System
User Manual

Open Simulation Framework

openSF

Javier Martin Ávila – Technical Responsible

Mercedes Pavía – Project Manager

Document Code: OPENSF-DMS-TEC-SUM01

Version: 4.9

Date: 18/06/2025

Confidentiality Level: Unclassified

Indra Deimos

indracompany.com

This page intentionally left blank

Document Status Log

Issue	Section	Change Description	Date
1.0		First issue of this document	21/12/09
1.1		Version of this document after openSF AR1	15/03/10
	3.6.2	Installation details section completed	
	N/A	New chapter describing the openSF web page.	
	Annex D	New chapter – tutorial for defining an E2E simulation in openSF	
	4.5.4	Product tools section updated, with a list of popular product tools	
1.2		New version in response to ESA assessment for openSF version 1.1	20/04/10
	3.5.1	Update clarifying that the openSF installation mechanism for Linux platforms is the same as the Windows one. Installation of JRE under Linux completed.	
	N/A	Bin folder reference removed. OSFI folder added	
	4.5.4	Section 4.5.4 updated. Tools for MacOS issue.	
	Annex A	Added annex detailing how to build openSF from sources files.	
	N/A	Updated with latest changes for the web site.	
	N/A	Updated folder structure guidelines.	
1.3		Minor corrections:	26/04/10
	5.2	Clarifications	
	6	Typos fixed	
1.4		New version for the openSF v2 acceptance	22/09/10
	N/A	Section added: Management of databases	
1.5	N/A	Added Annex B detailing Parameter Editor functionality	15/10/10
1.6		Update after openSF AR 2 meeting:	12/11/10
	N/A	Added IDL and Matlab windows for the Linux installation	
	N/A	Updated functioning of databases in the multi-repository: Independence between databases.	
	N/A	Added IDL requirements for Linux installation: Problems with installation path and different types of licenses	
	N/A	Added Matlab requirements for Linux installation: licenses	
	Annex A	Updated introduction.	
	N/A	Removed import capabilities in Simulation creation	
	(all)	Updated obsolete screenshots.	
	Annex B	Added new section for module developers.	
2.0		New version including extended capabilities for openSF 2.2:	16/02/12
		Parameter Perturbation plug-in (from SEPSO)	
	4.1.1	Parameter Editor integration	
	4.3.5	Tool management extension	
	N/A	Check output generation	
	N/A	MATLAB errors inclusion	
	4.4.4	Import/Export capability	
	N/A	Extended log capabilities	
	N/A	Keyboard shortcuts	

Issue	Section	Change Description	Date
2.1		New version after version 2.2 acceptance meeting	02/03/12
	3.5.1.1	Added IDL version selection to openSF installer	
	N/A	Plot perturbation capabilities	
3.0		New version including extended capabilities for openSF V3:	22/11/13
	3.4.3	Updated framework pre-requisites	
	3.5.1	Updated installation instructions, leaving only references to the supported operating system – Linux	
	N/A	Added references to OSFEG libraries;	
	3.5.1	Added section on migration from previous versions to V3	
	4	Updated framework figures	
	4.5.2	Added reference to new system configuration parameter to control module parallelisation	
	4.2.2.5	Added section on importing an XML database definition	
	4.6.3	Added section on the CPU core usage view that supports module parallelisation	
	4.3	Added copy capabilities for several openSF elements	
	N/A	Updated the module chain management in a simulation according to the openSF framework revision	
	4.4.1.4	Added capability for removal of intermediate output files	
	N/A	Added capabilities for simplified module management:	
	4.4.1.1	Switching a module version	
	4.4.1.2	Bypass/Switch-off module execution	
	N/A	Run from a given point in the module chain	
	N/A	Added capabilities for parallelisation of module execution	
	4.4.4	Added capability for exporting/importing module of an executed simulations	
3.1		New version answering the comments generated by ESA on the openSF V3 AR documentation package. Implementation of the following RIDs:	15/01/14
	4.5.1	OSF-AR3-05: Update semantics of maximum number of threads parameter with associated warning message	
	4.5.2	OSF-AR3-06: Updates database view related figures	
	4.11.6.1	OSF-AR3-07: Added clarification on the log functionality in the case of module parallelisation	
	4.13	OSF-AR3-10: Clarification on simplification of simulation directory name	
	4.19	OSF-AR3-11: Renamed section to "Table of keyboard shortcuts"	
	N/A	OSF-AR3-RF-01: Updated change log to list sections changed for openSF v3;	
	2	OSF-AR3-RF-02: Updated Applicable and Reference documents	
	Annex D	OSF-AR3-RF-03: Revised Tutorial according to openSF V3 and added a reference to the training material (course handouts);	
	3.3.3.3	OSF-AR3-CE-01: Corrected the supported IDL versions	
	3.3.3	OSF-AR3-CE-02: updated pre-requisites section to appear chronologically before framework installation	
	N/A	OSF-AR3-CE-03: Updated section to clarify typical definition of OPENSF_HOME;	
	3.5.1	OSF-AR3-CE-04: Added a clarification regarding the installation an execution of openSF in the appropriate machine architecture.	
3.2		New version including the integration of Python modules in openSF.	04/01/14

Issue	Section	Change Description	Date
3.3	3.3.3.5	Updated after review comments from ESA: implemented RIDS OPENSF_v3.2_RID_01 and OPENSF_v3.2_RID_07.	30/04/14
3.4		New version with updated installation procedure including the porting to OSX.	03/06/15
3.5	4.4.2.4	New version including extended capabilities for openSF V3.4: Time Based Scenario Orchestration.	02/12/15
	N/A	Included special conditions for installation in OSX.	
3.6	4.4.2	OPENSF-AN-004: The "iterate parameters" functionality is able to import parameter iteration definitions from file	16/03/16
	(all)	OPENSF-AN-017: Updated SUM to use "modules" instead of "models".	
	N/A	Added section on execution of script modules and special conditions applicable to OSX 10.11.	
3.7		Updated with extended capabilities for openSF V3.5:	15/06/16
	4.4.6	OPENSF-AN-003: Remote orchestration;	
	3.6.3	OPENSF-AN-019: Automatic openSF version checking;	
	4.3.3.6.6	OPENSF-AN-030: Select subset of parameters to monitor;	
	4.1	Miscellanea HMI corrections and simplifications.	
	4.3.1	Descriptor syntax clarifications.	
3.8	(all)	Overall review based on ESA comments	06/06/16
3.9	3.4	Updated installation requirements.	20/01/17
3.10	(all)	Updated with HMI revamping for Eclipse RCP.	09/06/17
3.11	3.3.2	Updated with COTS requirement	28/08/17
3.12	3.4.3	Remove OSFI from framework pre-requisites	15/12/17
	N/A	Update MySQL tools path configuration during installation	
	N/A	Database created during installation selected by the user	
	3.3.1	Update folder structure	
	4.2.2	Small updates on database connection, deletion and backup due to bug fixes	
	N/A	OpenSF log messages moved to simulation dedicated log files	
	N/A	Move ParameterEditor appendix to dedicated SUM	
3.13		Update for openSF version 3.7.2	15/06/18
	3	Removal of the V2 upgrade path.	
	3.4	Update of the supported platforms.	
	3.4.3	Reorganization, splitting off the module-specific requirements into sec.6.3.	
	3.6	Update of the command line arguments section.	
	4.9.6	New dialog for missing configuration files.	
	4.9.6.9	Update simulation export/import files, dialog	
	4.12, 4.17	Add per-module log files.	
	Annex E	Update of build instructions	
3.14		Update for openSF version 3.7.3:	14/12/18
	3,4	Added Windows as a supported platform.	
	3.4.2	Updated the folder structure of the OpenSF installation folder.	
	4.11.5.4, 4.13	Documented new feature for grouping timeline and iteration/perturbation simulations.	
	3,4	Updated the dependency from MySQL, since the client tools are not necessary.	

Issue	Section	Change Description	Date
	4.5.1	XSD Validation and other changes in Preferences dialog.	
	(all)	Updated several images to comply with the newly delivered version.	
	3.4.3	Updated licencing scheme	
	N/A	Various minor changes	
3.15		Update for openSF version 3.8.0:	06/06/19
	(all)	General revision and update of the document.	
	2	Update the description of the installation process.	
	4.9.3.2	New section on iterations on parameter sets.	
	4.13	Include the concept of the "User Roles" and explain them.	
	4.14	Addition of some guidelines on how to implement a Monte Carlo study in openSF.	
3.16		Update for openSF version 3.8.1:	17/07/19
	3.4.3	Modification of supported versions to align them with the pre-requisites	
	N/A	Added description of the new preferences added to the Application Settings preferences page.	
3.17		Update for openSF version 3.9.0:	27/11/19
	(all)	Removal of the Stage and Simulation concepts.	
	(all)	Renaming of the previous "Session" concept to "Simulation".	
	3.5.1	Clarify the setup of the $INSTALL4J_JAVA_HOME environment variable.	
	(all)	Update all images to the latest openSF HMI.	
	(all)	Removed "product" concept.	
3.18		Update for openSF version 3.9.2	12/03/20
	4.3.3.6.6	New feature: Python-like format for parameter value edition.	
3.19		Update for openSF version 3.9.3	03/07/20
		Added preference "Symlink handling mode"	
	N/A	New section Parameter Series	
	4.9.2	Sub-sections renumbered, updated images	
	N/A	New conditions for inline editing	
	N/A	Introduction of E2E_HOME variable and new meaning for OPENSF_HOME	
	4.9.3	Moved Perturbations explanation	
4.0		Update for openSF version 3.9.5	17/12/20
	(all)	Overall review of the document	
	4.5.4	New section	
	6.3	New section	
4.1		Update for openSF version 3.10	24/06/21
	3.6.2	New section	
	Annex E	New section	
	3.5.1, 3.5.2, 4.6.1, 3.4.3, 4.2.1	Updated	
4.2		Update for openSF version 3.11	01/12/21
	3.1, 3.4.3, 3.5	Removed requirement of a pre-installed Java runtime (openSF now bundles it), modified GTK+ pre-requisite	

Issue	Section	Change Description	Date
4.3	3.5.1.3	Updated uninstall mechanism for files removal and related screenshot	27/05/22
	3.6.3	Added update dialog screenshot.	
	4.2, 3.5.1.4	New section on the workspace selection feature. Also modified related table.	
	4.2.3	Updated with comment about symlink creation on Windows	
	4.3.3.6.5	Updated with explanation about string parsing	
	4.3.5.1	Updated with clarifications about the "new tool" dialog fields	
	3.4.3.1	Updated for required MySQL/MariaDB server settings	
	8.1	Updated JDK version to 11, and updated the build procedure	
		Update for openSF version 4.0	
	3.1, 3.4	Updated minimum requirements.	
	3.6	Expand explanations on launcher options and advise against running long simulations via SSH.	
	4.2.2.4	Update the description of the XML import/export mechanism, and explain the non-portable paths warning	
	4.3.2.2, 4.5.2	Update module view explanation, remove the XSD schema validation.	
	4.3.3	Update the simulation view description and screenshot for the new graphical simulation editor and execution monitor.	
	Annex C	New Annex on the packaging and delivery of complete simulators	
4.4		Update for openSF version 4.1	15/12/22
	(all)	Updated screenshots overall	
	3.4.2, 3.6.1	Added workaround for Linux Wayland issues.	
	4.3.2.2.2	Remove per-module visibility rules.	
	4.3.5	Added built-in tool "OS default app".	
	4.4.3.6, 4.4.3	Added reference to the possible new warnings during parameters reload.	
	4.3.3.6.5	Update with the redesigned parameters visibility dialog and add the new per-simulation visibility rules files.	
	4.4.2	Update subsections with the redesigned UI for the multi-execution of a simulation, the DB persistence of the configurations, the individual dialogs for each configuration type, the file import/export functionality and the issues that may be caused by changing parameters after multi-execution is configured.	
	4.4.3	Update description of simulation execution log table scroll behaviour.	
	4.5.3	Remove unused "temp" folder.	
	Annex D	Update example simulation structure (changed in 4.0)	
	Annex E	Update pre-requisites for building openSF in Sec. 9.1.	
4.5		Update for openSF version 4.2	23/05/23
	4.2.2.4	Added information on XML DB upgrade.	
	4.3.3.6	Modified module creation UI description and screenshots.	
	4.3.3.6.5	Updated descriptions of parameter validity.	
	4.4.2.1	Updated parameter iteration UI explanation.	
	4.4.3,4.5.2	Added new options for the action on simulation failure.	
	E.2.2	Added mention of the static analysis build target.	
4.6		Update for openSF version 4.3	01/12/23

Issue	Section	Change Description	Date
	3.5.1	Coalesced installation sections for each OS into a single section and added information on workspace selection.	
	3.5.1.3	Updated the folder structure to reflect the packaged example simulator and the new optional default_database.xml file.	
	3.6.2	Added a note about the workspace.	
	4.2	Updated to state that using the program install folder as a workspace is deprecated and strongly advised against.	
	4.2.2.1	Added caution that all views will be closed on a database switch.	
	4.2.2.2	Updated with new "defaults from DB" option.	
	4.2.2.4	Updated with new "export only simulation elements" option.	
	4.3.3.6.2	New section explaining the algorithm that resolves conflicts in the file names when a simulation is launched.	
	4.3.4,4.4.1	Updated screenshots.	
	4.4.1.2	Removed	
	4.4.2.1	Sec. rewritten after redesign of the arithmetic sequence editor.	
	(all)	Update paths.	
	4.4.3.4	Add mention to a simulation group being aborted on tab closure.	
	4.4.5	Update to state that a script aborts on any module failing.	
	4.5.3	Rewrite to explain the new fields in the folders preferences.	
	4.5.4	Remove exception for Python modules in Windows.	
	B.4	New section advising how to deal with large reference data files.	
	7.2, 7.3	Add mentions to the default database XML file and the example simulator package	
4.7		Update for openSF version 4.4	31/05/24
	2	Updated reference and applicable documents.	
	3.4	Updated system requirements after Eclipse RCP upgrade.	
	3.5.1.3	Updated folder structure to reflect that database files are now placed in the workspace folder by default.	
	3.6.1	Add instructions for troubleshooting certain graphical issues.	
	4.2.2	New section about database version mismatch and upgrade.	
	4.3.3.6.2	Mention modules with identical LCF names.	
	4.3.3.6.6	Updated the parameters visibility dialog.	
	4.4.4	Removed the mentions to SQL files and updated the destination path to the workspace folder.	
	4.4.4.2	Clarify where an imported simulation appears.	
4.8		Update for openSF version 4.5	19/11/24
	2	Update reference and applicable documents.	
	3.3.1, 3.3.2, 3.5.1.1	Merged the mentioned sections into the first one, and rewrite it explaining the conventional paths inside openSF. Also, remove some mentions to legacy versions of openSF.	
	3.4	Add support for Apple Silicon.	
	3.6.3	Recommend a workspace backup, and remove legacy mentions.	
	4	Update screenshots.	
	4.2.2.5	Add notice that importing a database converts the stored paths to the system native format.	

Issue	Section	Change Description	Date
4.9	4.3.2.2.3, 4.3.3.6	Add mention to the monitorization of descriptor/module changes in the module/simulation editors.	18/06/25
	4.3.3.5	Simplify the simulation modification description.	
	4.4.1.3	Restore section explaining pre-existing functionality, which was dropped in a prior version by mistake.	
	4.4.3	Explain the dialogs when closing a simulation editor tab with running simulations.	
		Update for openSF v4.6	
	4.1.1.1	New description of the file system tab and its find functionality.	
	4.3.5.6	Explain new "when to run" column in post-processing tools table.	
	4.4.1.1	New description and corresponding screenshot for the dialog shown when modules of different versions do not fully match in inputs and outputs.	
	4.4.3	Update screenshot and description of the executions view.	
	7	Rewrite sections to reference the workspace instead of the installation folder as appropriate, and clarify how to package simulators from a live workspace.	
	9.2	Update build instructions, mention the OSFI-Java pre-built package.	

Table of Contents

Indra Deimos

IND-CMS-SUPTR09-SUM-10-E

List of Figures

List of Tables

1. INTRODUCTION

This document has been produced by DEIMOS within the frame of different openSF contracts and it represents the System User Manual for the openSF platform.

OpenSF is a software framework aimed at supporting a standardised end-to-end simulation capability ([AD-E2E]) allowing the 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 has been conceived to support concept and feasibility studies for the ESA Earth Observation Programs (EOP) activities, where the mission performance up to the final data products needs to be predicted by means of end-to-end (E2E) simulators; in later development phases, openSF becomes a coherent test bed for L1PP and L2PP, and to support the verification of space segment performance and associated sensitivity analysis.

Nevertheless, openSF has been designed and developed in a generic way, allowing its use as a simulation framework for any E2E processing chain in domains different from EO E2E performance simulators.

The openSF framework is released frequently, making updates and bugs fixes available to users multiple times a year.

1.1. Purpose

The objective of this document is to provide a clear description of all the openSF functionalities, as also an operational guide for developing and integrating an E2E simulation processing chain.

The intended readerships for this document are scientific module developers, E2E processing chains integrators, E2E performance analysts and E2E performance engineers. For more details, please refer to Sec. 3.2.1.

1.2. Scope

This document contains openSF v4.5 and its contents are organised as follows:

- ❑ Chapter 1, the present chapter, describes this document and sets the basis for its understanding.
- ❑ Chapter 2 collects the references to this SUM.
- ❑ Chapter 3 details the procedures for installing and setting up openSF.
- ❑ Chapter 4 describes all the different functionalities of openSF.
- ❑ Appendix A describes the openSF error messages
- ❑ Appendix B contains some guidelines for module developers.
- ❑ Appendix C presents a tutorial of the generation of an E2E simulation.Appendix D explains how to build the application.

1.3. Acronyms and Abbreviations

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

- **AD:** Applicable Document
- **API:** Application Programming Interface
- **CFI:** Customer Furnished Item
- **COTS:** Commercial Off The Shelf
- **CPU:** Central Processing Unit
- **DB:** Database
- **DBMS:** Database Management System

- **DMS:** DEIMOS Space

- **E2E:** End to end simulation

- **EOP:** Earth -Observation Programmes

- **ESA:** European Space Agency

- **GCF:** Global Configuration File

- **GUI:** Graphical User Interface

- **HMI:** Human Machine Interface

- **IO:** Input/Output

- **ICD:** Interface Control Document

- **IDL:** Interactive Data Language

- **JRE:** Java Runtime Environment

- **L1PP:** Level 1 Processor Prototype

- **L2PP:** Level 2 Processor Prototype

- **LCF:** Local Configuration File

- **MC:** Monte Carlo

- **OS:** Operating System

- **OSFI:** OpenSF Integration Library

- **RD:** Reference Document

- **SEPSO :** Statistical E2E Performance Simulator for Optical Imaging Sensors

- **SUM:** System User Manual

- **TBC:** To Be Confirmed

- **TBD:** To Be Defined / Decided

- **TN:** Technical Note

- **UML:** Unified Modelling Language

1.4. Definitions

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

- **Batch mode:** It is the capability of the simulator to perform consecutive runs without a continuous interaction with the user. Batch mode assesses 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:

    o   Iteratively, executing one or more simulations

    o   Iteratively, executing the same simulation several times depending on the parameters' configuration

    o   Same as above but by executing a batch script.

See section 3.6 for further details.

- **Configuration File:** An XML file that contains parameters necessary to execute a module. A configuration file instance must comply with the corresponding XML schema defined at module creation time. A special case is the global configuration file that defines the configuration parameters that may be common to different modules.

- **Descriptor:** The descriptors define the set of input and output files used to connect modules in simulation runs. Each module has two descriptors associated, one for the inputs and the other for the outputs. They define the number and location of each of the IO files. Descriptors are thoroughly described in section 4.3.1.

- **Framework:** Software infrastructure designed to support and control the simulation definition and execution. It includes the GUI, and persistence capabilities that enable to perform all the functionality of the simulator.

- **Module:** Executable entity that can take part in a simulation. A module can be understood, broadly speaking, also as an "algorithm". Basically, it contains the recipe to produce products as a function of inputs. A module 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 module consists of:

  o The source code and its binary compiled counterpart (or interpretable script)

  o A configuration file with its parameters

  o An input descriptor that characterizes its inputs (number and their default names)

  o An output descriptor that characterizes its outputs

Further details about modules are given in section 4.3.2.

❑ **Parameter:** An element of the system whose value characterizes a given aspect of a module, and is given in the configuration files. Parameters are user-editable, they can represent system constants or initial values of simulation variables.

❑ **Simulation:** A simulation is defined as an execution of a set of modules (either a unique execution or an iterative one with different parameter values). The restriction of how to concatenate these modules and the order on which they are executed is based on the logic imposed by the relation between their descriptors. For further details see section 4.3.3.

❑ **Time-Based Execution:** The Time-Based scenario execution implements the notion of time driven execution of a simulation whereby each simulation module is invoked in a sequence of time segments. See section 4.4.2.4 for further details.

❑ **Tool:** A tool is an external program that performs a given action taking as input a certain group of files. openSF can associate tools to a certain file extension. These tools can be automatically invoked to perform operations taking as input the output of a simulation. Tools are described in section 4.3.5.

2. RELATED DOCUMENTS

This section details the list of applicable and reference documents used for the generation of this document, as well as the standards that have been applied. Note that the latest issue and dates of the documents can be found on the openSF website (http://eop-cfi.esa.int/index.php/opensf).

2.1. Applicable Documents

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

Table 2-1: Applicable documents

Reference	Code	Title	Issue
[AD 1]	OPENSF-DMS-ICD-001	openSF Interface Control Document	3.0.1
[AD 2]	OPENSF-DMS-ADD-001	openSF Architecture Design Document	2.2
[AD-E2E]	PE-ID-ESA-GS-464	ESA generic E2E simulator Interface Control Document	1.4.2

2.2. Reference Documents

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

Table 2-2: Reference documents

Reference	Code	Title	Issue
[RD 1]	OPENSF-DMS-OSFI-DM	openSF Integration Libraries Developers Manual	1.25
[RD 2]	OPENSF-DMS-OSFEG-DM	openSF Error Generation Libraries Developers Manual	1.8
[RD-3]	OPENSF-DMS-PMD-HAO-WS1	openSF Training Workshop 2018	1.0
[RD-4]	OPENSF-DMS-PE-SUM	ParameterEditor Software User's Manual	1.12

2.3. Standards

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

Table 2-3: Standards

Reference	Code	Title	Issue
[STD 1]	ECSS-E-ST-40C	Software Engineering Standard	06/03/09
[STD 2]	www.w3.org/TR/xml11/	Extensible Markup Language (XML)	1.1

3. GETTING STARTED

3.1. Introduction

During the concept and feasibility studies for the ESA Earth Observation activities, the mission performance up to the final scientific products data needs to be predicted by means of end-to-end (E2E) simulators. The observing system characteristics that impact data quality need to be determined in order to achieve the scientific goals. On subsequent implementation phases, these mission E2E simulators become a coherent test bed for L1PP and L2PP and to support the verification of space segment performance and associated sensitivity analysis.

