 SR-FUN-1580/3.0 SR-FUN-1590/3.0 SR-FUN-1610/3.0 SR-FUN-1620/3.0 SR-FUN-1630/3.0 SR-FUN-1670/3.0
ModelView	SR-FUN-0060 SR-FUN-0070 SR-FUN-0080 SR-FUN-0090 SR-FUN-0220 SR-FUN-0230 SR-FUN-0240 SR-FUN-0280 SR-FUN-0290 SR-FUN-0310 SR-FUN-0320 SR-FUN-1540/3.0 SR-FUN-1550/3.0 SR-FUN-1560/3.0 SR-FUN-1580/3.0
Instrument	SR-FUN-0100 - DELETED SR-FUN-0110 - DELETED SR-FUN-0120 - DELETED
Descriptor	SR-FUN-0130 SR-FUN-0140 SR-FUN-0150 SR-FUN-0160 SR-FUN-0170

Design component	System Req.
	SR-FUN-0180 SR-FUN-0190
DescriptorMgr	SR-FUN-0130 SR-FUN-0140 SR-FUN-0150 SR-FUN-0160 SR-FUN-0170 SR-FUN-0180 SR-FUN-0190 SR-FUN-0460 SR-FUN-0510 SR-FUN-0900
DescriptorDC	SR-FUN-0130 SR-FUN-0140 SR-FUN-0150 SR-FUN-0160 SR-FUN-0170 SR-FUN-0180 SR-FUN-0190 SR-FUN-1620/3.0 SR-FUN-1630/3.0 SR-FUN-1670/3.0
DescriptorView	SR-FUN-0130 SR-FUN-0140 SR-FUN-0150 SR-FUN-0160 SR-FUN-0180
Stage	SR-FUN-0340 SR-FUN-0350 SR-FUN-0360 SR-FUN-0370 SR-FUN-0380 SR-FUN-0390 SR-FUN-0400 SR-FUN-0410 SR-FUN-0420
StageMgr	SR-FUN-0340 SR-FUN-0350 SR-FUN-0360 SR-FUN-0370 SR-FUN-0380 SR-FUN-0390 SR-FUN-0400 SR-FUN-0410 SR-FUN-0420 SR-FUN-0470
StageDC	SR-FUN-0340

Design component	System Req.
	SR-FUN-0350 SR-FUN-0360 SR-FUN-0370 SR-FUN-0390 SR-FUN-0410 SR-FUN-1670/3.0
StageView	SR-FUN-0340 SR-FUN-0350 SR-FUN-0360 SR-FUN-0370 SR-FUN-0390 SR-FUN-0410
Sim	SR-FUN-0010 SR-FUN-0430 SR-FUN-0440 SR-FUN-0450 SR-FUN-0460 SR-FUN-0470 SR-FUN-0480 SR-FUN-0490 SR-FUN-0500 SR-FUN-0510 SR-FUN-0520 SR-FUN-0530 SR-FUN-0540 SR-FUN-0550 SR-FUN-0560 SR-FUN-0570 SR-FUN-0580
SimMgr	SR-FUN-0010 SR-FUN-0430 SR-FUN-0440 SR-FUN-0450 SR-FUN-0460 SR-FUN-0470 SR-FUN-0480 SR-FUN-0490 SR-FUN-0500 SR-FUN-0510 SR-FUN-0520 SR-FUN-0530 SR-FUN-0540 SR-FUN-0550 SR-FUN-0560 SR-FUN-0570 SR-FUN-0580
SimDC	SR-FUN-0430 SR-FUN-0440

Design component	System Req.
	SR-FUN-0450 SR-FUN-0460 SR-FUN-0470 SR-FUN-0480 SR-FUN-0520 SR-FUN-0530 SR-FUN-0540 SR-FUN-0560 SR-FUN-0570 SR-FUN-1620/3.0 SR-FUN-1630/3.0 SR-FUN-1670/3.0
SimView	SR-FUN-0430 SR-FUN-0440 SR-FUN-0450 SR-FUN-0460 SR-FUN-0470 SR-FUN-0480 SR-FUN-0510 SR-FUN-0520 SR-FUN-0530 SR-FUN-0540 SR-FUN-0560 SR-FUN-0570
Session	SR-FUN-0010 SR-FUN-0020 SR-FUN-0030 SR-FUN-0590 SR-FUN-0600 SR-FUN-0610 SR-FUN-0620 SR-FUN-0630 SR-FUN-0640 SR-FUN-0650 SR-FUN-0660 SR-FUN-0670 SR-FUN-0680 SR-FUN-0690 SR-FUN-0700 SR-FUN-0720 SR-FUN-0730 SR-FUN-0740 SR-FUN-0750 SR-FUN-0760 SR-FUN-0770 SR-FUN-0820
SessionMgr	SR-FUN-0020 SR-FUN-0590