A mission E2E simulator is able to reproduce all significant processes, design and steps that impact the mission performance as well as output simulated data products.

Commonalities in the structure of these E2E simulators highlighted the need for a common modular framework. openSF is an open software framework to support a standardised set of E2E mission simulation capabilities allowing the assessment of the science goals and engineering requirements with respect to the mission objectives.

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

For the installation, detailed System Requirements are presented in section 3.4. For a quick installation strategy, the recommended base system is the following:

❑ Ubuntu 20.04 LTS (or higher), macOS 11 or higher, or Windows 10

3.2. OpenSF User Profiles and Roles

openSF is designed to accommodate different use cases for different kind of users. This section introduces the different openSF user profiles according to the intended use of openSF; and how this manual has been tailored to each of them.

3.2.1. User profiles definition and quick start guide

Four possible user profiles have been identified for openSF, and for each of them a quick-start guide is made available. Throughout the manual, specific tags indicate which users is a certain section addressing to. To increase readability, a tag assigned to a given section applies in cascade to all its sub-sections (if not differently specified).

The reader, once identified with one of the profiles, has the possibility to:

❑ Quickly scan through the quick-start guide and jump to the referenced section of interest.
❑ Read the manual thoroughly, skipping all the sections that do not contain the tag associated to the profile of interest.

Note that the user is free to read the manual to taste and that the current section only serves as a general recommendation.

3.2.1.1. Scientific modules developer

The first openSF user identified is the module developer. This user has the objective to develop the executable scientific processing modules that will compose the E2E processing chain meant to be integrated into openSF. The module developer is not mainly interested in the functioning of the integration framework itself, but only in the interfaces between openSF and the module(s) under development.

The module developer's tag used in this manual is: **M**

The sections of this manual tagged for the module developer user are:

❑ The manual's conventions, useful to understand the rest of the manual (section 3.3)
❑ The modules' pre-requisites in order to run openSF (section 6.3)
❑ The modules creation process (sections 4.3.2.2, 4.3.2.3)
❑ The parallelization techniques employed by openSF (section 4.4.3.1)
❑ The dedicated guide to develop modules for openSF (section 6)

If the module developer wishes to perform tests on the modules created, the reading of the sections reserved to the E2E performance analyst and processing chain integrator profiles is recommended, with special attention to the processes to create descriptors (section 4.3.1) and simulations (sections 4.3.3.2, 4.3.3.6).

3.2.1.2. *E2E processing chains integrator*

The End to End processing chain integrator is interested in setting up the simulation environment, integrating the modules into a simulation and delivering an E2E simulator to the user.

The E2E integrator's tag used in this manual is: | **!** |

The sections of this manual tagged for the E2E Integrator user are:

- ❑ The conventions used in this manual and the system requirements (sections 3.3, 3.4)
- ❑ The procedure to install and run the framework (sections 3.5, 3.6)
- ❑ The general UI elements of openSF (section 4.1)
- ❑ The data structure of openSF (section 4.2)
- ❑ The elements composing the core of openSF:
  - o The descriptors (section 4.3.1)
  - o The modules (section 4.3.2)
  - o The simulations:
    - ▪ How to list the available simulations (section 4.3.3.1)
    - ▪ How to create a simulation (sections 4.3.3.2, 4.3.3.6)
    - ▪ How to modify, copy and delete a simulation (sections 4.3.3.2, 4.3.3.4 and 4.3.3.3)
    - ▪ How to access other functionalities for a simulation run (section 4.4.1)
    - ▪ How the parallelization of module execution mechanism works (section 4.4.3.1)
    - ▪ How to export a simulation and generate a script from it (sections 4.4.4, 4.4.5)
- ❑ The preferences settings of openSF (section 4.5)
- ❑ Meta-data about openSF:
  - o The openSF license information (section 4.6.1)
  - o The external documents linked to openSF (section 4.6.2)
- ❑ The tutorial that explains how to generate and run a simulation from scratch (section 8)
- ❑ How to build the framework (section 9)
- ❑ How the environment variables are exported by openSF to the modules (section 6.2)

3.2.1.3. *E2E performance engineer*

The E2E performance engineer is interested in running simulations with openSF like the E2E performance analyst, but wants to also be able to finely control the simulation, exploiting openSF at its best.

The E2E performance engineer's tag used in this manual is: | **E** |

The E2E performance engineer is an extension of the E2E performance analyst, hence the great majority of the sections of interest are already included in those of the E2E performance analyst.

The sections of this manual tagged for the E2E performance engineer are the same for the E2E performance analyst (section 3.2.1.4) plus the following ones:

- ❑ The elements composing the core of openSF:
  - o The simulations:
    - ▪ How to create a simulation (sections 4.3.3.2, 4.3.3.6)
    - ▪ How to modify, copy and delete a simulation (sections 4.3.3.2, 4.3.3.4 and 4.3.3.3)
    - ▪ How to apply parameters variation methods to a simulation (section 4.4.2)
    - ▪ How to access other functionalities for a simulation run (section 4.4.1)
    - ▪ How the parallelization of module execution mechanism works (section 4.4.3.1)
    - ▪ How to export a simulation and generate a script from it (sections 4.4.4, 4.4.5)
  - o The external tools applicable to openSF' results (section 4.3.5)
- ❑ How to perform a multi-node simulation (section 4.4.6)
- ❑ Meta-data about openSF:
  - o How to monitor the CPU usage (section 4.6.3)
- ❑ The tutorial that explains how to generate and run a simulation from scratch (section 8)
- ❑ How to build the framework (section 9)
- ❑ How the environment variables are exported by openSF to the modules (section 6.2)

Indra Deimos

IND-CMS-SUPTR09-SUM-10-E

3.2.1.4. *E2E performance analyst*

The E2E performance analyst wants to run simulations already integrated in openSF, getting acquainted with all the steps that this would require.

The E2E performance analyst's tag used in this manual is: **A**

The sections of this manual tagged for the E2E performance analyst are:

- ❑ The conventions used in this manual and the system requirements (sections 3.3, 3.4)
- ❑ The procedure to install and run the framework (sections 3.5, 3.6)
- ❑ The general UI elements of openSF (section 4.1)
- ❑ The data structure of openSF (section 4.2)
- ❑ The elements composing the core of openSF:
  - o The descriptors (section 4.3.1)
  - o The modules (section 4.3.2.1)
  - o The simulations:
    - ▪ How to list the available simulations (section 4.3.3.1)
    - ▪ How to apply parameters variation methods to a simulation (section 4.4.2)
    - ▪ How to resume a simulation (section 4.4.3.2)
    - ▪ How to read the logs generated by openSF (section 4.4.3.3)
    - ▪ How simulations are grouped (section 4.4.3.4)
  - o The results (section 4.3.4)
- ❑ The preferences settings of openSF (section 4.5)
- ❑ Meta-data about openSF:
  - o The openSF license information (section 4.6.1)
  - o The external documents linked to openSF (section 4.6.2)
- ❑ The error message list (section 5)

3.2.2. *openSF user roles*

openSF, as a simulator integration framework, intends to support different types of users whose goals are clearly distinct. Each such type of user requires a different set of features for their typical use of the tool.

On the one hand, there is the user responsible for the development of the simulator modules and their integration into openSF to compose the simulator. This user is expected to have a deep understanding of openSF, its capacities and limitations. Typically, the work of this user unfolds during the development phase of the simulator, and it is expected that a significant number of modifications to the openSF simulation elements (i.e. descriptors, modules and simulations) will be needed until a state of maturity is reached and the simulator can be considered production ready. For these users, openSF provides the features enabled by "Developer" role. According to the definition of openSF user profiles (section 3.2.1), the "Developer" role is targeted to the Module Developer and E2E Integrator profile.

On the other hand, there is the user of the E2E simulators integrated in openSF. This user is not expected to understand openSF in detail, as their goal is to use the fully integrated simulator. This type of user is typically interested in modifying simulation inputs, executing the pre-defined simulations and collecting the results. In order to ensure reliable and repeatable results, this type of user should be not required (and eventually denied) to modify the openSF simulation elements. openSF provides a restricted set of features for these users, known as "Normal" role. According to the definition of user profiles (section 3.2.1), the "Normal" role is targeted to both E2E performance analyst and engineer profiles.

By default, openSF is configured in "Normal" role and the user role selection toolbar is hidden. If the user desires to create or modify any openSF simulation element, the "Enable user role selection" option can be enabled in the Application Settings (see section 4.5) to display the user role toolbar, and thus select the "Developer" role.
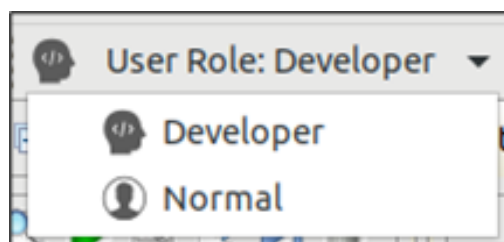
Figure 3-1: User role selection toolbar

The main differences between these two roles are the following:

- Developer:
  - The user can create new simulation elements (e.g. descriptors, modules, simulations).
  - The user can modify previously created simulation elements.
  - The user can execute simulations.
  - The repeatable production of the results is not guaranteed as simulation elements can be modified between two simulation executions.
- Normal:
  - The user can create, modify and execute simulations.
  - The reliable and repeatable production of the results is guaranteed.

In both cases the user can delete the existing simulation elements, potentially resulting in cascading deletions of any related element which depends on deleted elements: deleting a descriptor will cause the deletion of all modules that refer to it, all simulations that refers to those deleted modules, and all results of those simulations.

Therefore, if during the development stage of the simulator, one of its module inputs get modified and a created descriptor is no longer needed but it is desired to keep the rest of the simulator unchanged, the user cannot directly delete it. Instead, while in "Developer" role, the affected modules shall be first modified to remove that descriptor from its inputs/outputs and only then it can be safely removed.

openSF applies a simulation fingerprint to keep track of the simulation results whose reliability is not assured. Unreliable results are displayed in the Executions tab of the Navigation pane with a light red background. Replicable results are shown on a white background (see Figure 3-2). At the end of the simulator development stage and before entering in production stage, all the results with an unassured replicability should be removed.
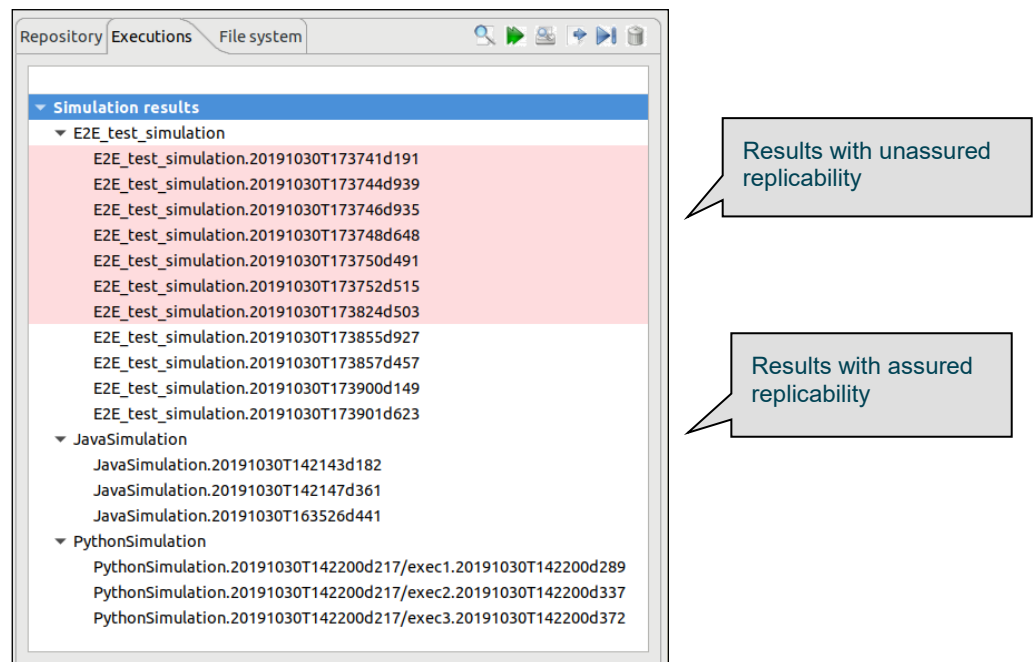
Figure 3-2: Executions tab and color code based on results replicability

A result is considered as potentially not reliable if any of the following criteria applies:

❑ The result was obtained while in "Developer" role.

❑ The computed fingerprint of the simulation, or any of its components, at calculation's time does not match the current fingerprint of that simulation and its current components.

3.3. Conventions used in this Manual

This chapter lists all the conventions used throughout this System User Manual.

3.3.1. Folder structure

This section provides a general description of the openSF folder structure and its contents. Detailed information about the contents of each folder is provided in Table 3-1 below.

<OPENSF_INSTDIR> represents the **installation directory** of openSF. This path is fixed and it is determined at installation time (see Figure 3-5). In a shared installation, it might be read-only.

<workspace> represents the **current workspace** of the application. It contains elements such as the openSF configuration and file-based databases. It must be writable for the user running openSF.

$E2E_HOME represents the **home folder** for simulators and their data. openSF uses it internally to resolve any relative paths in descriptors, modules and simulations e.g. to find a configuration or input file. This variable is also exported to the environment of running modules.

Like other environment variables, its value can be set through the system preferences window (see section 4.5.1). If it is not defined, openSF will first try to use the value of another environment variable ($OPENSF_HOME) that was used in prior versions for backward compatibility, and ultimately default to the workspace folder.

Table 3-1: Folder structure of openSF, its workspace and E2E_HOME

Root	Path	Contents
<OPENSF_INSTDIR>	/	openSF program folder root. After installation, this folder may be read-only. It contains many files, but the most relevant are: • *openSF*: framework launcher • *openSF_updater* and *uninstall*: utility programs • *openSF_defaults.properties*: optional configuration file that may contain default settings for all workspaces. See section 4.2.1.2.
	/data	Folder with a template for global configuration files.
	/features, /plugins, /p2, /configuration	Folders that contain Eclipse related files with fixed configuration. Not present in macOS installations.
	/ParameterEditor	Folder that contains the ParameterEditor executable (see section 4.3.3.6.4). In macOS installation, it is named "ParameterEditor.app".
	/resources /documentation	Default folder for framework documentation.
	/resources /simulators	Package with an example database, modules and related files
<workspace>	/	The workspace folder, which can be selected from within openSF. The user will be prompted to select a workspace folder on installation, see section 3.5.1. Must be writable by the user running openSF, contains file-based databases (*.db)
	/.metadata	Eclipse configuration applicable for each workspace.
	/openSF.properties	openSF configuration applicable for each workspace.
	/executions	openSF simulations root folder, configurable in the settings. This folder contains the output of all executions. Must be writable by the user running openSF.
	/executions/<ID>	Simulation folder. Every simulation, once executed, has one directory structured as this one. However, if the execution is of type Timeline, Iteration, Perturbation or Batch, there will be a parent folder with several sub-simulations under it. Each normal simulation folder (or timeline/iteration subfolder) contains the input and configuration files used by the modules and their outputs.
$E2E_HOME	/	Root location for "user" data, including module binaries and configuration/input files. If unspecified in the configuration/environment, defaults to <workspace>. Most relative paths specified by the user are composed relative to this location. It may also contain file-based databases (*.db files) if upgraded from a prior version.
	/default_database.xml	If it exists, it will be offered as an option when creating a database.

Note that on macOS, all Eclipse related files are inside the app package.

Users can also find useful the guidelines about how to organize the folder structure of a simulation project integrated in openSF. These guidelines are described in section 8.2.

3.3.2. Data types

The data types supported by openSF configuration files are described in [AD-E2E].

3.4. System Requirements

M	!	A	E

The openSF framework is developed and runs on the Eclipse Rich Client Platform (RCP). The current version of openSF is based on Eclipse 2024-03 (4.31), for which the target platforms officially supported by the Eclipse project can be found in the Eclipse project plan page.

3.4.1. Hardware requirements

!	A	E

Hardware must at least fulfil the following requirements:

- x86-64 or Apple Silicon processor
- 4 GB of RAM memory installed
- 400MB of free space to install.

3.4.2. Operating system requirements

M	!	A	E

Not all the platforms targeted by the Eclipse platform are officially supported by openSF. Binary distributions are currently provided for the following platforms:

- Linux: any sufficiently recent glibc-based distribution. In particular, openSF has been tested with Ubuntu 22.04. In systems using Wayland-based graphics, special configuration might be required, see sec. 3.6.1.
- macOS, version 12 or higher.
- Windows 10

3.4.3. Framework pre-requisites

!	A	E

All the openSF software pre-requisites are freely downloadable.

Table 3-2: Software pre-requisites

Pre-requisite	Purpose	Licensing	Distribution site
For server-based databases only: MariaDB server 10.5.2 or newer; MySQL server 5.7.x or 8.0[1]	openSF stores information in this relational database	GPL or Proprietary License	https://dev.mysql.com/downloads/mysql/ [MySQL] https://mariadb.org/download/ [MariaDB]
For server-based databases only: DB user with database creation and modification privileges	openSF needs DB creation/modification privileges		

---

[1] The DB server can be accessed over the network; it does not need to run in the same computer as openSF

GTK+ v3.22 or higher (Linux only)	openSF uses GTK+ as the graphics library in Linux Note: GTK known bug with Table/Tree editing that affects versions < 3.20.	GNU LGPL	https://www.gtk.org/download/index.php
mpstat (Linux only)	openSF uses this library to assess the CPU core usage statistics	GNU GPL	https://linux.die.net/man/1/mpstat
sshfs	openSF uses this library for remote execution	GNU GPL	https://osxfuse.github.io/ (OSX) https://github.com/libfuse/sshfs (Linux)

3.4.3.1. MySQL/MariaDB installation

If using server-based databases (versus file-based databases), the current openSF version ensures full compatibility with MySQL server v5.7.x and MariaDB server 10.5.2 (and higher). MySQL server version 8.0 is tentatively supported, but has not been tested extensively with openSF.

Most common Linux distributions include either MySQL or MariaDB in their default repositories, so it can normally be installed with the distribution default tools. Windows and macOS users may download either database server from either project's webpage. Documentation related to the installation of the database servers can be found in:

- ❑ MariaDB: https://mariadb.com/kb/en/binary-packages/
- ❑ MySQL: https://dev.mysql.com/doc/refman/5.7/en/installing.html

The configuration of the database server on MySQL and MariaDB must ensure that the server variable "lower_case_table_names" is never set to 1. Note that this is the default value on Windows, where it must be set to 2, which is case-preserving but case-insensitive.

Refer to section 4.2.2 for further details on databases.

3.4.3.2. Remote execution installation

Remote execution in openSF relies on mounting a remote file system through sshfs. To enable this, solution some pre-requisite software needs to be installed before openSF remote execution orchestration. Note that the Windows version of openSF does not support remote execution.

3.4.3.2.1. Linux installation

The sshfs installation method in case the Linux distribution provides a software package manager consists of installing the following packages: sshfs, fuse-utils. In case the Linux distribution does not provide an online package manager it is suggested to visit sshfs website (https://github.com/libfuse/sshfs) and look for alternative installation methods.

3.4.3.2.2. macOS installation

Sshfs installation package for macOS can be obtained from OXS Fuse official website (https://osxfuse.github.io/). It is recommended to download the DMG archive. The stable releases of both OSXFuse and SSHFS should be installed. Installation is based on a GUI installer with default configuration.

3.4.3.2.3. SSH access permission configuration

To ease the access to remote file system through sshfs it is required to enable access by sharing ssh keys (so that it is not required writing the password every time the connection is established).

The following commands implement the sharing of ssh keys between the participating computers:

```
~/$> ssh-keygen -t dsa
```

followed by:

```
~/$> ssh-copy-id -i .ssh/id_rsa.pub <user>@<machine>
```

For the above configuration the following packages are required: ssh-keygen, ssh-copy-id. Note that ssh-copy-id is not an officially OSX supported package so either an unofficial installer can be used (e.g. brew) or the public key needs to be copied manually.

3.5. How to Install the Framework

| I | A | E |

Provided that every pre-requisite is fulfilled, users can now proceed to install the application.

The openSF distribution package consists of an installer for each target platform. The installer will be in charge of the system deployment and the pre-requisites checking.

3.5.1. Installer guide setup

openSF is installed via a multi-platform GUI installer. To download the openSF software and documentation perform the following steps:

1. If not already done, register as a user on https://eop-cfi.esa.int/ (see "Create an account" link in right pane).

2. If not already done, register as openSF user at https://eop-cfi.esa.int/index.php/opensf/opensf-registration

3. Download the Software at https://eop-cfi.esa.int/index.php/opensf/download-installation-packages

Figure 3-3: openSF web page

After downloading the installer corresponding to the right machine architecture and operating system, users will execute it by the normal means for the platform, and follow the instructions that appear on the screen. The program first checks if openSF is already installed, offering to update the existing version (Figure 3-4). Otherwise, the first screen merely contains a welcome message.
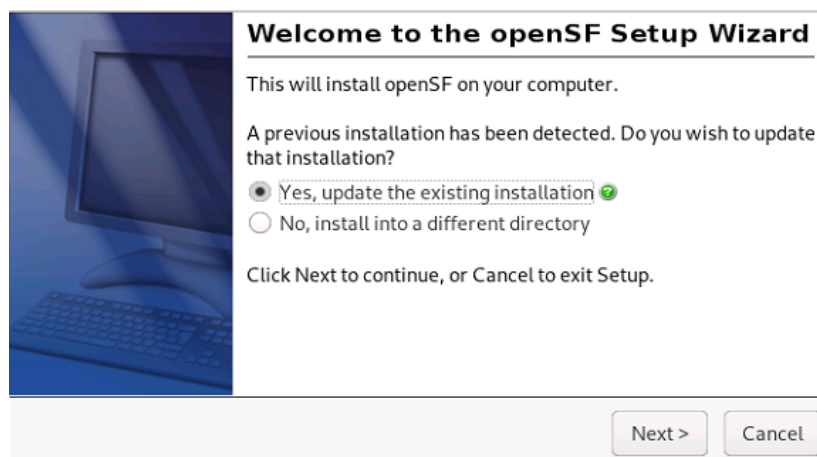
Figure 3-4: Installation confirmation screen

If an upgrade is possible and the user chooses to update the current installation, the installer will try to read the existing openSF configuration file, in order to keep the current settings in the updated system. This is normally automatic and transparent to the user. However, if the configuration cannot be read, a dialog warns the user and allows the installation to be cancelled before any changes are made. If the user nevertheless decides to continue with the upgrade, the new openSF will have default settings.

Once the installer has checked the system pre-requisites the user shall select the destination folder to hold the openSF software (Figure 3-5). The default installation folder is under the user home, so that the installation does not require any administrative privileges.

It is important to note that it is recommended to have a separate workspace folder from the program installation. The installer offers it, but if the user does not select one in the installer, openSF will ask the first time it is started. Refer to sections 4.2.1 and 3.3.1 for more details.
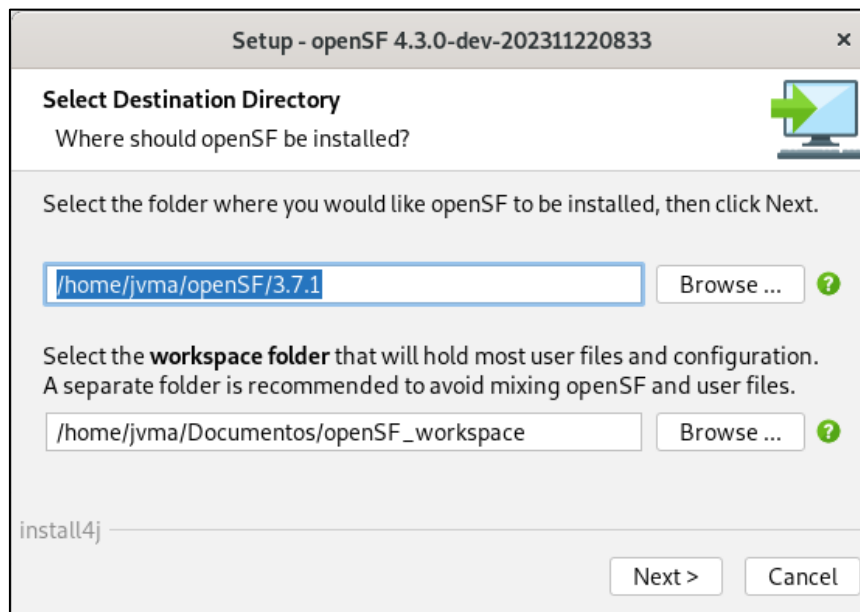
Figure 3-5: Installer folder selection window

In the next window users shall check that the information is correct and click next to proceed with the software installation (Figure 3-6).
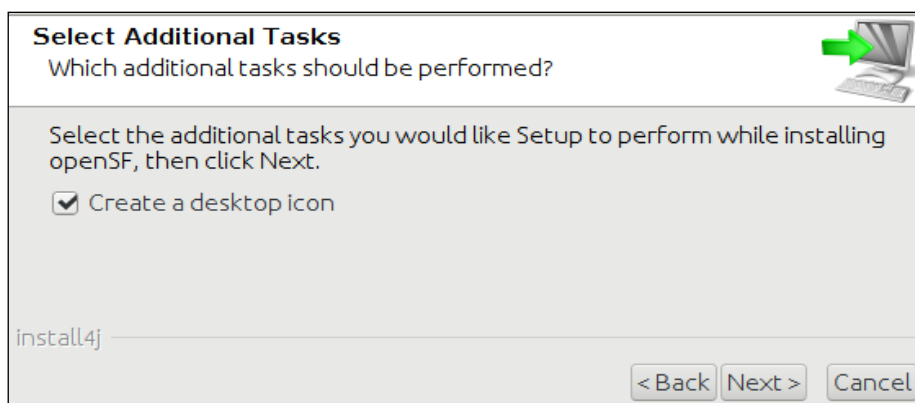
Figure 3-6: Installation icon window

If the installation process has been successful an "Install Complete" dialog will appear allowing to automatically launch the openSF software (Figure 3-7). To launch openSF at a later time, users may either user the openSF desktop icon[2], or open a terminal window, go to the openSF installation folder and run openSF manually. For further details on launching openSF refer to section 3.6.

---

[2] The openSF desktop icon is only available if the user so chooses during the installation process
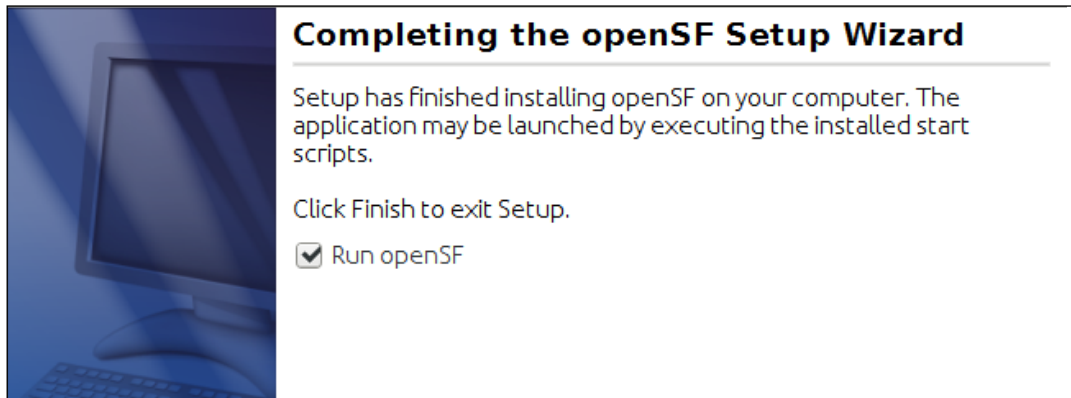
Figure 3-7: Installation successful screen

The installation package for macOS and Windows is also provided with a GUI installer, which in the case of macOS is inside a DMG archive. After launching the installer, the procedure to follow is the same as described above.

3.5.2. Uninstalling openSF

The installation process places an uninstaller application in the application's root folder, which launches a GUI uninstaller as shown in Figure 3-8.

Note that the confirmation requested to "delete user data under the installation folder" applies only to user files *inside the <OPENSF_INSTDIR> directory*. If selected, both the "simulations" folder and any H2DB files in the installation directory will be removed. User files elsewhere will not be affected even if the option is selected. For example, if the user moves the "simulations" folder, or creates a new workspace outside the installation folder, those files will not be affected by the uninstallation.
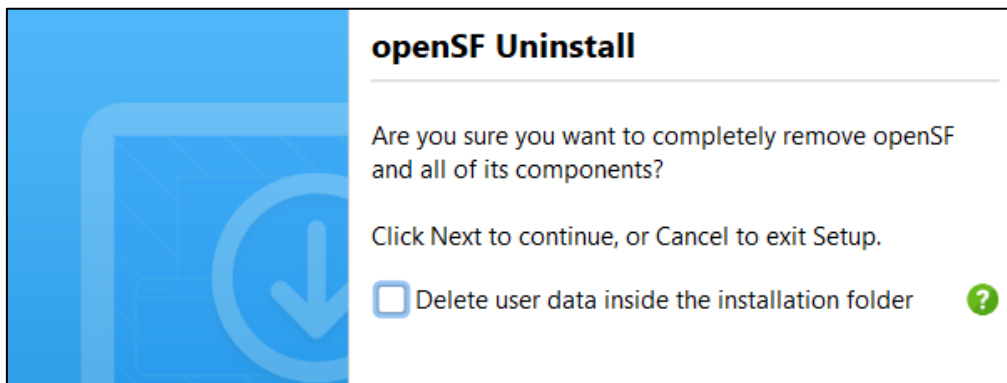
Figure 3-8: Uninstall confirmation screen under Windows

3.5.3. Licensing scheme

openSF uses a licensing scheme that allows integrating it in any kind of third-party developments. It is distributed under the terms of the "ESA Software Community Licence Permissive" as published by the European Space Agency; either version 2.4 of the License, or (at your option) any later version.

A copy of the "ESA Software Community Licence Permissive - v2.4" is distributed with openSF, or can be found at https://eop-cfi.esa.int/index.php/docs-and-mission-data/licensing-documents.

3.6. Running openSF

| I | A | E |

INDRA

openSF

System User Manual

OPENSF-DMS-TEC-SUM01

4.9

18/06/2025

3.6.1. How to start the application

The openSF system can be launched (under Windows, macOS and Linux) by: (a) double clicking on the openSF desktop icon[3] or (b) using a command line interface and executing the following command.

```
<OPENSF_INSTDIR>/openSF                                        (Linux)
open -a <OPENSF_INSTDIR>/openSF.app                            (macOS)
<OPENSF_INSTDIR>\openSF.exe                                    (Windows)
```

If openSF is launched with no parameters, the GUI will show up normally. This behaviour can be modified providing the following parameters:

- *--execute <simulation_identifier>*: The framework will launch the execution of a previously defined simulation with the stored parameter values. For example, to run in batch mode a simulation named "Radar", use the following command arguments:

  ```
  <OPENSF_INSTDIR>/openSF --execute Radar
  ```

  This form will execute openSF, find a simulation named "Radar" and execute it, intercepting all the events and storing the results in the database. Then, the system will stop.

  Note that running long simulations via the command-line interface is discouraged, especially over a remote connection like SSH. In case that running a long simulation over a remote connection is necessary, a detachable GUI connection like VNC or RDP should be used instead.

- ❏ --dbCfg <db_config>: Use the given database settings, in the same format that is stored in the configuration file[4]. If absent, openSF will use the connection that was last used successfully.

  For example, the following will run openSF with a file-based database named "filedb" directly under $E2E_HOME:

  ```
  <OPENSF_INSTDIR>/openSF --dbCfg 'h2$$filedb'
  ```

Note that command-line options **must** use double dashes. Since openSF is built on the Eclipse RCP technology, options with a single dash *may* be intercepted by the Eclipse launcher code, not reaching the openSF application code at all. In particular, Eclipse launcher options that may be used as well include:

- ❏ -consoleLog: Sends any log output to the command shell. This can be useful in case of an openSF misbehavior
- ❏ -noSplash: Prevents the splash screen from being displayed.
- ❏ -data <location>: Sets the workspace location for this session. See section 4.2.1.1 for details.

Please refer to the Eclipse official website[5] for the full list of options.

Furthermore, when running openSF on a Linux system with Wayland-based graphics, it is possible to experience crashes and graphical glitches. In that case, launch openSF with the environment variable "GDK_BACKEND" set to the value "x11", for example by calling it as follows:

```
GDK_BACKEND=x11 <OPENSF_INSTDIR>/openSF [args…]              (Linux)
```

Note that both name and value of the environment variable are case-sensitive.

The graphical simulation viewer may also fail in GPU-constrained environments, which may include remote desktop connections. If the program crashes when first opening or launching a simulation, the user can try running openSF with the following additional command line arguments:

```
<OPENSF_INSTDIR>/openSF -vmargs -Dprism.order=sw
```

---

[3] The openSF desktop icon is only available if the user so chooses during the installation process

[4] Note that the shell syntax may require escaping certain characters in this string, e.g. in Linux or macOS it may be necessary to escape "$" characters, or quote the entire string.

[5] Eclipse documentation under help.eclipse.org, topic "The Eclipse runtime options"

3.6.2. First start-up

The framework requires both a valid workspace and database for seamless functionality. For this purpose, when the user first starts the application, or if no "last used database" is defined in the configuration, openSF is opened in a temporary state that allows the user to quickly create a database with a default name. This is offered via the first-start welcome dialog depicted in Figure 3-9.

In addition to the database configuration, it's essential to note that a valid workspace is also crucial for the proper operation of the framework. If the workspace selection was left blank during the installation process, or if upgrading from a version of openSF that utilized the program installation folder as the workspace, the user will be prompted to select a workspace folder. This ensures that the framework has a designated location to store and manage workspace-related files. For more details on this process, refer to section 4.2.1 of the documentation.
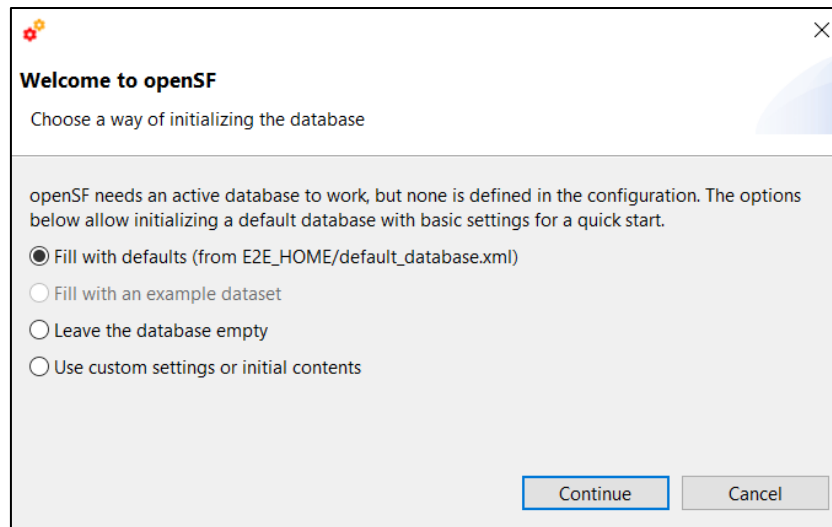
Figure 3-9: First start dialog

The top three options all create a database with an autogenerated name, with different initial contents: the first with data from E2E_HOME/default_database.xml; the second with the example simulator package distributed with openSF; and the third with no data whatsoever.

The last option instead opens the "databases" dialog, allowing the full range of settings (name, location, initial dataset) to be used. See section 4.2.2 for more details on the advanced procedure.

3.6.3. Check for updates

openSF performs an automatic check for new versions by connecting to a remote server. In case a new version is identified the user is given the choice of downloading the software. Afterwards the user can upgrade the software version following the standard installation procedure for openSF software.

Make sure to back up your workspace folder before the upgrade, as it will be upgraded once the newer version uses it, with no guarantee that it can be opened without issues by the old version.

This check can be disabled if an upgrade is not desired – note, however, that running the latest version of the framework is always recommended, since old versions are not supported.
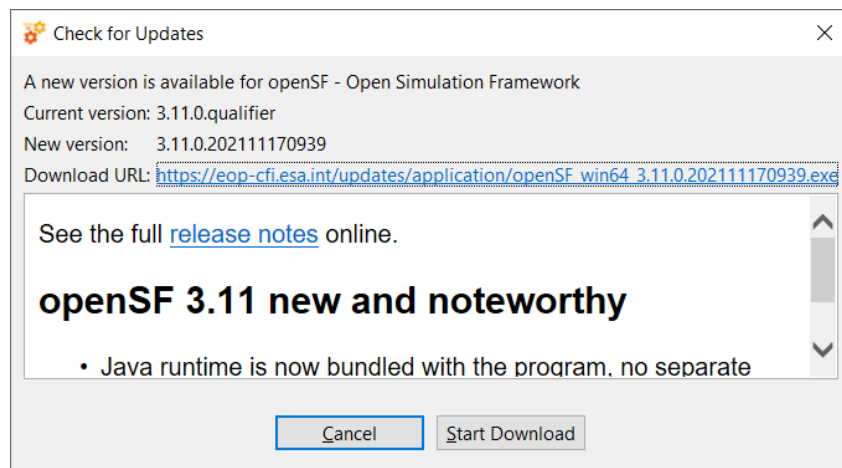
Figure 3-10: Dialog shown when an update is available

3.6.4. Exit the system

Upon the selection of this function, openSF will inform the user whenever a simulation is executing. Upon user confirmation, openSF will stop every internal process (including on-going external modules) and will end its execution. This is the recommended way of ending the application.

4. REFERENCE MANUAL

4.1. HMI Description

| I | A | E |

In this section the look-and-feel, operational behaviour and design features of openSF HMI are presented.

4.1.1. Main window

The HMI accepts inputs via devices such as the computer keyboard and mouse and provides articulated graphical output on the display. The HMI has been designed to be flexible, to let users organize the layout of the information as desired, showing only relevant windows and in the way users want. The layout consists of a main container that can host inside several internal frames. These internal frames are intended to present independent modules of openSF. For example, each time the user wishes to perform operations with the list of modules in the repository, a module manager frame will pop-up inside the bounds of the main window listing the list of modules currently available within openSF.
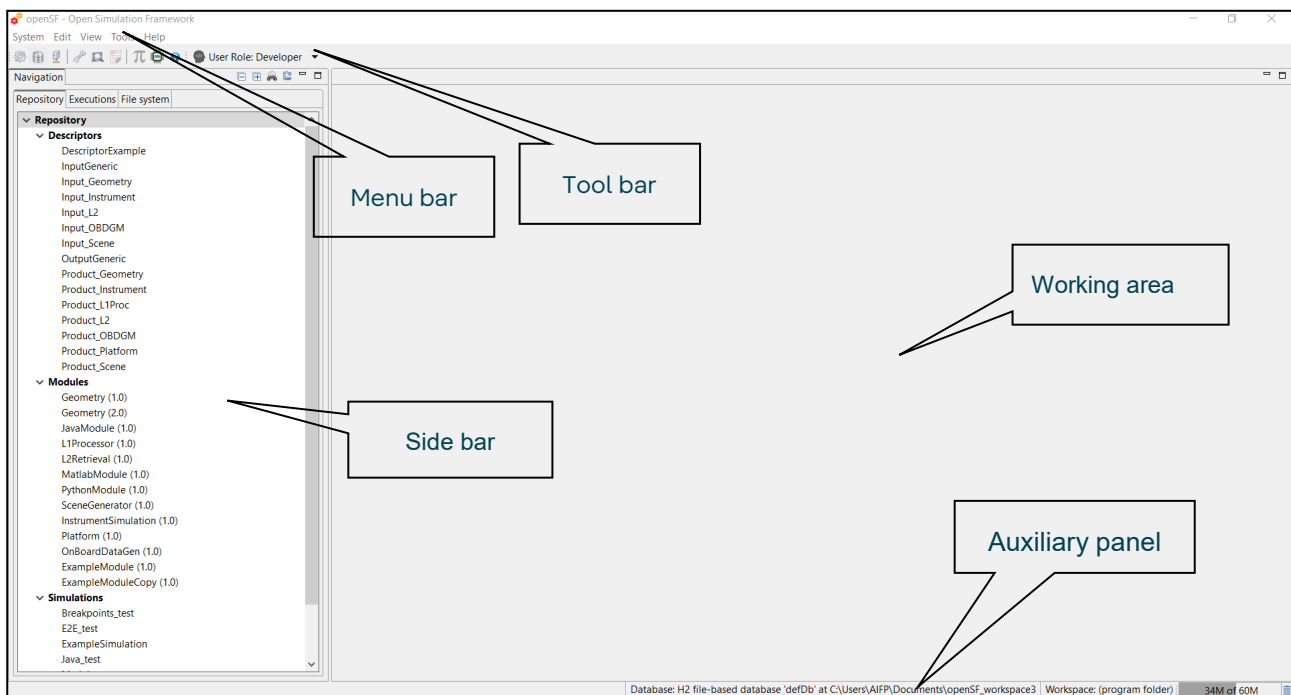
Figure 4-1: Main window appearance

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

This main window shown in Figure 4-1 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.

Occupying the central and main region there is a working area. This area is where all internal frames are going to be created and the main interaction with the user is held. On its side, 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 the system objects navigator, a "side bar" aiming to provide a quick access handler to every item defined in the system: repository of descriptors, modules and simulations, the list of simulation execution results and also a file system browser to navigate through the contents of the application's directory. Section 4.1.1.1 deeply describes this part.

The main window's footer area shows application status information.

The HMI provides a menu bar (Figure 4-3) at the upper side of the main frame to show some capabilities of the system. Below the menu bar there is also a toolbar to quick access critical functionalities (Figure 4-2).
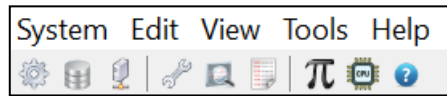
Figure 4-2: Detail of main menu bar
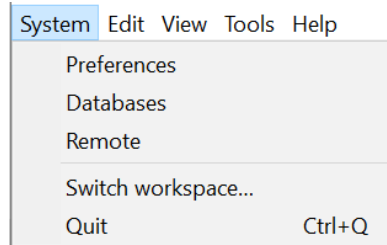
Figure 4-3: Detail of a menu, showing menu items

Figure 4-3 shows that a menu item is composed of the name of the function and a quick access key combination. Users can quickly access this functionality pressing this key combination or the first letter 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[6]. These pop-up menus have the same appearance of the menus rolling down from the menu bar. Here, icons are added at the left of the function names that graphically describe them.
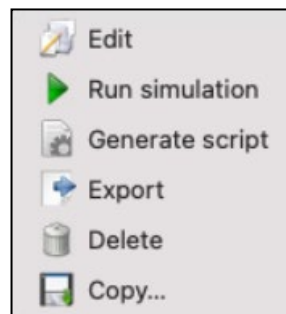
Figure 4-4: Detail of a contextual menu

It can be seen in Figure 4-4 that a pop-up menu acts exactly like a menu at the main frame. They also provide mouse and keyboard access to certain capabilities.

4.1.1.1. *Side bar*

On the left side of the main frame there is the side bar, grouping different views of the openSF areas: Repository, Executions, and File system.

As can be seen at the right-upper corner of Figure 4-5, standard buttons are available to minimize or maximize the side bar. The side bar can also be dragged to dynamically change its width.

---

[6] In macOS the contextual menu behaviour may depend on setting the correct gesture for Bluetooth mouse and track pad: the "Secondary click" gesture (e.g. "Click with two fingers" in track pad or 'Click on right side" in Bluetooth mouse) should be applied to allow using the options in the contextual menus.

The repository will show only the elements containing a given substring of characters written in this text box.

Elements are structured into the repository by element type (descriptors, modules and simulations).

Navigation
Find:
Repository Executions File system
∨ **Repository**
∨ **Descriptors**
InputGeneric
Input_Geometry
Input_Instrument
Input_L2
Input_OBDGM
Input_Scene
OutputGeneric
Product_Geometry
Product_Instrument
Product_L1Proc
Product_L2
Product_OBDGM
Product_Platform
Product_Scene
∨ **Modules**
Geometry (1.0)
Geometry (2.0)
JavaModule (1.0)
L1Processor (1.0)
L2Retrieval (1.0)
MatlabModule (1.0)
PythonModule (1.0)
SceneGenerator (1.0)
InstrumentSimulation (1.0)
Platform (1.0)
OnBoardDataGen (1.0)
∨ **Simulations**
Breakpoints_test

Figure 4-5: Side bar

Into the Repository tab of the side bar, users can find a tree-like structure containing all the known modules, descriptors and simulations.

This tree-like structure can be collapsed or expanded by clicking in the arrow-shaped icon.
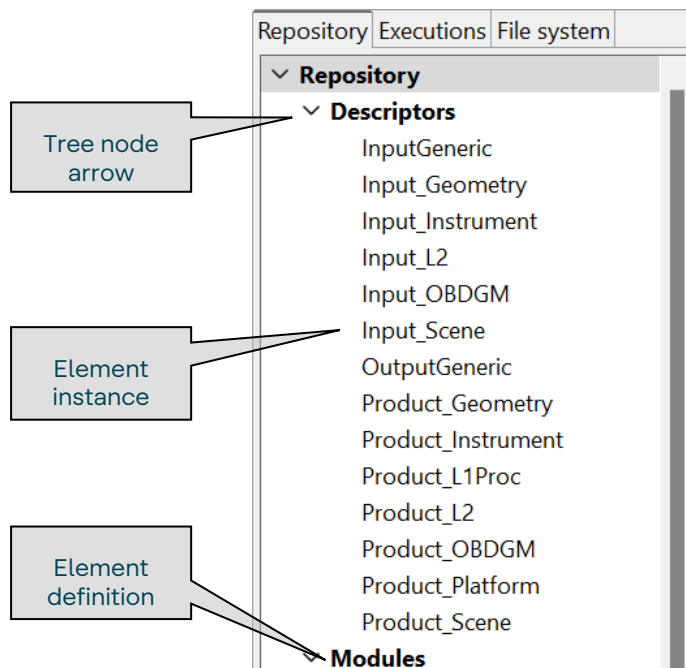
Figure 4-6: Repository view

Every row marked with an arrow-shaped icon represents and element definition of the repository. Every row without icon represents an element instance. Right-clicking over both of them, a menu pops up containing some associated commands. These menus are context-sensitive, meaning that different types of elements have their own associated commands. These commands are going to be explained in detail in each element's section (sections 4.3.1, 4.3.2, 4.3.3).