Design component	System Req.
	SR-FUN-0600 SR-FUN-0610 SR-FUN-0620 SR-FUN-0630 SR-FUN-0640 SR-FUN-0650 SR-FUN-0660 SR-FUN-0670 SR-FUN-0680 SR-FUN-0690 SR-FUN-0700 SR-FUN-0720 SR-FUN-0730 SR-FUN-0740 SR-FUN-0750 SR-FUN-0760 SR-FUN-0770 SR-FUN-0820 SR-FUN-0830 SR-FUN-0840 SR-FUN-0860 SR-FUN-0870 SR-FUN-0890 SR-FUN-0920 SR-FUN-0930 SR-FUN-0940 SR-FUN-0990 SR-FUN-1000 SR-OPE-0020
SessionDC	SR-FUN-0590 SR-FUN-0600 SR-FUN-0610 SR-FUN-0620 SR-FUN-0670 SR-FUN-0680 SR-FUN-0690 SR-FUN-0700 SR-FUN-0730 SR-FUN-0740 SR-FUN-0760 SR-FUN-0770 SR-FUN-0780 SR-FUN-0790 SR-FUN-0820 SR-FUN-0940 SR-FUN-0950 SR-FUN-0960 SR-FUN-0970 SR-FUN-1620/3.0

Design component	System Req.
	SR-FUN-1630/3.0 SR-FUN-1670/3.0
SessionView	SR-FUN-0020 SR-FUN-0590 SR-FUN-0600 SR-FUN-0610 SR-FUN-0620 SR-FUN-0630 SR-FUN-0640 SR-FUN-0670 SR-FUN-0680 SR-FUN-0690 SR-FUN-0700 SR-FUN-0710 SR-FUN-0730 SR-FUN-0740 SR-FUN-0760 SR-FUN-0770 SR-FUN-0780 SR-FUN-0790 SR-FUN-0800 SR-FUN-0810 SR-FUN-0820 SR-FUN-0900 SR-FUN-0910 SR-FUN-0930 SR-FUN-0950 SR-FUN-0960 SR-FUN-0970 SR-FUN-0980 SR-FUN-1010
Parameter	SR-FUN-0710 SR-FUN-0780 SR-FUN-0790
Tool	SR-FUN-0940 SR-FUN-0990 SR-FUN-1000
ToolMgr	SR-FUN-0030 SR-FUN-0760 SR-FUN-0940 SR-FUN-0990 SR-FUN-1000
ToolDC	SR-FUN-1620/3.0 SR-FUN-1630/3.0 SR-FUN-1670/3.0
Logger	SR-FUN-0880 SR-FUN-0970

Design component	System Req.
	SR-FUN-1010 SR-FUN-1020 SR-FUN-1345/3.0 SR-INT-0030
SessionExecutor	SR-FUN-0010 SR-FUN-0830 SR-FUN-0840 SR-FUN-0850 SR-FUN-0860 SR-FUN-0890 SR-FUN-0920 SR-FUN-0930 SR-FUN-1470/3.0 SR-INT-0030 SR-OPE-0020
ModelChainExecutor	SR-FUN-0010 SR-FUN-0840 SR-FUN-0860 SR-FUN-0890 SR-FUN-0920 SR-FUN-0930 SR-FUN-1480/3.0 SR-FUN-1490/3.0 SR-FUN-1500/3.0 SR-FUN-1510/3.0 SR-FUN-1520/3.0 SR-FUN-1530/3.0 SR-INT-0030 SR-OPE-0020
ParallelScheduler	SR-FUN-1450/3.0 SR-FUN-1460/3.0 SR-FUN-1470/3.0
Controller	SR-FUN-1480/3.0 SR-FUN-1490/3.0 SR-FUN-1500/3.0 SR-FUN-1510/3.0 SR-FUN-1520/3.0 SR-FUN-1530/3.0 SR-FUN-1620/3.0 SR-FUN-1630/3.0 SR-FUN-1640/3.0 SR-FUN-1660/3.0
Mmi	SR-RES-0010
ConnectionControl	SR-FUN-1100 SR-FUN-1110 SR-FUN-1540/3.0 SR-FUN-1640/3.0

Design component	System Req.
	SR-FUN-1670/3.0
pms.Parameter	SR-FUN-0780 SR-FUN-0790 SR-FUN-1120 SR-FUN-1130 SR-FUN-1160
pms.Rule	SR-FUN-0780 SR-FUN-0790 SR-FUN-1120 SR-FUN-1130 SR-FUN-1160
pms.ParameterEditor	SR-FUN-0800 SR-FUN-0810 SR-FUN-1130 SR-FUN-1140 SR-FUN-1150 SR-FUN-1160 SR-FUN-1170
pms.RulesEditor	SR-FUN-1130 SR-FUN-1160
OSFEG library	SR-FUN-1680/3.0 SR-FUN-1690/3.0 SR-FUN-1700/3.0 SR-FUN-1710/3.0 SR-FUN-1720/3.0 SR-FUN-1730/3.0

End of document