A left double click over the elements will activate the first associated command of the menu (typically, editing).

The last tab in the side-bar (i.e. File system) is a browser to easily access both the folder structure under the openSF installation directory and the simulations execution directory.

Organized in the tree-like structure, the user can easily locate every needed file. This structure is refreshed every time an operation involving files is performed or when the user presses the "refresh" button.

The File System tree always shows the openSF workspace as its root node. If the simulations execution directory is located outside of it (as selected by the user from the preferences page, see section 4.5.3), another root-level node is added for it. Regardless, the view automatically expands the tree to reach the simulations directory and sets focus on it. See Figure 4-7.

The File System tree uses a colour code to ease finding the resulting files of the executions. If a node represents a folder containing the results of an execution which is also present in the currently selected database, that node, its parents and all its children are rendered in the default system colour (e.g. black). By contrast, if a tree node represents either a file or a folder that does not match an execution result in the currently connected database, the tree node is rendered in a "disabled" colour (e.g. light grey).

Double-clicking on any file or folder opens it using the system default application.

The Find function in this view works by filtering the tree rather than selecting a matching item. An item is visible in the tree if its name or the name of one of its children matches the filter text.
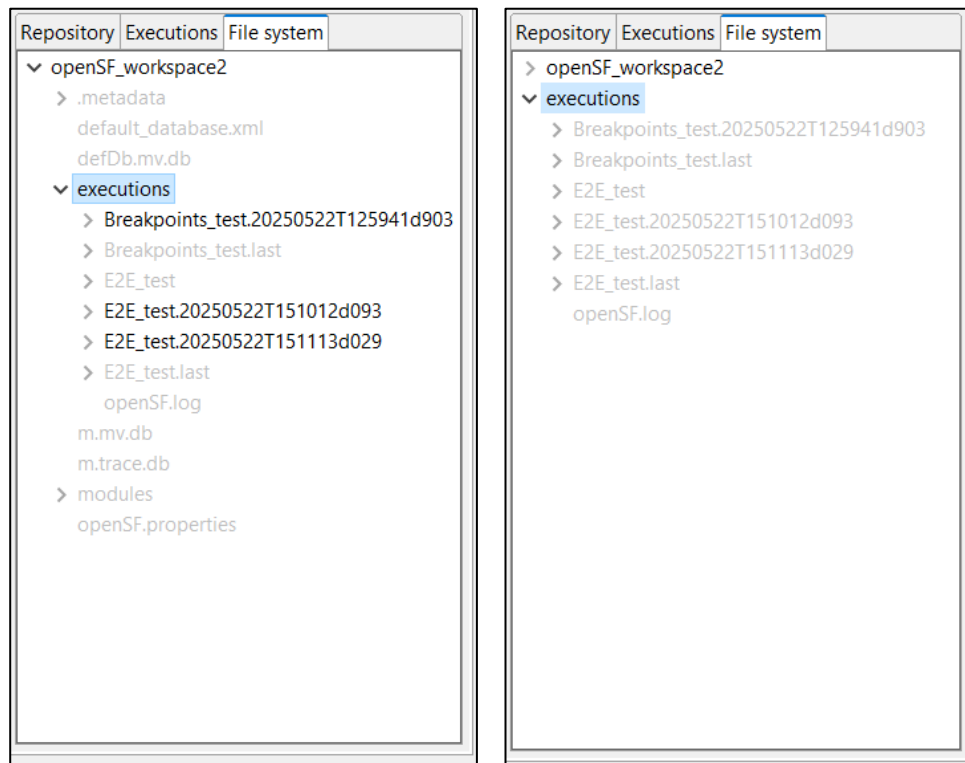
Figure 4-7: File system view with an executions folder inside (left) and outside (right) of the workspace

4.1.2. Frame management

Accessing the "View" menu of the main menu bar the user can find all the functionality provided for the frame management. Option 'Reset Views' allows restoring the original window layout (as defined on installation).
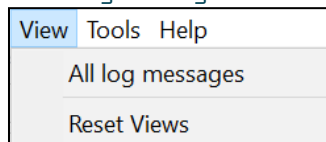
Figure 4-8: Frame management menu

Other frame management functionalities can be found in the header of every main frame (hence not in dialogues).
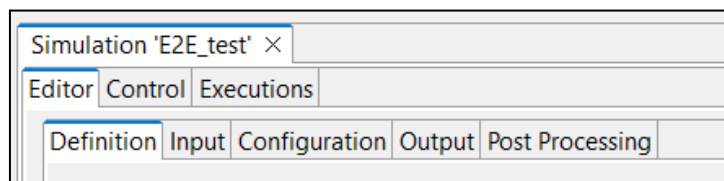
Figure 4-9: Internal frame header

Note the two little icons at the right border of the header (Figure 4-9), to "minimize" and "maximize" the frame.

If the user minimizes a frame, it disappears from the working area but it can be restored from the "available frames toolbar" visible at the side of the main frame.

4.1.3. Generic functionalities, dialogues and displays

This section is meant to describe the design of HMI generic functionalities, dialogs and displays.

There are some functionalities of the HMI that show a "file chooser" dialog as shown in Figure 4-10.
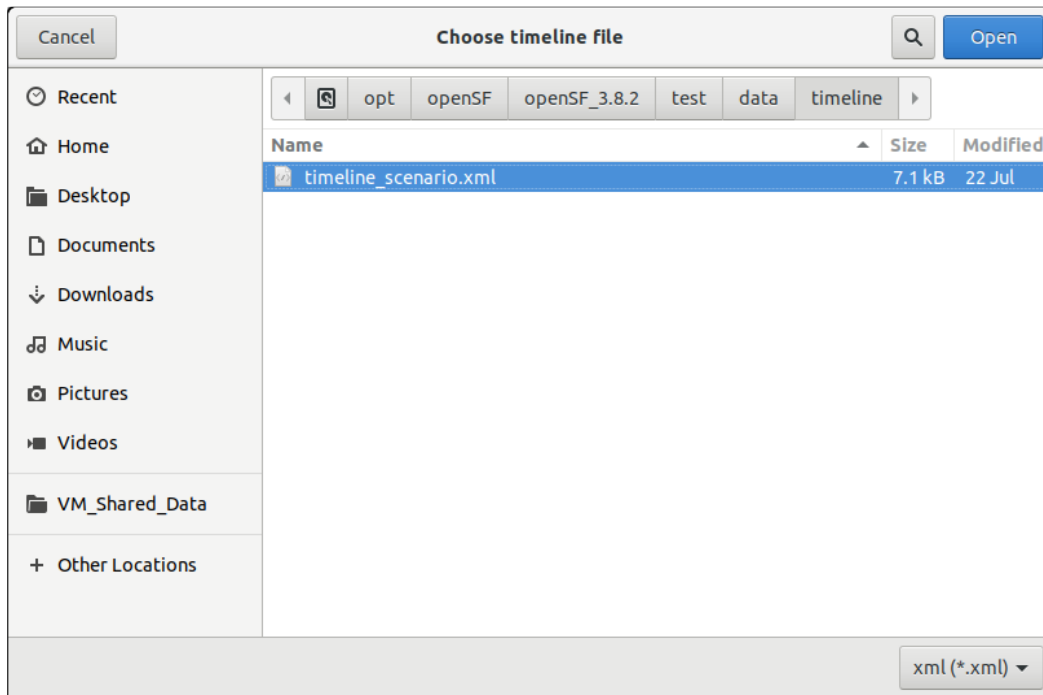
Figure 4-10: File chooser dialog

This dialog allows the user to browse the system directory to select a certain file or list of files. It provides sorting, filtering and file operations.

Throughout the openSF HMI some functionalities may show information to the user and might ask for some input in response to a specific question. The HMI will present modal dialogs that will get the system focus until the user provides an answer. These modal dialogs will block the input to other areas of the application until a response is given.
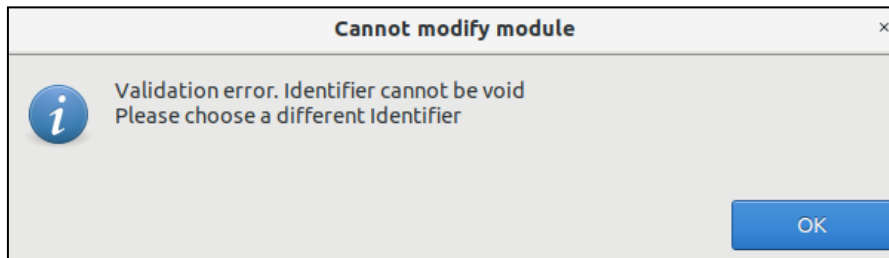
Figure 4-11: Dialog example

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

4.2. Data Structure

| I | A | E |

Most information systems must store information in a persistent way. openSF trusts a relational database to store structural information and the file system to store the input/output/configuration files. The following table shows which openSF elements are stored in the database and which into the file system.

Table 4-1: openSF information management system

Element	Storage
System.Configuration	File system. <workspace_root>/openSF.properties

System.Tools	Database
Repository.Descriptors	Database
Repository.Modules	Database
Repository.Simulations	Database
Repository.Simulation script	File system. <simulations_folder>/<sim_id> /<name>.sh, where the simulations folder is that defined in 4.5.3.
Executions.Results	Database
Executions.Logs	File system: <simulations_folder>/openSF.log (global) File system: <simulations_folder>/<sim_id>/log/simulation.log (for each simulation) File system: <simulations_folder>/<sim_id>/log/MODULE.log (for each simulation and module, only if enabled in application settings)
Execution.Dumped log simulation	File system
Execution.Input/output/configuration files	File system <simulations_folder>/<sim_id>/<filename>

The folder structure of openSF has already been presented in section 3.3.1.

In the openSF database, the following string types are used:

- Short string: 25 characters
- Normal string: 75 characters
- Medium string: 255 characters

4.2.1. Workspaces

Since the framework may be installed on a shared folder that is not writable by the user, openSF stores its configuration under the <workspace> folder (see section 3.3.1). The main window shows the current database and workspace in the status bar at the bottom:

Database: H2 file-based database 'defDb' at (E2E_HOME) | Workspace: (program folder) : 47M of 99M

Figure 4-12: Current database and workspace indication

It is strongly recommended to use a separate workspace directory. This practice is advisable as it helps avoid the mixing of openSF and user data, contributing to a more organized and efficient system.

Although the program folder can be utilized as the workspace (if it is writable), especially for updates from versions where this was the default behaviour, this approach is now considered deprecated and the option could be removed in future versions. In the interest of best practices and future compatibility, users are encouraged to transition to using dedicated workspace directories See section 4.2.1.1 for details on changing the workspace.

4.2.1.1. Switching workspaces

The user may change the workspace folder from within openSF by using the menu item "System » Switch workspace". This action invokes the workspace selection dialog (as shown in Figure 4-13), which offers a selection of folders recently used as a workspace. If a new workspace folder is selected, openSF restarts immediately using the new <workspace> setting.

If the selected folder does not exist, the dialog will offer to create it. If the user enables the "store and do not ask again", the new default workspace is persisted in a special configuration file outside of any workspace (see section 4.2.1.2), so openSF will use the same folder until the setting is changed again.
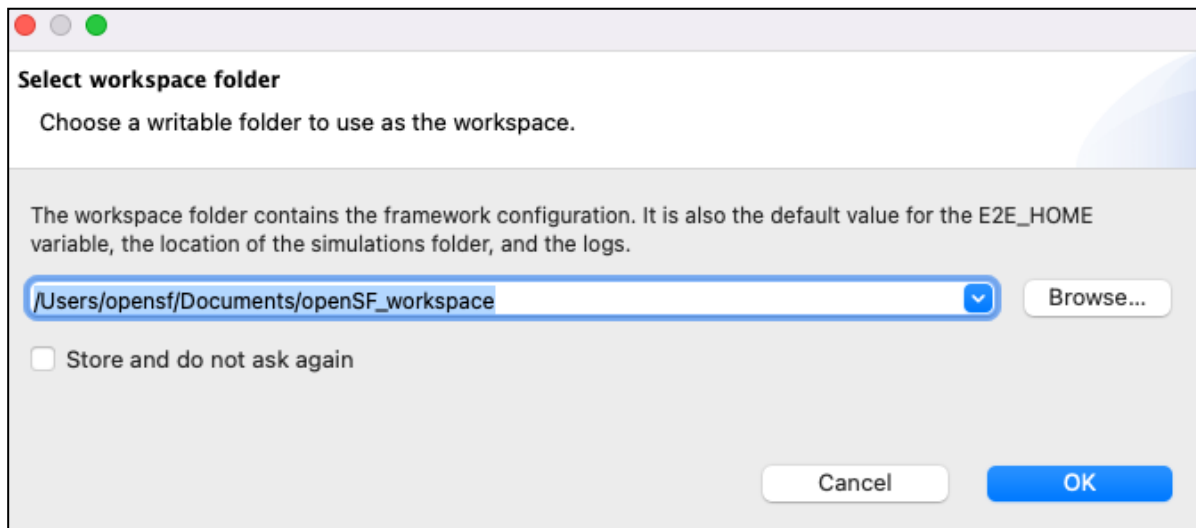
Figure 4-13: Select workspace dialog

In addition to using the dialog, openSF can be started with a specific workspace folder by using the "-data" command line option, as follows:

```
<OPENSF_INSTDIR>/openSF –data /path/to/writable/workspace/folder
```

4.2.1.2. Configuration persisted across workspaces

The framework stores the workspace selection settings in a file "workspace.properties" under the path "$XDG_CONFIG_HOME/openSF". If this variable is undefined, the default value of this cross-workspace configuration path varies per platform:

- ❑ In Linux: "$HOME/.config/openSF"
- ❑ In macOS: "$HOME/Library/Application Support/openSF"
- ❑ In Windows: "%LOCALAPPDATA%\openSF", if it is defined. Otherwise, an equivalent to the path used in Linux: "%USERPROFILE%\.config\openSF".

This location contains only the recent/default workspace configuration. It is shared by all openSF installations in the system, although the data in the file is keyed to each specific installation. Thus, even though the file is shared, openSF installations in different folders (e.g. simulators from different projects) each have their own different default workspace.

In addition to this cross-workspace file, there may be a file "openSF_defaults.properties" inside the program installation folder <OPENSF_INSTDIR>. If it exists, it is used to provide default values for the configurations stored in the respective "openSF.properties" files in each workspace. This may be used, for example, by an integrator in order to ensure that any new workspace created has suitable settings, see section 7.4.

4.2.2. Databases

Within each workspace, openSF has the ability of working with multiple repositories. This implies that users can create different databases, so that all of them are independent. Users can work with different input data by selecting only the corresponding database in the repository. If the user changes e.g. the modules that make up a simulation, only the database that is loaded in that moment is modified, as all databases are independent. Thus, openSF can hold more than one database in one single instance.

Databases may be either server-based or file-based.

When the user selects the "Databases" option from the menu "System", the window presented in Figure 4-14 will show up.
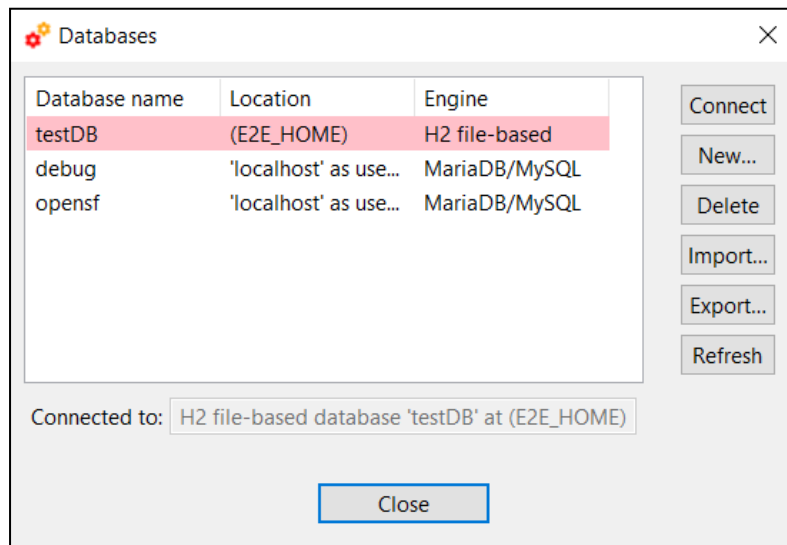
Figure 4-14: Database management window

At the bottom of the window there are seven buttons, which allow users to perform different actions over the databases. Users can connect to an existing database, create a new database, delete an existing database, import database definitions specified in XML format, export database definitions in the same format or add databases residing in the same server/folder to the list.

The central area of the window shows a list with the stored databases. Along with the database name, this area shows the database location and engine. The location could be either a system path (in case of a file-based database) or a set of address and user (in case of a server-based database).

The bottom of the window shows a label with the database currently in use by openSF. In this case, the application is connected to the 'testDB' database, so all configuration associated with its descriptors, modules and simulations is loaded by the system.

4.2.2.1. *Database version mismatch and upgrade*

When a user attempts to connect to a database that was created or last accessed with a different version of openSF, there could be compatibility issues or missing features. To address this issue, openSF prompt the user with an upgrade notification when attempting to connect to a database with a different version, as it can be seen in Figure 4-15. This prompt would inform the user about the version disparity and provide options for upgrading the database to match the current version of openSF to ensure compatibility and optimal performance. Note that the same dialog is shown from the databases view and during initialization.
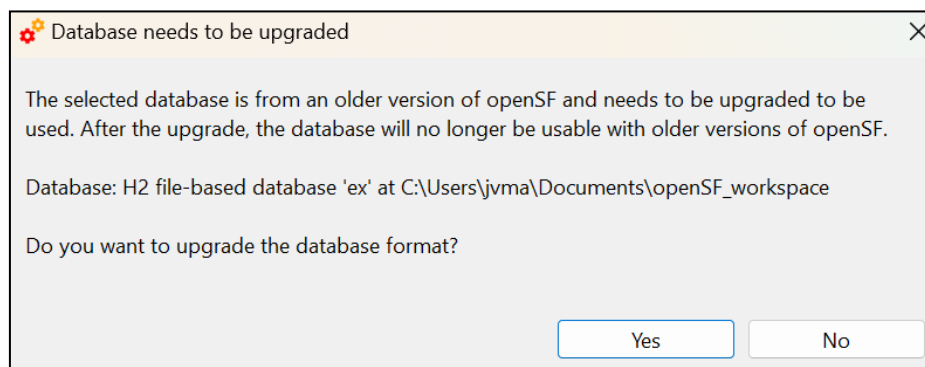
Figure 4-15: UI upgrade prompt.

In the event that the database needs an upgrade and the user rejects the upgrade prompt, the handling will be slightly different depending on whether the rejection occurs within the databases view or during initialization.

If the user rejects the upgrade while in the databases view, the application will simply return to the databases view without modifying the database. A normal error dialog will be shown to inform the user of the rejection, providing any necessary details or instructions.

If the user rejects the upgrade during initialization (when first launching the application), the application will display a normal error dialog indicating the version mismatch. Subsequently, the application will redirect the user to the databases view, allowing them to create a new database for this workspace if they wish to proceed with a version that matches the current openSF application. This case is shown in Figure 4-16.
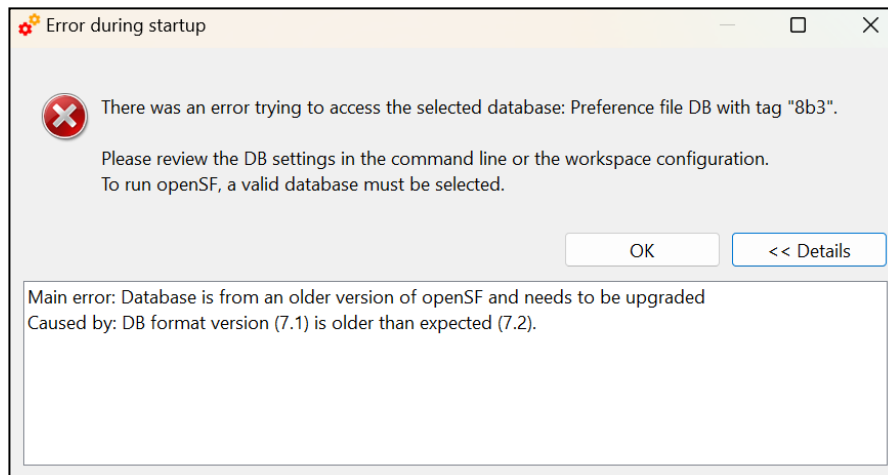
Figure 4-16: UI upgrade rejected during initialization.

In the opposite situation, when the database is too new, error dialogs will be displayed, indicating that the database is from a newer version of openSF and that version is not compatible with the current version of the openSF application. In this case, the newer database cannot be downgraded, and can only be opened with the corresponding version of openSF.
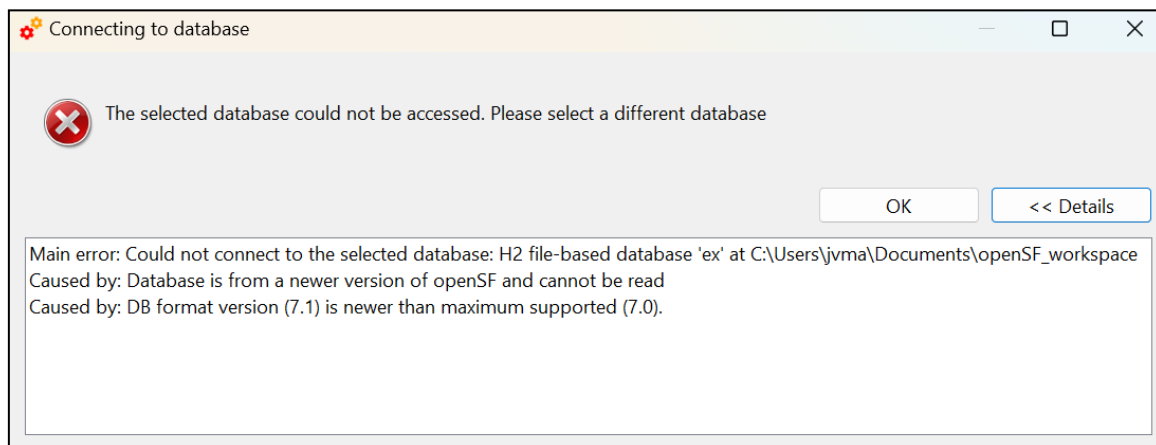
Figure 4-17: Error shown if a database is too new

4.2.2.2. Connect to a database

Users can switch between the different databases located in the DB server. For this, the user has to select a database from the list, and click on the "Connect" button. Automatically the system is connected to it, and then, the name of the selected database is shown beside the label "Connected to". An example of this procedure is shown below:
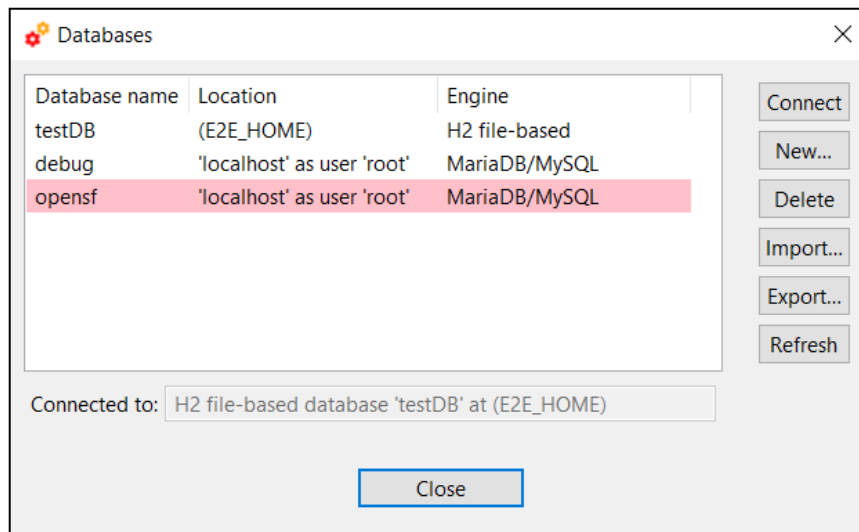
Figure 4-18: Connect to a database

The window shows that the system is connected to the "testDB" database. If the user selects the "opensf" database from the list, and clicks on the "Connect" button, the system automatically connects to this database, as it is shown in the corresponding label. If an error occurs during the procedure and the connection cannot be performed, openSF shows a message reporting the error.

It is also important to note that while a simulation is running, connecting to another database is not allowed. In addition, if any view is open while switching databases, it will be closed.

4.2.2.3. *Create a new database*

If the user wants to create a new database, he has to click on the 'New' button, and a dialog will be shown by the application, as it can be seen in Figure 4-19.
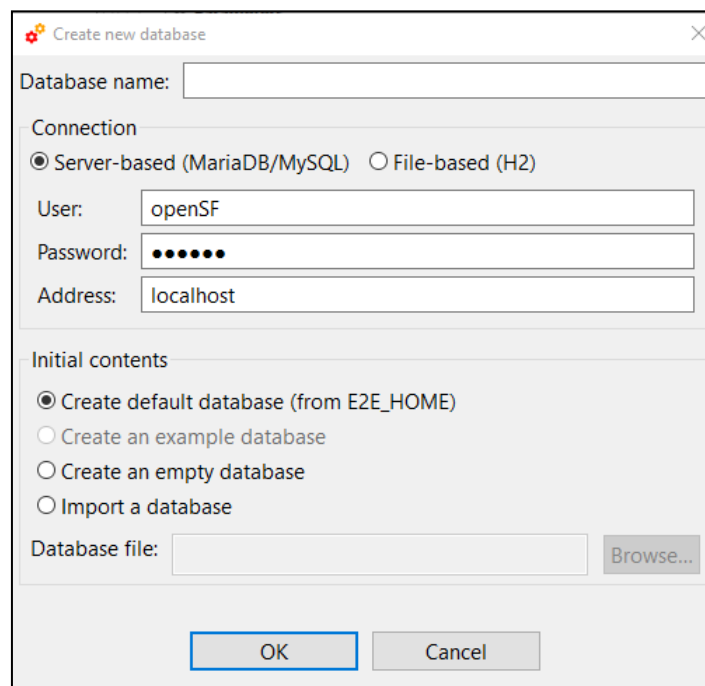
Figure 4-19: Create new database

The user can now choose the database name and type. Depending on the type, the user shall fill the remaining fields. For a server-based database (as shown in Figure 4-19) the fields are:

- **User and password**: Credentials for the user that connects to the databases server (requires privileges to create a database);
- **Address**: Hostname (and optionally port) where the database server is located;

For a file-based database, the only field is:

- **Folder**: The system folder in which the database file is going to be saved. Paths located inside the workspace folder will be stored as relative to it.

Some fields are presented with a default value e.g. if the system is currently connected to a server-based database, the default is to create a new server-based database using the same server and credentials.

All fields are mandatory except the server-based password, which may be empty.

In the lower part of the window, the user can choose the initial contents of the database:

- Default database: from the default file under E2E_HOME (see section 3.3.1 for more details)
- Empty database: no elements, just the basic structure
- Example database: containing example modules and simulations
- Importing another database XML representation

When the user enters all the information correctly and clicks on the 'OK' button, the new database is created, and openSF is automatically connected to it.

In case some field has been entered incorrectly (as for example an already existing database name, or the user or password to connect with the server are incorrect) openSF shows a message reporting the error as shown in Figure 4-20, and force the user to enter the correct information.
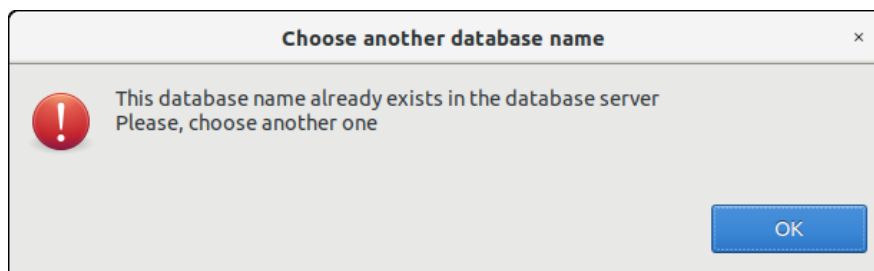
Figure 4-20: DB creation error message (wrong name)

If the user clicks on the 'Cancel' button on the new database window, no action is performed.

4.2.2.4. *Delete a database*

To remove a database from the databases server, the user has to select the database to remove from the list, and click on the 'Delete' button, as shown in Figure 4-21.
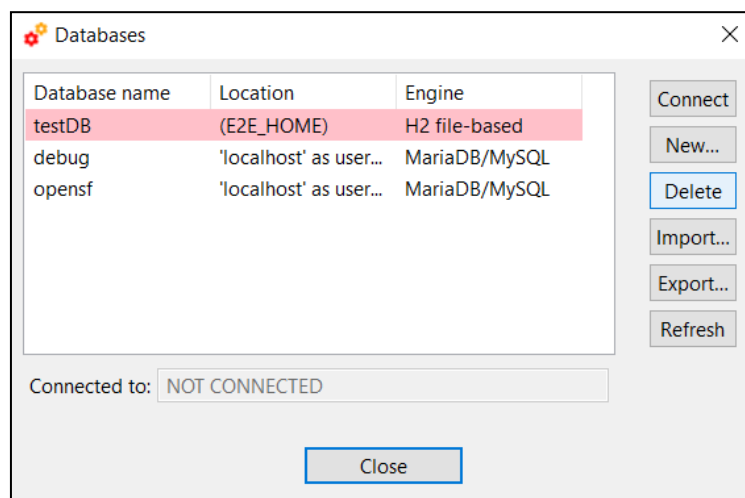
Figure 4-21: Delete a database

A new dialog is shown to confirm the action. If the user clicks on 'OK', the database is deleted.
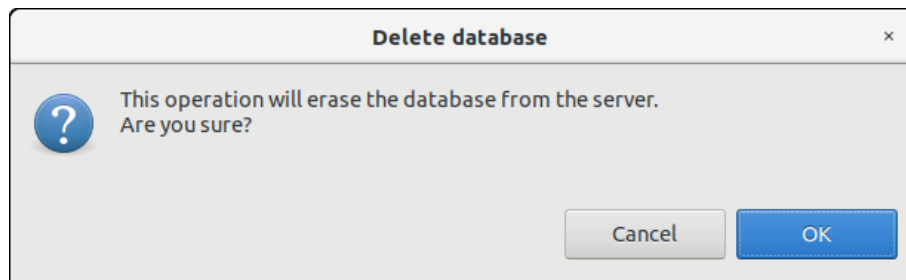
Figure 4-22: Confirm deletion operation

openSF needs to be always connected to an existing database. Thus, the user cannot delete the connected database otherwise the system will send an error message. Therefore, the user must connect to another database to delete the current one.

4.2.2.5. Import and Export a database

The capability to import and export elements definition specified using XML (refer to [AD 1] for the definition of the expected XML format) is accessible from the Database Management window, presented in section 4.2.2.

To import a database, the user shall click on the "Import" button appearing at the bottom of the window and select the XML file containing the database definitions. Note that exported databases in XML format exported by older openSF versions can be also imported, or used to create a new DB. They will be automatically upgraded on the fly, without modifying the original file. The procedure may require access to the simulator files, so ensure that any paths are valid (e.g., that the simulator is unpacked where the DB file expects it). Any issues or warnings will be shown in a dialog when the import process is complete.
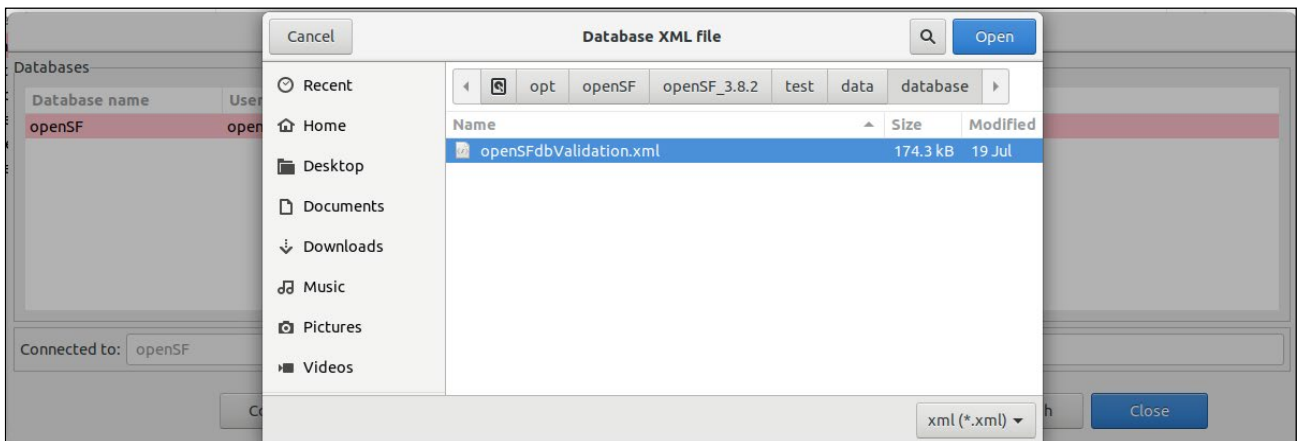
Figure 4-23: Database import

The outcome of the operation is that the elements included in the XML file are introduced in the currently selected DB. Note that importing the database will convert the paths to the native format.

To export a database, the user shall select a database from the list and press on the "Export" button. The user can now select the desired file path where to save the database. In addition, the user can choose between two export options: export the simulation skeleton but not its executions or export the entire database. If the exported database contains non-portable paths a warning message will show up. Such paths might cause problems if the user wants to use the exported database on other machines. Examples of non-portable paths are absolute paths, or paths that contain characters that might be invalid in other platforms.
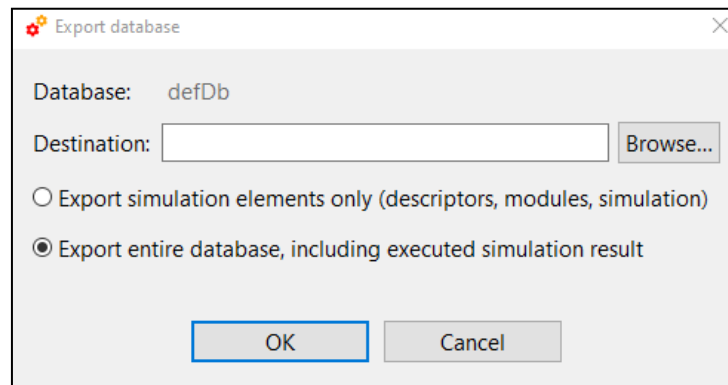
Figure 4-24: Database export

4.2.2.6. Refresh database list

The database management dialog provides the capability to refresh the list of openSF databases available in the server or folder to which openSF is connected to. This functionality identifies among the available databases the ones that are compatible with openSF. This capability is applicable in situations as manual migration of openSF database or automatic database upgrade, both occurring typically when upgrading to a more recent version of openSF.

4.2.2.7. Database maintenance

Currently, openSF can connect to a local or remote server. If the last database to which openSF was connected is not accessible during openSF initialization, openSF will start in a temporary state described in section 3.6.2 that will allow the user to connect to a valid database.

The user (or the database server administrator) is responsible to regularly back up, de-fragment, clean and perform similar maintenance operations to guarantee the database integrity. In case of a MySQL or MariaDB database, users can execute the following script to perform a manual backup of the openSF database, which can later be restored using the same utilities:

```
~/openSF$ mysqldump --user=USER --password=PASS DB_NAME > openSFdb.bk.sql
```

4.2.3. Simulation Results Naming Conventions

openSF naming convention for simulation execution directories, as well as their relative supporting files, involves the use of names with a timestamp. The use of a timestamp is meant to ensure a unique identification of the simulation folder and files.

In case of a nominal simulation (single execution) the execution result (whether it was successful or not) is stored in a <simulations> folder, named as "<simulation_id>.<starting_time>". Starting time is coded as "YYYYMMDDTHHmmSSdsss"[7] in local time – see the format in Figure 4-25.

In case of timeline-based and iteration/perturbation-based simulations, openSF groups related simulations execution results into separate output folders (since these two types of executions can end up to hundreds of simulations). Also, in these two cases, in order to uniquely identify the executions, a timestamp is appended to the execution identifier, as shown in Figure 4-25.

Note that the prefix "exec#" is used for each iteration folder as a way of facilitating the user's distinction between Timeline and Iteration/Perturbation executions.

Summarising, the naming scheme for simulation result folders is as follows:

- Nominal simulation (single execution)
  - *<simulation_id>.<start_time>*, where
    - *simulation_id* is the simulation name (as seen in the "Repository" view)
    - *start_time* is the simulation execution start time, formatted as YYYYMMDDTHHmmSSdsss

---

[7] "sss" denotes milliseconds

- Iteration/perturbation-based simulation (multiple executions, based on the specification)
  - *<simulation_id><start_time>*/exec*#.<start_time2>*, where
    - *simulation_id* is the parent simulation name (as seen in the "Repository" view)
    - *start_time* is the simulation execution start time, formatted as YYYYMMDDTHHmmSSdsss
    - *#* is an incremental integer defining the order of execution of the multiple executions
    - *start_time2* is the iterated execution start time, formatted as YYYYMMDDTHHmmSSdsss
- Timeline-based simulation (multiple executions, based on time segments specified)
  - *<simulation_id><start_time>*/*<time_segment_start>.<start_time2>*
    - *simulation_id* is the parent simulation name (as seen in the "Repository" view)
    - *start_time* is the simulation execution start time, formatted as YYYYMMDDTHHmmSSdsss
    - *time_segment_start* is the time segment start time (as defined by the user)
    - *start_time2* is the time segment execution start time, formatted as YYYYMMDDTHHmmSSdsss
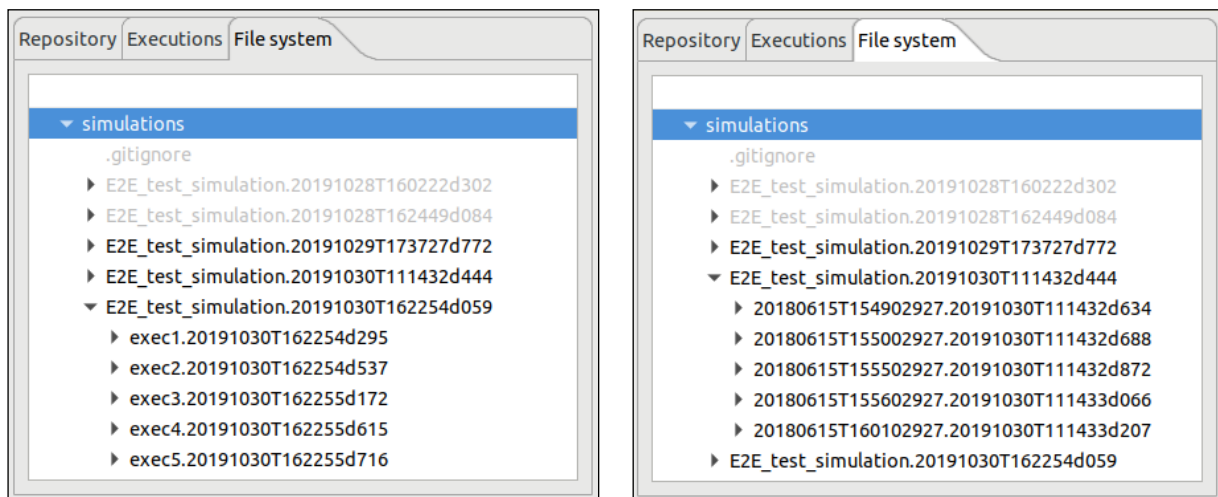
Figure 4-25: Grouping of iteration/perturbation (left) and timeline (right) simulations

In order to simplify simulation results directory names, symbolic links are used. Each time a simulation is executed a symbolic link is generated in the file system with the name of the simulation being executed (appended by ".last") and pointing to the corresponding simulation execution directory. Each time a simulation is re-run the symbolic link is re-generated pointing to the latest simulation execution (the one with the latest timestamp). Please note that these symbolic links are not generated in Windows. Therefore, a warning is not thrown when the symbolic link is not generated. The contents of openSF installation directory and the simulation execution ones can be reviewed in the File System tab of the side bar, see section 4.1.1.1.
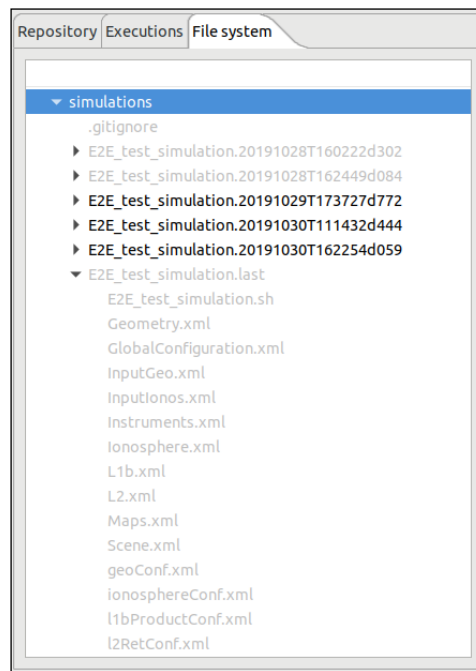
Figure 4-26: File system in the side bar, including symbolic link to last simulation

4.3. Framework Elements

M	I	A	E

In this section the elements at the base of openSF are described. They are collectively called the "domain" elements of openSF.

4.3.1. Descriptors

M	I	A	E

openSF has the possibility to define the set of input and output files (called descriptors) used to connect different modules in simulation runs.

Users can access the list of nominal descriptors (those provided in the default distribution) inside the repository view of the side bar, as seen in Figure 4-27.

Accessing the corresponding menu of the main menu bar (Edit → Elements → Descriptors or Edit → Create → Descriptors) or the context-menu of the side bar, users can activate the following functionalities:

- *List* – presents the list of existing descriptors;
- *Creation* – creates a new descriptor into the system;
- *View* – displays the contents of an existing descriptor
- *Modification* – edits an existing descriptor to enter changes;
- *Deletion* – deletes a descriptor from the system;
- *Copy* – creates a copy of an existing descriptor.

The options availability depends on the active user role as explained in section 3.2.2.
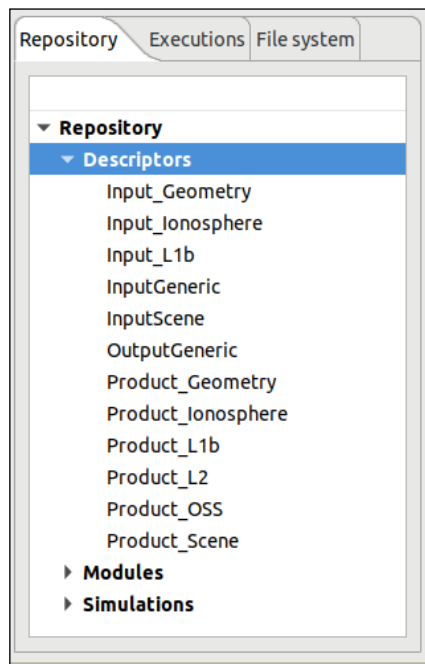
Figure 4-27: Descriptors in the side bar

Users can access operations in the menu bar of the main window (Edit → Elements → Descriptors or Edit → Create → Descriptors) or through the corresponding context-menu of the repository view.

4.3.1.1. *Descriptor list*

Users can access to a window that provides a tree-like structure with the list of descriptors known by the system, just as in the side bar but with the additional information of its description and the number of associated files.
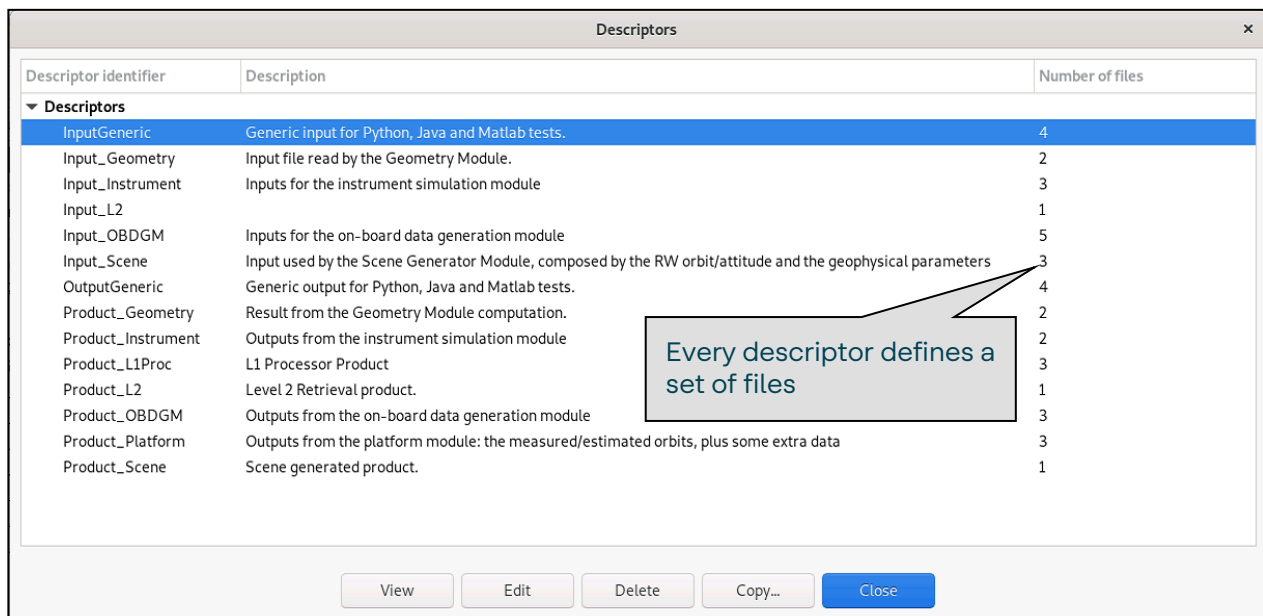
Figure 4-28: Descriptors list view

4.3.1.2. Descriptor creation

| M | ! |

Users can define new descriptors in case they want to accommodate third-party modules that cannot make use of any of the nominal descriptors. The frame shown in Figure 4-29 is responsible to define the descriptor's characteristics.

The attributes that identify the descriptor and that shall be set by the user are the following:

Attribute name	Format	Purpose	Example
	Short string	Descriptor's unique name.	LIDAR In
	Medium string	A brief description of its composition or the purpose of the set of files.	Orbit information and radiative transfer information

It is possible to alter the set of files that integrates the descriptor. Users can edit, add or remove files. Each individual file must be described by these two parameters:

Attribute name	Format	Purpose	Sample
	Medium string	The default location and name of the file. This is the file that is going to be suggested during the simulation definition (see section 4.3.3).	orbit.xml
	Medium string	Brief description of the file's composition, its purpose or its type (XML, NETCDF, etc.).	XML file with Orbital information

Figure 4-29: Create a new descriptor

It is important to note that <u>the default file name is the way to know if two modules are compatible and to connect them in the simulation definition</u>. A descriptor could also have zero files. Multiple descriptors with zero files could be created, but they are all equivalent to openSF.

![INDRA logo] ✕ INDRA

openSF

System User Manual

OPENSF-DMS-TEC-SUM01

4.9

18/06/2025

The order that the files occupy in the descriptor list is significant. The order must fulfil the directives of the command line specification of the simulation module because the way in which the files are arranged will define the order of the input and output files in the command line of the module execution (see [AD 1]). This order can be altered with the "up" and "down" buttons.

Upon creating a descriptor its default file field is considered as a template name, i.e., this definition may be used as-is during simulation execution or it can be changed in the simulation edition window. When changing the actual file descriptor in simulation edition the file location can be set to any path, either absolute or relative one. By default, the descriptor is considered as a relative path. It may be relative to (a) the user's work folder (as per $E2E_HOME), or (b) the simulation execution folder:

(a) It will be relative to the user's work folder when the descriptor is used for input only (e.g. the input descriptor of the first module of a processing chain).

(b) It will be relative to the simulation folder when the descriptor is used both as input descriptor and output descriptor (e.g. a folder generated by one module and later used as input to another).

4.3.1.3. *Descriptor modification*

!

Users can select a certain descriptor and choose the option to edit it. Note that any modifications performed on the descriptor will also affect all the modules and simulations that use it.

4.3.1.4. *Descriptor deletion*

!

Users can also select a descriptor and delete it. Once users confirm the operation the descriptor is erased from the repository. **Note that for consistency purposes, every module or simulation (and its results) that make use of this descriptor will also be deleted from the system**.
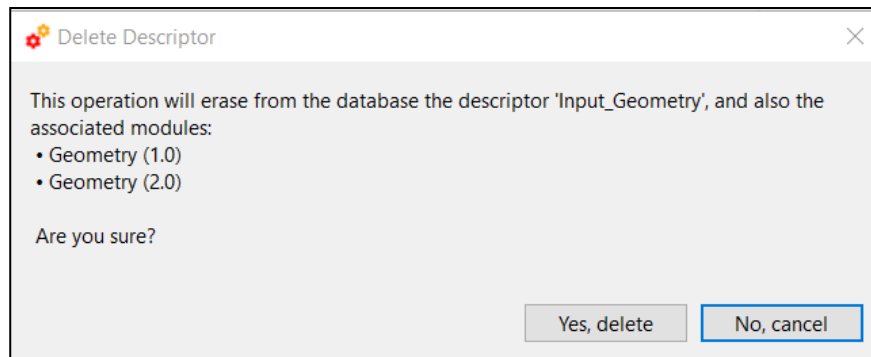
Figure 4-30: Confirmation dialog to delete a descriptor and its associated modules

4.3.1.5. *Descriptor copy*

!

Users can select a certain descriptor and choose the option to copy it. The user needs to specify a new name for the descriptor, unique with respect to the existing descriptors.
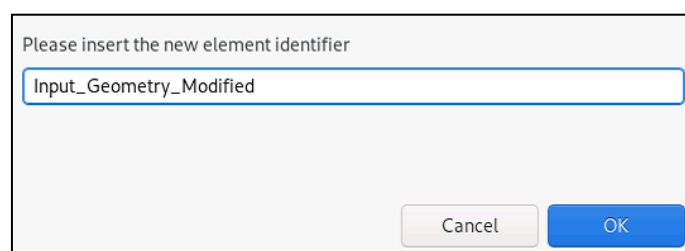
Figure 4-31: Copy of a descriptor

4.3.2. Modules

M	!	A	E

According to the definition given in section 1.4, a module is an executable entity that can take part in a simulation. Users are able to manage all modules that take part in openSF simulations. The operations upon modules, which vary depending on the active user role (see section 3.2.2), are:

- List – present the list of existing modules;

- Creation – capability to create a new module into the system;

- View – displays the contents of an existing descriptor

- New version – create a new version of an existing module;

- Modification – edit an existing module to enter changes;

- Deletion – delete a module from the system;

- Copy – creates a copy of an existing module.
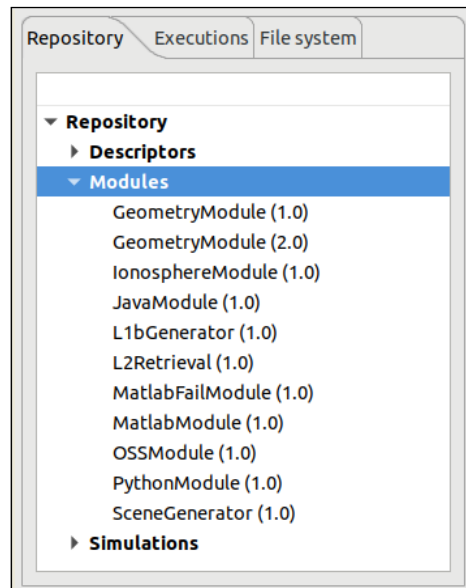
Figure 4-32: Repository view: modules

Users can access some of these operations in the menu bar of the main window (Edit → Elements → Modules or Edit → Create → Modules) or through the corresponding context-menu of the repository view.
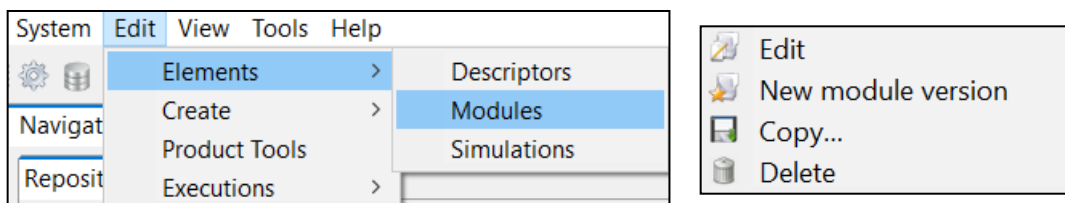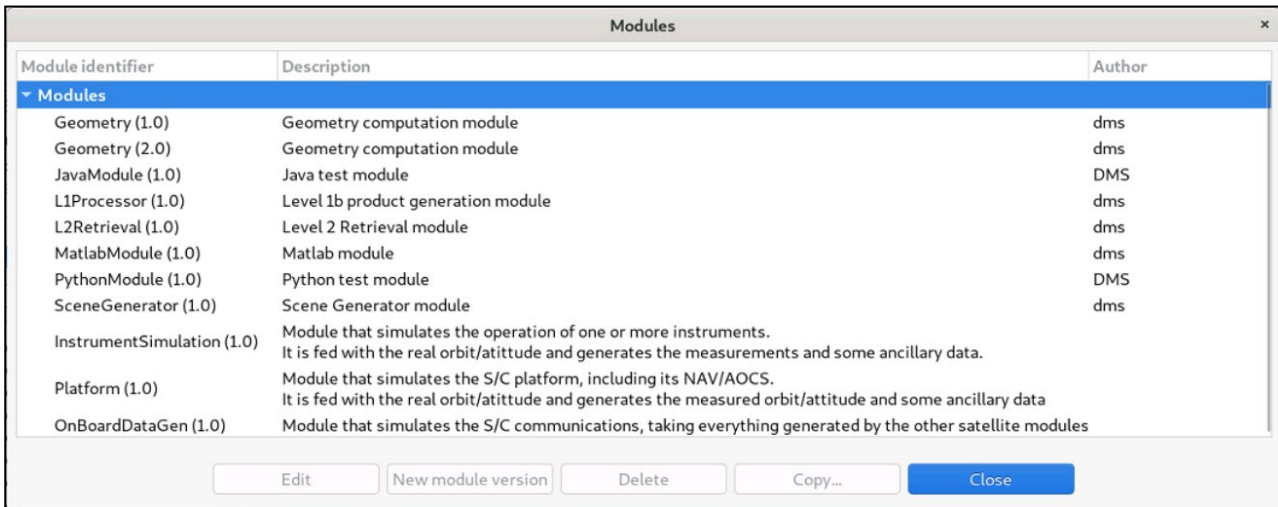
Figure 4-33: Operations from the main window (left) and module pop-up menu (right)

4.3.2.1. Module list

!	A	E

Accessing to this functionality from the main menu or from the repository, the system will show a list of modules known by the system. Figure 4-34 shows an example of the window that appears upon its selection. Users can select a certain version of a module and perform the operations in the toolbar.

Data attributes shown in this tree-table are module ID, version number, description and the name of the author.

Figure 4-34: Module list view

4.3.2.2. Module creation

Users can add a new module accessing this functionality from the main menu or alternatively clicking over the "New" button in the module manager.

This frame contains the components needed to introduce all data to define a new module in the system. These data (module attributes) are grouped with the following structure:

- General
- Configuration
- Input / Output

Each category is analysed in the following sub-sections.

4.3.2.2.1. General data

In this group (Figure 4-35) users must define general information about the module to create. The fields to be defined are the following:

Attribute Name	Format	Purpose	Example
Identifier	Short string	Unique module identification.	LIDAR
Module version	Float	In a new module this field will be filled with a default value.	1.0
Description	Medium string	Free writing area where to briefly describe the module.	State-of-the-art LIDAR instrument module
Author	Short string	Text field for the author's name.	DMS
Source file	Medium string	Optional field for a file[8] representing the source code of the module.	modules/LIDAR/src/lidar.f90
Executable	Medium string	The executable[9] file (either a binary or script) to be invoked in a simulation.	module/LIDAR/bin/lidar

---

[8]   Non-trivial modules in general will have several routines, so the "source" is likely to be an archive (e.g. module-src.tar.gz). This field is merely informative and has no effect on execution.

[9]   The compilation of the modules is a process outside the scope of openSF.

Figure 4-35: Module general data

4.3.2.2.2. Configuration

Selecting the "Configuration" tab (Figure 4-36), users can select the XML configuration file using file-browser dialogues. The text area is also provided to preview the XML code.
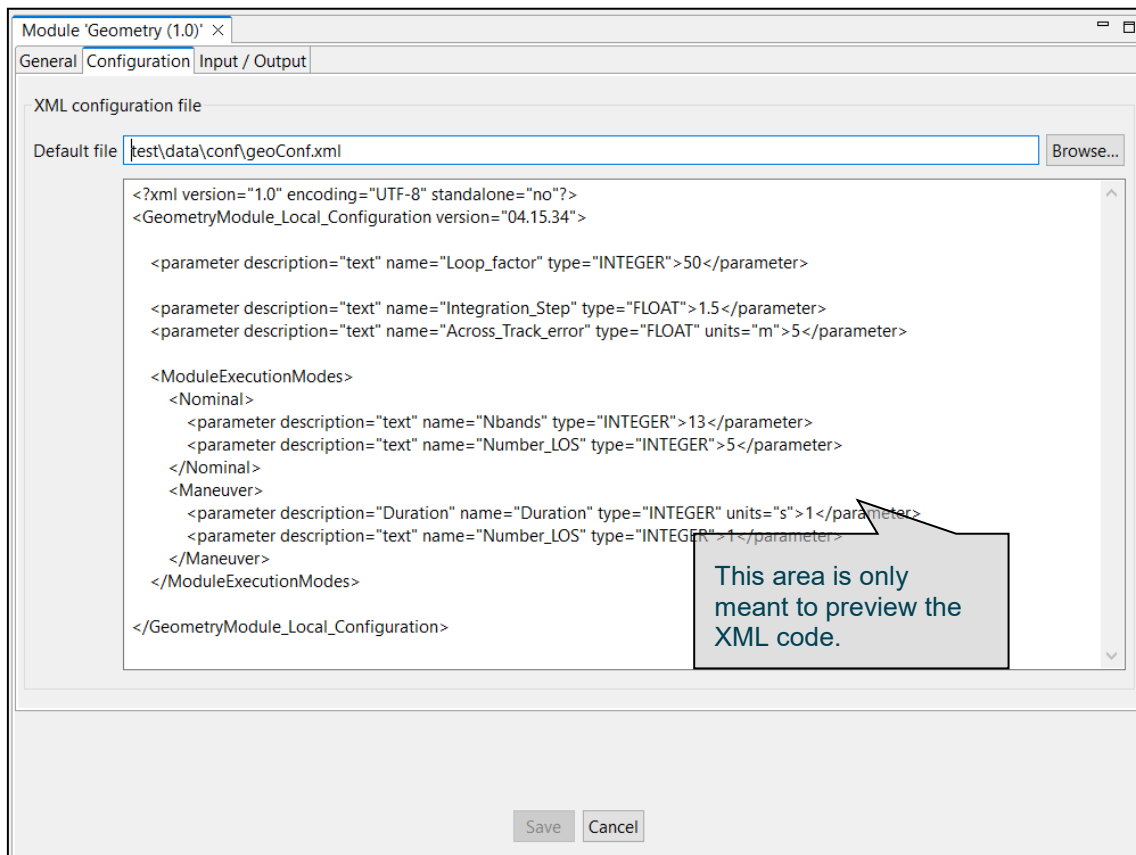
Figure 4-36: Module configuration

4.3.2.2.3. IO descriptors

The "Input/Output" tab (Figure 4-37) in the Module properties window enables users to specify, respectively, the contents of the input files expected for the module, and the output files the module produces as output. Thus, within this tab users can select the input and output descriptors for this module. The default file identifiers of the IO descriptors allow the module connections to be made automatically when defining simulations.

To enhance user experience, if descriptors are created or removed while a module is open for editing, opening the "Input descriptor" and "Output descriptor" combo boxes display an up-to-date list of values.

Each IO descriptor has an identifier that uniquely identifies the descriptor among the system. It may happen that an IO descriptor for a new module may exist already in the system, that is, this module uses the same type of files and file contents as another module. Therefore, a combo-box component is presented with the list of known IO descriptors in case the user desires to select an existing one.

Figure 4-37: Module input/output specification

4.3.2.3. Module modification

| M | I | E |

When editing a module from the repository view or from the modules list, openSF will present the same window as in the previous section with all known data already filled. The window presents in write-mode only the data fields that are susceptible to be modified. If users want to change more attributes of a certain module, other operations must be used ("module creation" or "new module version").

This frame is intended to let users modify data of a certain module. Once they have finished with the editing, they can accept or cancel the changes made with the buttons at the bottom-side toolbar.

4.3.2.3.1. Module upgrade - New version

A new version of a module represents an upgrade of the implementation of a given module. This means that users can define a new module by altering any of the elements defined in the module creation, like for example the executable file of the module, the configuration file or the input and output files.

Users can create a new module version selecting the correspondent action from the context-menu of the repository view or alternatively clicking over the "New module version" button in the module list.

The system will automatically perform a "minor version upgrade" (for example, from 1.0 to 1.1), but this numbering can be manually modified by the user.

This way, users can have two versions of the same module with modifications between them.

4.3.2.4. Module deletion

| I |

Users can select a certain module and choose the option to delete it. Once users confirm the operation the module is erased from the repository and the file system. Note that also every simulation (and its results) that uses this module will be erased from the system for consistency purposes.
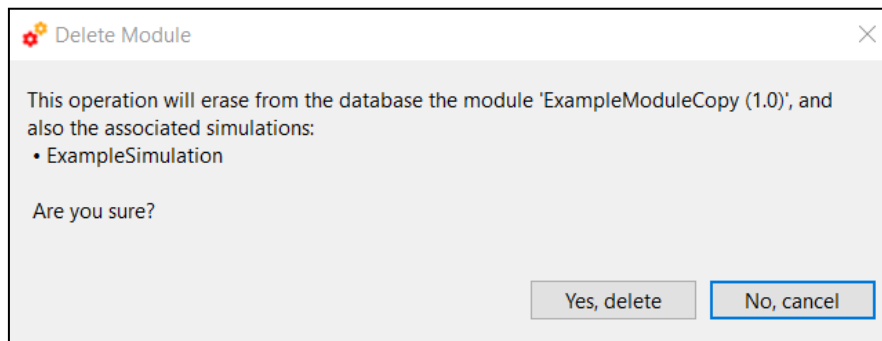
Figure 4-38: Confirmation dialog for deleting a module and its associated simulations.

4.3.2.5. Module copy

I

Users can select a certain module and choose the option to copy it. The user needs to specify a new name for the module, unique with respect to the existing modules. All module definitions are thus copied to the new module instance.
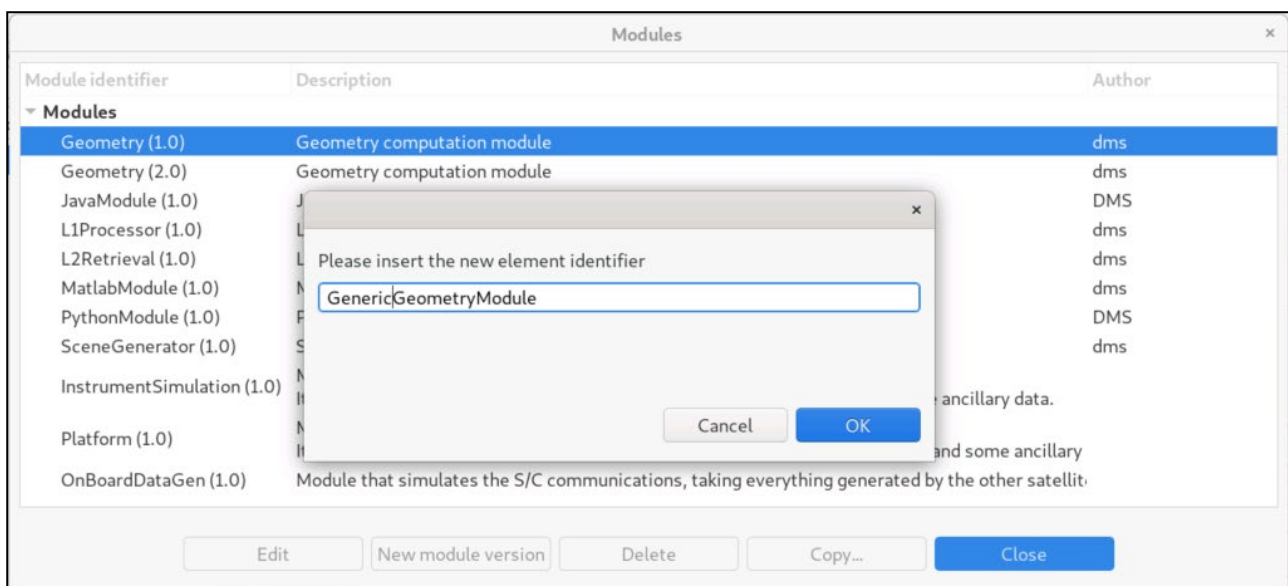
Figure 4-39: Module copy

4.3.3. Simulations

I	A	E

Per section 1.4, a simulation is defined as either an execution of a set of modules or an iterative execution of a set of modules with different parameter values.

Users can access to the list of simulations existing in the system in the "repository view" (Figure 4-6) of the side bar or via the "repository" menu from the main menu (Figure 4-1).
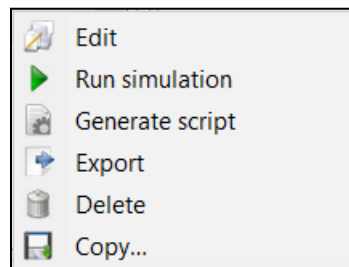
Figure 4-40: Simulations pop-up menu

Operations involving simulations include the following:

- *List* – present the list of existing simulations;
- *Creation* – capability to create a new simulation into the system;
- *Modification* – edit an existing simulation to enter changes;
- *Deletion* – delete a simulation from the system;
- *Run* – Starts a new simulation execution;
- *Script generation* – creates and stores a script describing the simulation;
- *Exportation* – Exports the entire simulation definition;
- *Copy* – Creates a copy of an existing simulation.

4.3.3.1. Simulation list

Users can access the simulation list via the menu bar "Edit"→"Elements"→"Simulations".

Figure 4-41 shows an example of the simulation list window that is presented upon selection. Below the table including the simulations existing in the system, there is a tool-bar with buttons to access the different functions listed previously. Users can thus select a certain simulation and perform the operations shown in the toolbar.

Data attributes shown in the simulation list table are simulation ID, description and the name of the author that created the simulation.
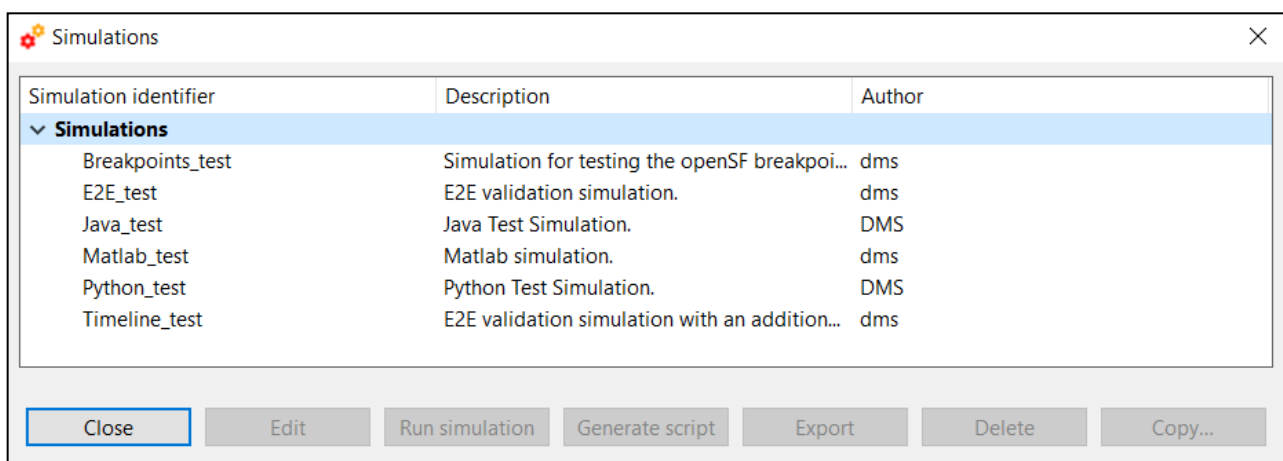
Figure 4-41: Simulation list view

4.3.3.2. Simulation creation

Users can create a new simulation definition by clicking on the corresponding action from the Simulation menu in the menu bar (Edit → New Elements → Simulation…) or by right clicking on the "Simulations" node in the repository side bar. The user is then guided through the steps of creating a simulation. These steps are described in section 4.3.3.6.

4.3.3.3. Simulation deletion

Users can select a certain simulation and choose the option to delete it. Once the operation is confirmed the simulation is deleted from the repository and the file system. **It is to be noticed that for consistency reasons, the simulation deletion causes the removal of all results generated from that simulation from the system**.

4.3.3.4. Simulation copy

Users can select a certain simulation and choose the option to copy it. The user needs to specify a new name for the simulation, unique with respect to the existing simulations. All simulation definitions thus copied to a new simulation instance.
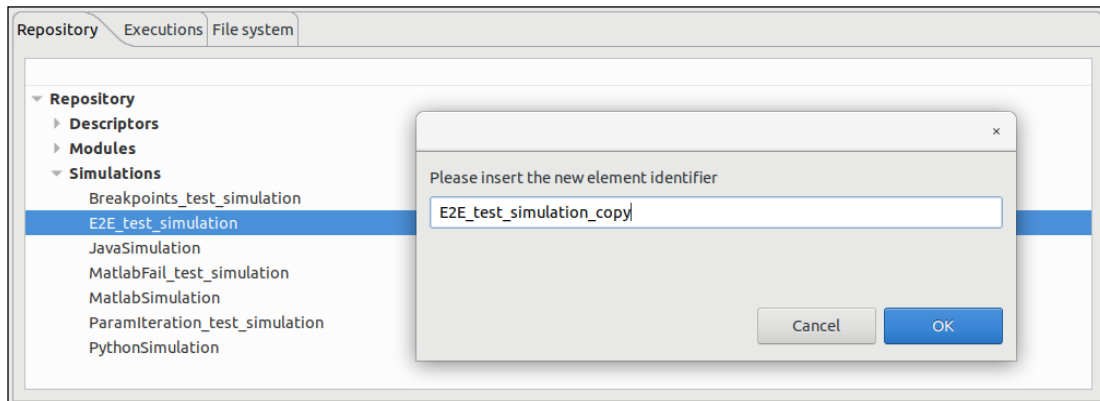
Figure 4-42: Simulation copy

4.3.3.5. Simulation modification

It is possible to edit a given simulation and create a different one altering the information previously stored. When editing a simulation, the system will show the same window as shown in the simulation creation. But this time, all the information concerning this simulation will fill every data field.

4.3.3.6. Settings in a simulation

When a simulation is created, a number of properties have to be specified. These properties are grouped into three different tabbed panels:

- **Editor** contains the definitional aspects of a simulation, including:
  - The modules that comprise the simulation and the connections between them (definition).
  - The files to use with each module (input/configuration/output).
  - Any tasks that should be run by default after completing the execution (post-processing).
- **Control** includes the transitory aspects to configure a specific run, including:
  - The parameter values (including multi-execution configurations).
  - Whether to stop execution at a specific module (breakpoints).
- **Executions** shows only the executions spawned from the current simulation. This view initially appears empty.

Editor and Control tabs are separate in behaviour, especially regarding persistence. In the **Editor** tab, all changes are persistent, but they are only saved when the user chooses to do so, with "Save" and "Revert" buttons provided. If switching away from the Editor tab and there are unsaved changes, the user is prompt to save and switch tab or cancel and continue editing.
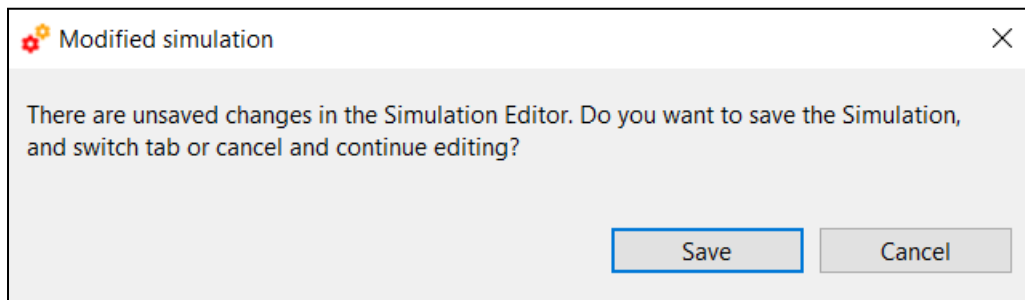
Figure 4-43: Save dialog to leave the Editor tab

In the **Control** tab, changes are either non-persistent (parameter values for a single run) or persist only as the last settings (multi-execution configurations, breakpoints, etc.), with changes saved automatically.

Note that in order to fully define a simulation the running settings have also to be specified. These are described throughout section 4.4.

4.3.3.6.1. Simulation definition

Figure 4-44 shows a blank simulation creation window. When creating a simulation only the "Save" button is available.
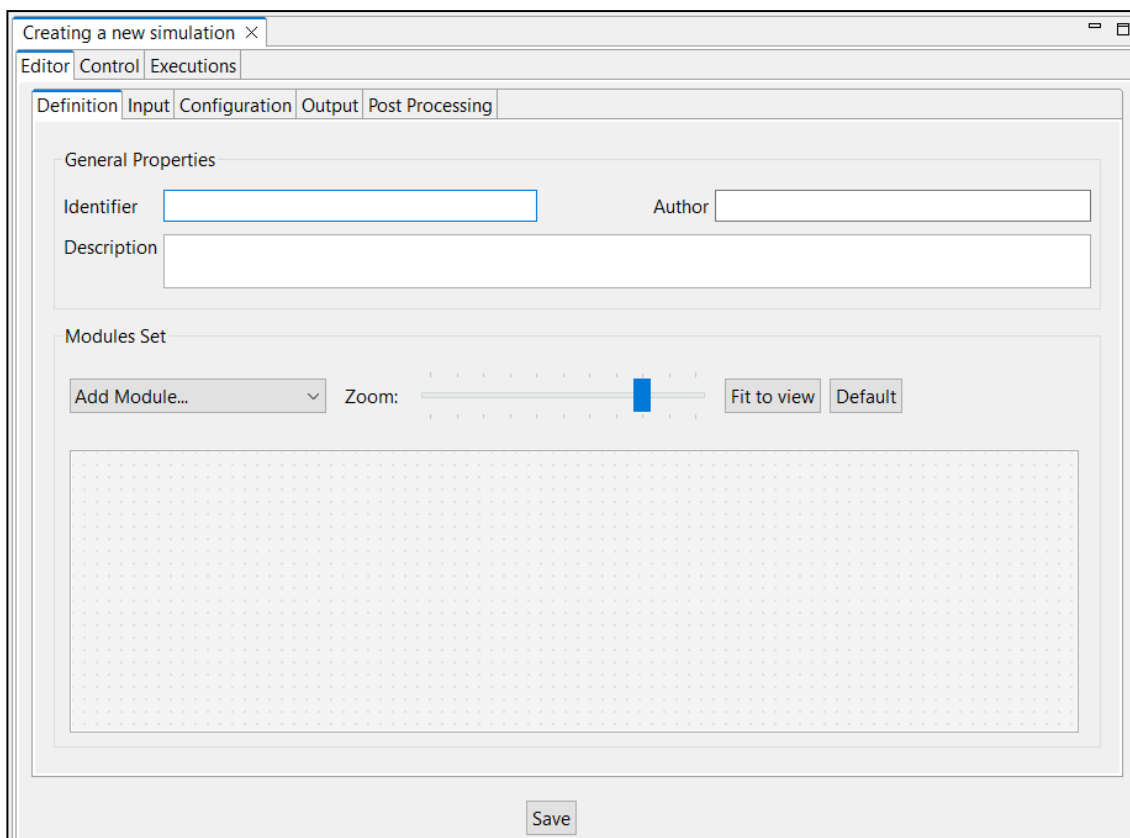
Figure 4-44: Simulation general properties

First thing, the user must fill the following general properties:

Attribute name	Format	Purpose	Example
	Medium string. No blank spaces allowed.	Uniquely identifies this simulation definition into the system.	E2E

	Medium string	Brief remarks about the goals and characteristics of the simulation	This is a full end-to-end simulation for the EarthCARE mission
	Normal string	Name of person or group responsible of the simulation definition	DMS

Then, to add a module, open the dropdown list named "Add module" and select the module to add. After each action, the list will go back to showing "Add module". To enhance user experience, if modules are created or removed while the simulation is open for editing, opening the "Add module" combo box display an up-to-date list of values.

The diagram below shows the simulation components and their relationships (see Figure 4-45). Every module is represented by a rectangle identified by the name of the module, with inputs or outputs displayed as square connectors on the side of the respective module. Input files are always shown on the left side of the module, while output files are shown on the right side of the module (e.g. visually the data flows from left to right). Data flow and dependencies between the modules are represented by arrows linking output to input connectors.
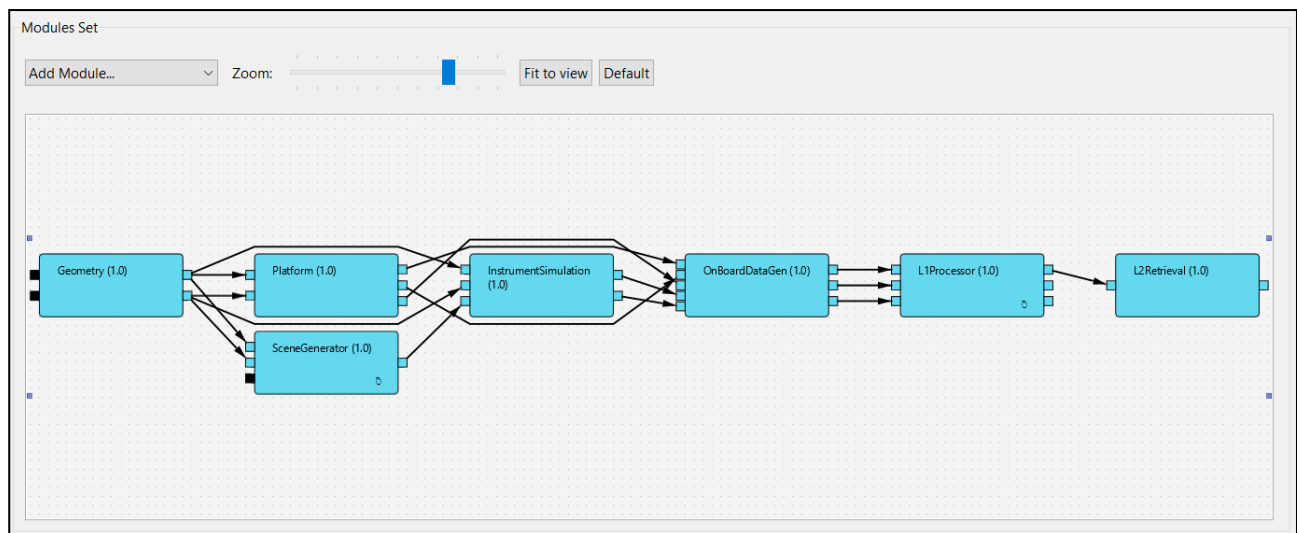
Figure 4-45 Simulation diagram

The diagram can be scrolled, panned and zoomed with the mouse wheel and by holding CTRL and using the wheel. The buttons at the top can also be used to zoom the diagram, fit it to the view or reset the zoom level to the default.

The colour of the connectors indicates whether the file is: available (black); pending (blue); or, missing (red). Hovering the mouse on top of a connector displays the related file location as a tooltip.

A user can link two connectors, indicating that the output of a module should become the input of another one, by clicking on two connectors sequentially. In particular, the input file location is set to the output file location.

Editing the settings of a module can be done by double clicking it. A side area will open as shown in Figure 4-46, allowing the user to edit the input, output and configuration file location.

Changing the location of a configuration file may change the available parameters, which may cause warnings to be displayed, see section 4.4.2.6.
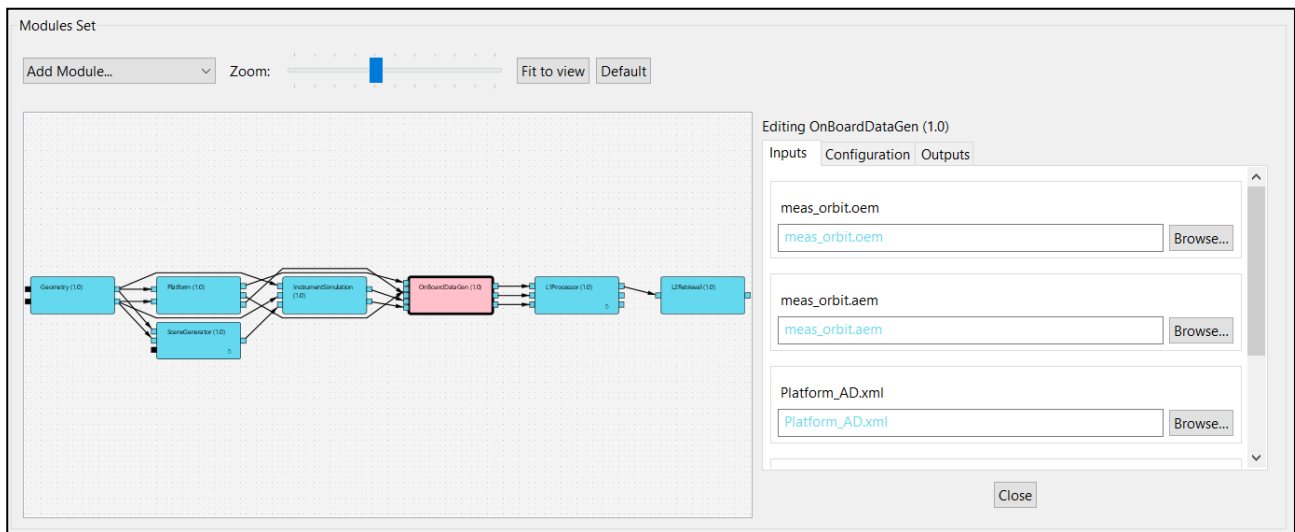
Figure 4-46 Module editor side area

Users can remove a module from a simulation by right clicking on it and selecting "Delete Module".

Finally, right-clicking on an arrow, it is possible to edit a connection between two modules. The context menu displays three options:

- ❏ Rename file: It allows to rename the file without breaking the links between modules. In fact, every file that previously had the same name will be renamed as well.
- ❏ Rename source: It allows to rename the source output file. This will break the connection between the two modules.
- ❏ Rename target: It allows to rename the target input file. This will break the connection between the two modules.

The file(s) can be renamed through the side area that gets displayed upon selection one of the options (see Figure 4-47).
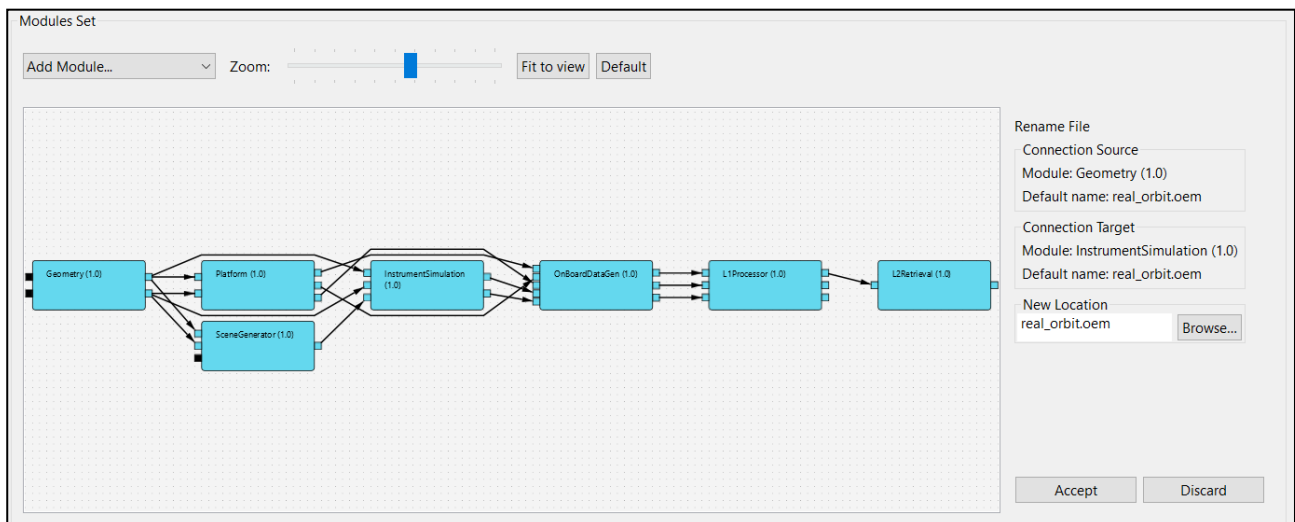
Figure 4-47: File edition side area

Figure 4-51 shows the simulation modification window. There are only three buttons; the fourth element (the warning icon) is not clickable:

- **Save As:** saves the simulation with a new identifier.
- **Save:** this button is only enabled when the simulation has been modified. If the user has modified any parameter in the parameters table of the Parameters tab under Control, when saving the simulation, a dialog like the one shown below will appear.
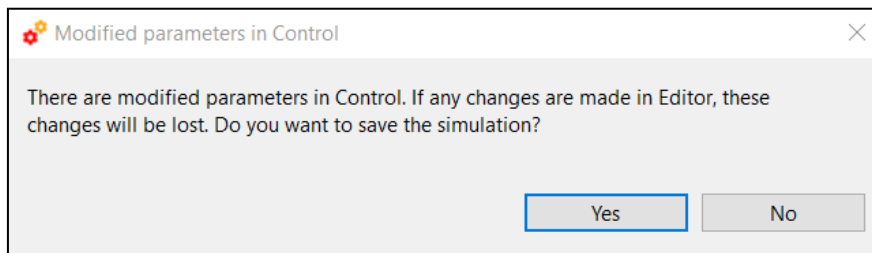
Figure 4-48: Dialog shown when saving with modified parameters.

- **Revert:** as with the "Save" button, this button is only enabled when the simulation has been modified. Prompts the user to confirm the restoration of simulation values to the last saved state. If the user confirms, all unsaved changes in the simulation will be discarded.
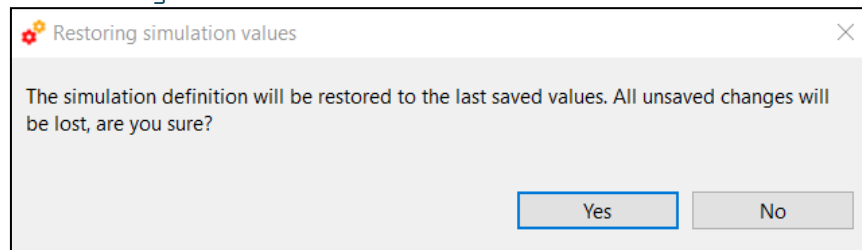
Figure 4-49: Dialog shown when pressing the Revert button.

- **Warning icon:** only visible after the parameters table has been modified. If the user moves the mouse over it, a tooltip will be displayed.
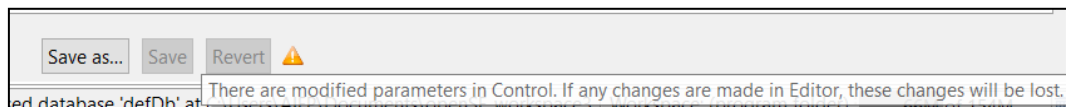
Figure 4-50: Tooltip of the icon warning
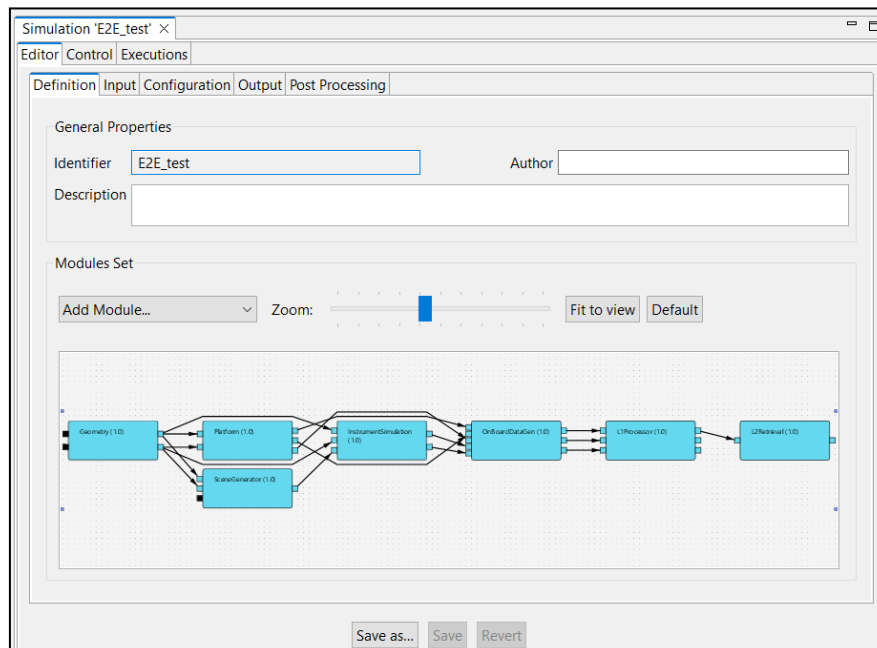
Figure 4-51: Simulation modification

4.3.3.6.2. Simulation conflicting files resolution

When defining simulations, inputs files, configuration files and output files are set up. Each of them has its own name and file location. However, it may happen that when selecting these files in the simulation definition, they have the same file name, producing a conflict when writing the files in the simulation folder. Therefore, depending on whether the file is an input, output or configuration file, their name in the simulation folder will be composed by a specific part of the path.

For inputs that are already inside the simulation folder, the same path is returned.

For configuration files and simulation-wide inputs (that is, inputs that are not linked to a module output, with initial status Available or Missing) the path will be calculated as just the file name. But if the tentative output path is already linked to a different input path, the name will have to be disambiguated. Therefore, to disambiguate overall input and configuration files, the file name is prefixed by the shortest part of the path that allows for disambiguation. For example, if there are two inputs `cons/sat25/data/orbit.oem` and `cons/sat27/data/orbit.oem`, the files in the simulation folder should be named *sat25_data_orbit.oem* and *sat27_data_orbit.oem* respectively.

Furthermore, configuration files for modules are can have the same name and final path. In such cases. these files will be renamed to avoid conflicts, allowing multiple modules with identically named local configuration files to coexist.

For output files, and any inputs that are linked to other outputs in the simulation (i.e., whose initial status is Pending) the full path will be kept as the output path. Thus, two different input paths with the same file name will reach different in-sim-folder outputs. For example, two inputs and two outputs with paths such as *a/x.txt* and *b/x.txt* can coexist without clashing.

4.3.3.6.3. Input files

Selecting the second tab under the simulation Editor tab (Figure 4-52), the system will ask for the location of the input file list needed to start the simulation.
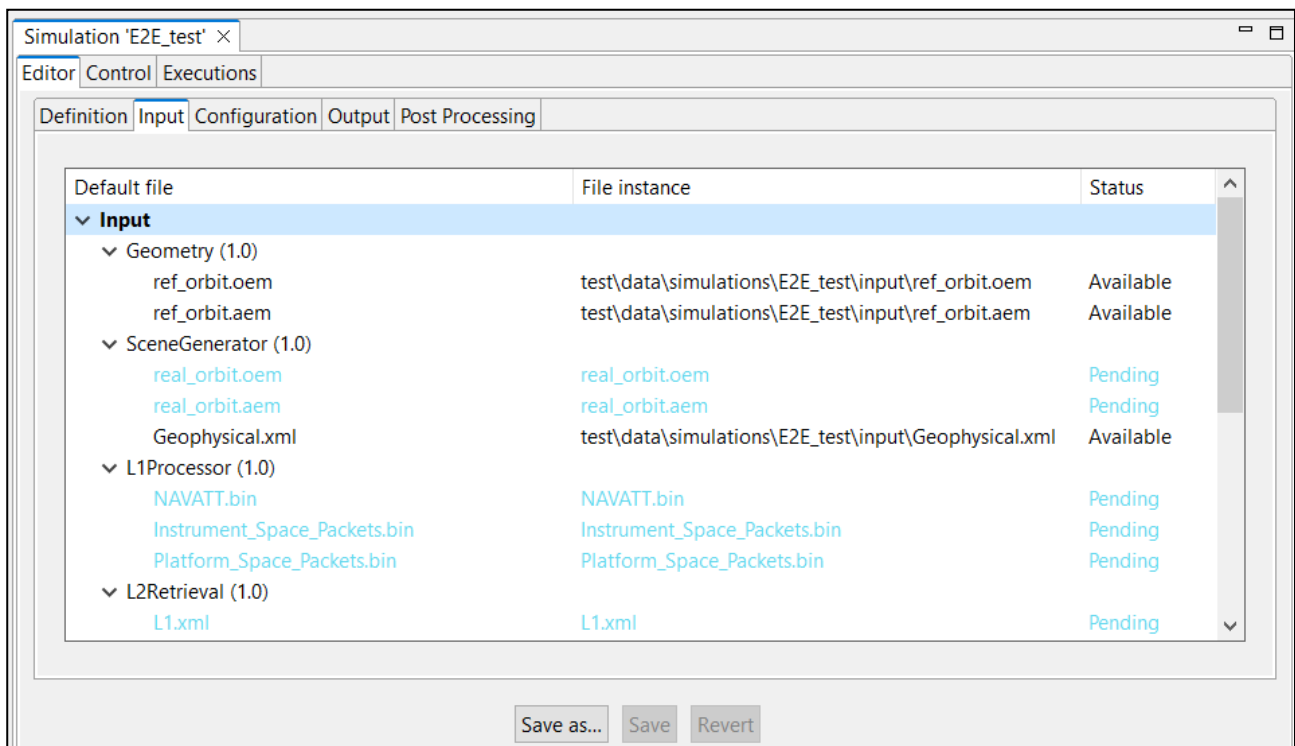
Figure 4-52: Simulation inputs definition

Double-clicking on the "file instance" column gives the possibility of writing the file path. Note that at the time of defining the modules (see section 4.3.2.2.3), input files are defined in an abstract way (i.e. specifying that the given module shall require a certain input file). It is in this step where the user can select the instance of each file (i.e. selecting in the file system the instance of the file specified at descriptor level).

The "Status" column will show one of three different options for each file:

- Available – the file instance is present, so the file is ready to be used for the module executions;

- Pending – the file is not present but it will be generated for the module executions before needed;

- Missing – the file is not present and is not scheduled to be generated before needed. Edit the file instance and change it so that it appears as an existing file.

File instance locations can be specified using absolute paths or relative to $E2E_HOME.

4.3.3.6.4. Configuration files

As it is done in the "Input" tab, the simulation needs to be provided with the location of configuration files needed by all the modules involved in the simulation (Figure 4-53). An extra configuration file is needed for this step, called global configuration file. This contains the general parameters characterizing a simulation, not bounded to a specific module.
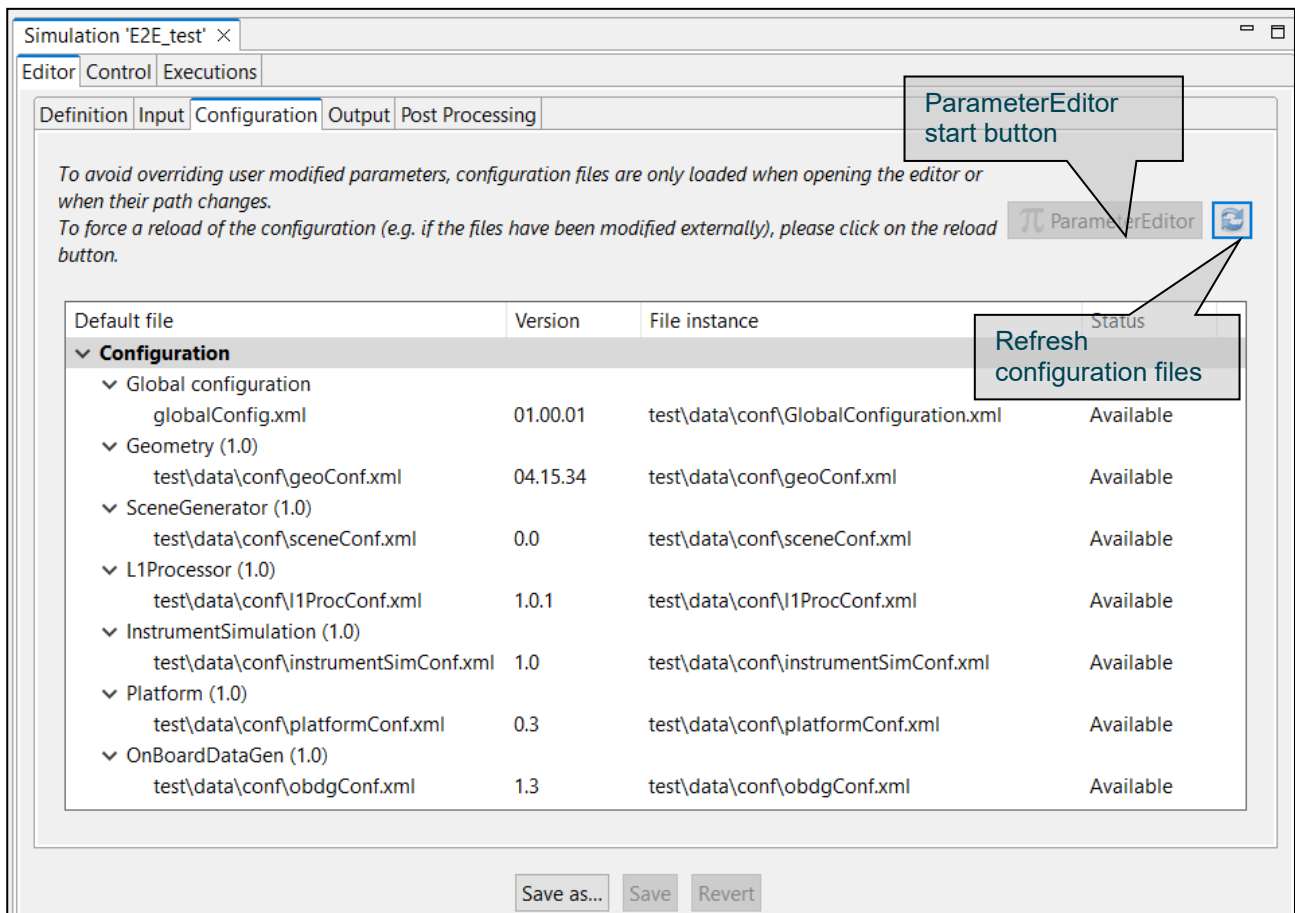
Figure 4-53: Configuration files definition

This window will present the list of modules present in the simulation and will ask for the location of each needed configuration file. At this stage, the modules cannot be changed nor deleted.

Double-right-clicking on a file row, the system will show a file browser to locate a specific configuration file. Providing an existing file will update the status to "Available".

The configuration file panel has two buttons providing the following capabilities:

- *ParameterEditor button* - located at the top of the configuration files list launches the ParameterEditor application with the selected configuration files already loaded. See [RD-4].

- *Refresh button* - reloads all configuration files, reading them from the file system and updating the simulation parameters. It is recommended to do this after editing configuration files with another tool. **Caution: All changes in parameter values made from openSF interface will be lost.**

Changing the location of a configuration file, or reloading them if they have been modified outside openSF, may change the available parameters and cause warnings to be displayed, see section 4.4.2.6.

Note that the configuration files for all active modules, plus the global configuration file for the simulation, must be available in order for a simulation to run.

As it has been seen in the previous section about input files, it is possible to provide the location of configuration files. With the same action it is possible to specify where the global configuration file is located. Global configuration file location can be edited every time the user creates a new simulation and can be found in the "Configuration" tab of the "Simulation Setup" panel.

4.3.3.6.5. Output files

Users can change the name and location of the output files that will be generated by execution of modules. Selecting the "Output" tab of Editor, the system will show a list of output files grouped by modules, and following the execution order.

By default, these files will have a "Pending" status, meaning that they will be produced by the action of the module's execution. Once a simulation has been executed and output files generated, file instance column will show the absolute path of the generated file and the status will be "Available".

Note that no pair of output files in a simulation can share the same location, since they would be conflicting.
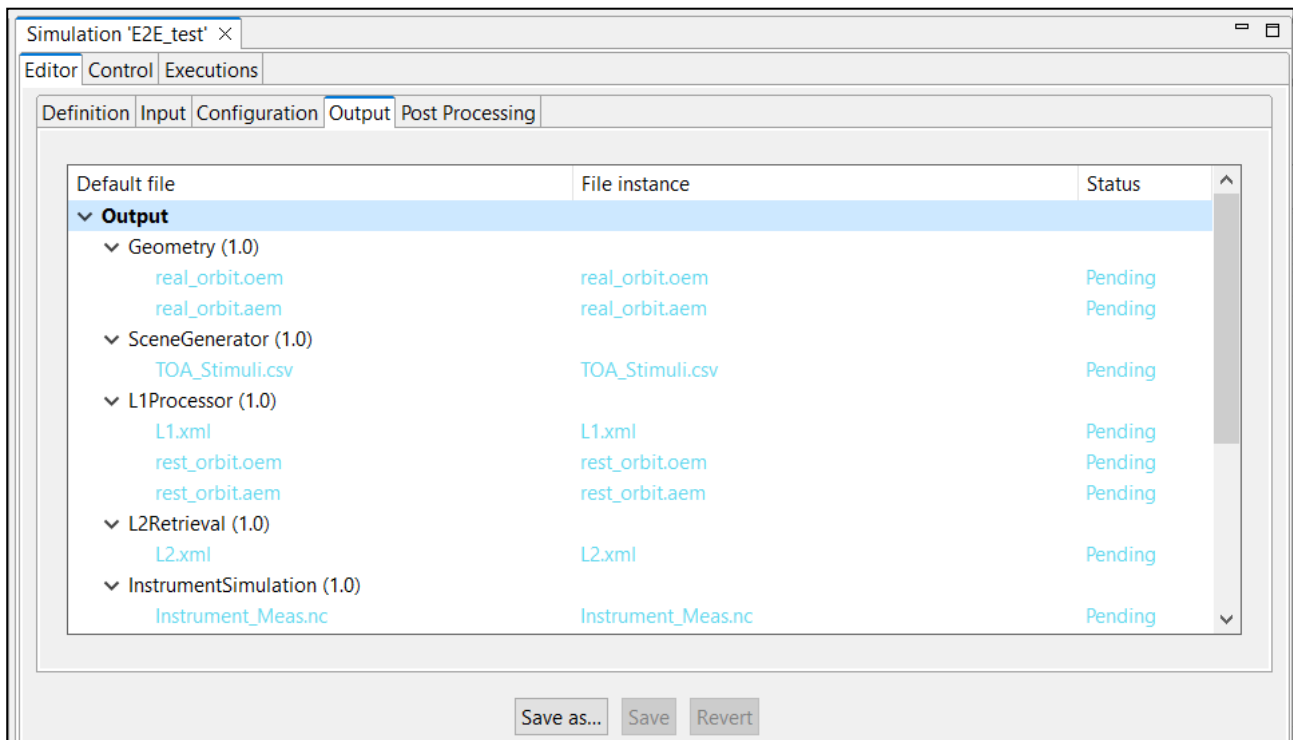
Figure 4-54: Simulation output definition

4.3.3.6.6. Parameters configuration

Selecting the first tab under the simulation Control tab (Figure 4-55), the user is able to alter the contents of the configuration files to change the behaviour of the module. Assuming that the module has been correctly integrated into the system and a valid configuration file is reported, this tab shown in Figure 4-55 will present the list of module parameter (and values) grouped by modules (sorted in execution order).

The user should also note that any changes made in the Parameters tab are local to the execution and do not affect the default configuration of the Simulation. It should be used the Parameter Editor for this purpose. Furthermore, in order to indicate that the parameters table has been modified, an "*" is included in the header. This effect is presented in Figure 4-56.
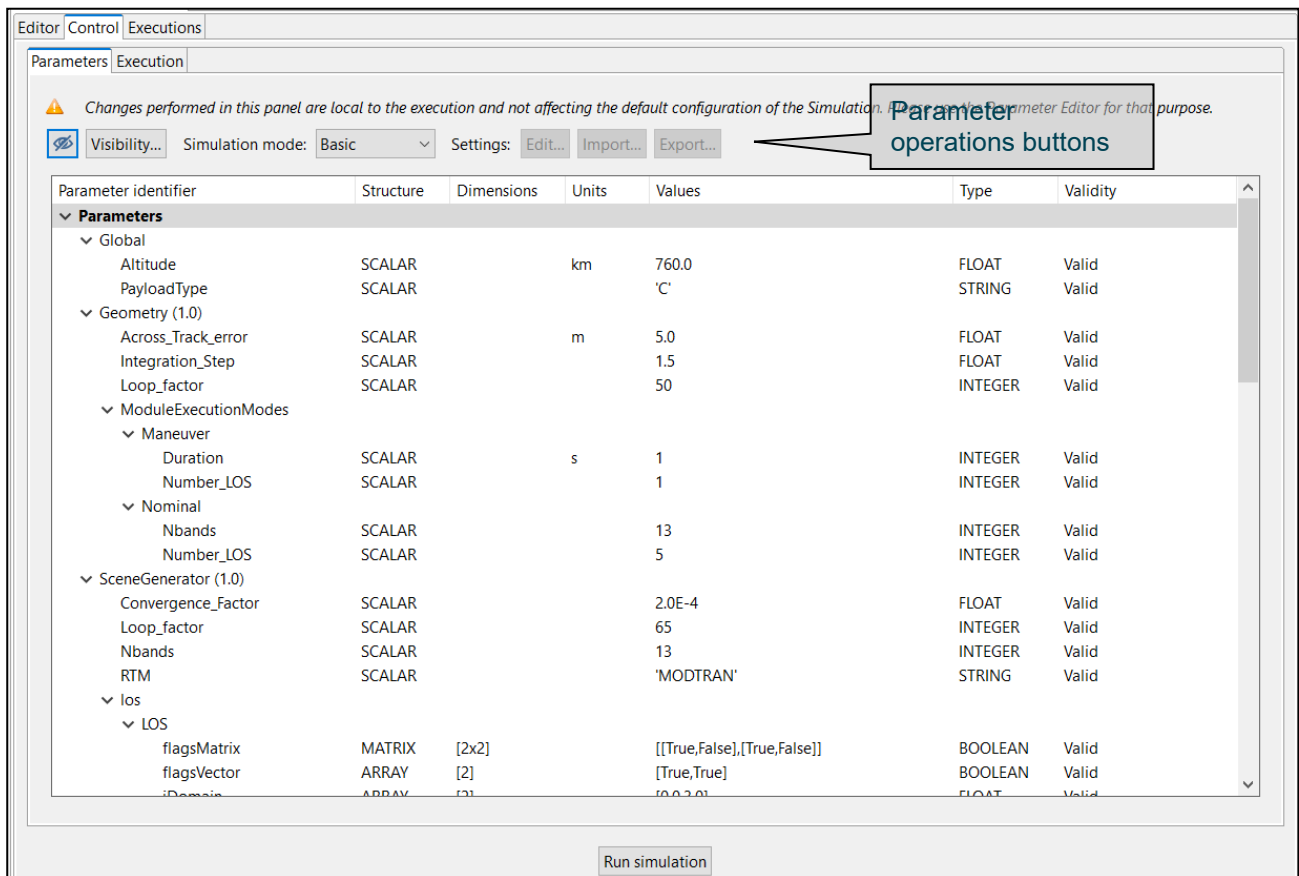
Figure 4-55: Simulation parameters definition
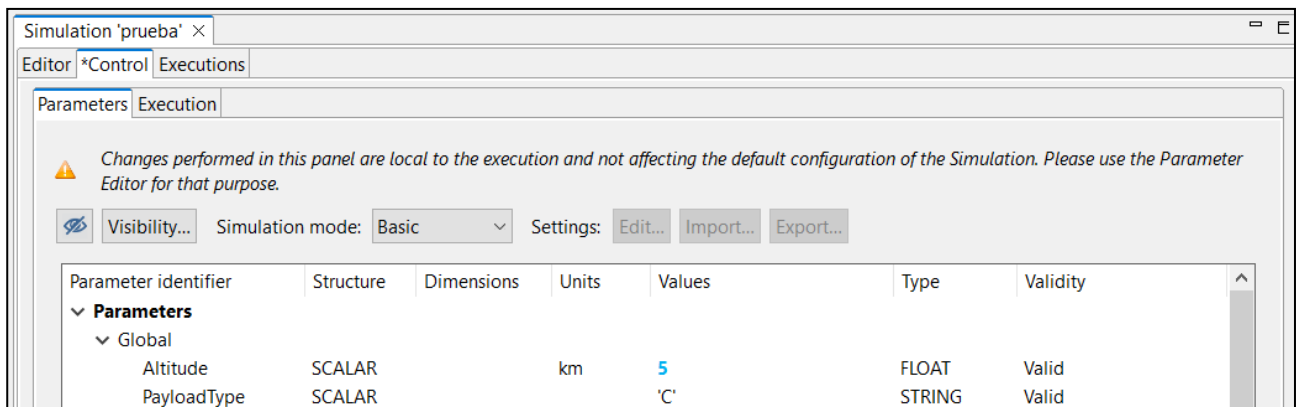
Figure 4-56: Parameters table modified

Users can consult the following list of attributes:

Attribute name	Format	Purpose	Sample
Parameter identifier	Medium string	Complete name of the parameter. A parameter identifier is formed by its path into the file structure (dot separated) and its parameter name.	parameters.execution_mode
Description	Medium string	Brief description of the parameter purposes and values	USER or CFI orbit
Type	Options: INTEGER, FLOAT, STRING, BOOLEAN, FILE, TIME and FOLDER	Parameter values type. Used to present different editor when editing the parameter value. See [AD-E2E].	STRING

Values	(Different types)	Parameter value. A unique value for each parameter is needed	CFI
Complex Type	Options: SCALAR, ARRAY, MATRIX	Parameter structured type. Used to indicate the multidimensional type of the parameter. See [AD-E2E].	ARRAY
Dimensions	[*cols* x *rows* x *layers*]	Size of the dimensions. Layers are only displayed if applicable.	[3x2x3]
Units	Short string	Physical units of measurements if applicable.	m/s
Validity	Valid, Non-existing file/folder, Illegal value format, Dimensions inconsistent with value, Invalid empty entries, Value outside allowed range.	Status of the parameter value integrity. openSF checks the parameter validity anytime its value is updated.	OK

Only the "values" column is editable to the user, the others just present useful information describing each parameter. To edit the value, double click the values cell in question and it will become an edition box. In addition, for scalar file parameters, a file browser is provided to locate a specific file in the graphical interface. Providing an existing file will update the status to "OK".

openSF checks the parameter validity anytime a run simulation is initiated, if any parameter is not valid the system shows a warning message.
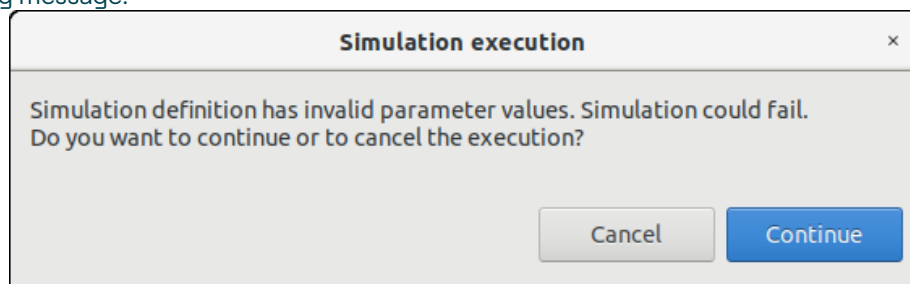
Figure 4-57: Simulation execution warning message

Note that changing the value of these variables will not affect the "template" configuration files specified in the "Configuration" tab. The variables involved in a simulation definition are stored in the database with the chosen values, meaning that the simulation will use them during the execution.

The specific format used for the single-line representation is the same as in ParameterEditor. The current representation (for non-scalars) is designed to be copied into Python. The same operation can occur in the other direction, from Python to the application, as long as the following conditions hold:

❑ All elements are literals. For example, [1, 2, 3] works but [1, sum(range(1,3))] does not

❑ Elements are homogeneous. Thus, ["a", 2, True] is not valid input for a parameter.

❑ ARRAY parameters may have staggered dimensions. ARRAYs support fully empty rows and layers but not fully empty columns. By extension, fully empty ARRAYs are not valid.

In addition, for scalar parameters that can be represented as a python string, the explicit string format is not needed, and a fully literal string is accepted. In other words, the string does not have to be between quotes, if it does not begin with a quote or a square bracket.

Note that this representation may change in future versions, and interoperability with Python (or any specific version or library) is not guaranteed in general.

The user can filter the parameters that are displayed in the HMI and modified for a simulation execution. The *'Parameter Visibility'* functionality allows marking each parameter as visible or not. In this way, those parameters that are configured seldom can be hidden to present to the user a leaner list of input configuration items, avoiding an overcrowded parameters tab. This setting can be triggered at parameter level, at module level or at the whole simulation level. To do so, the user can open the Visibility view through the dedicated button. Black parameters are visible, while the rows for parameters that are hidden by the visibility rules of the module are greyed out.
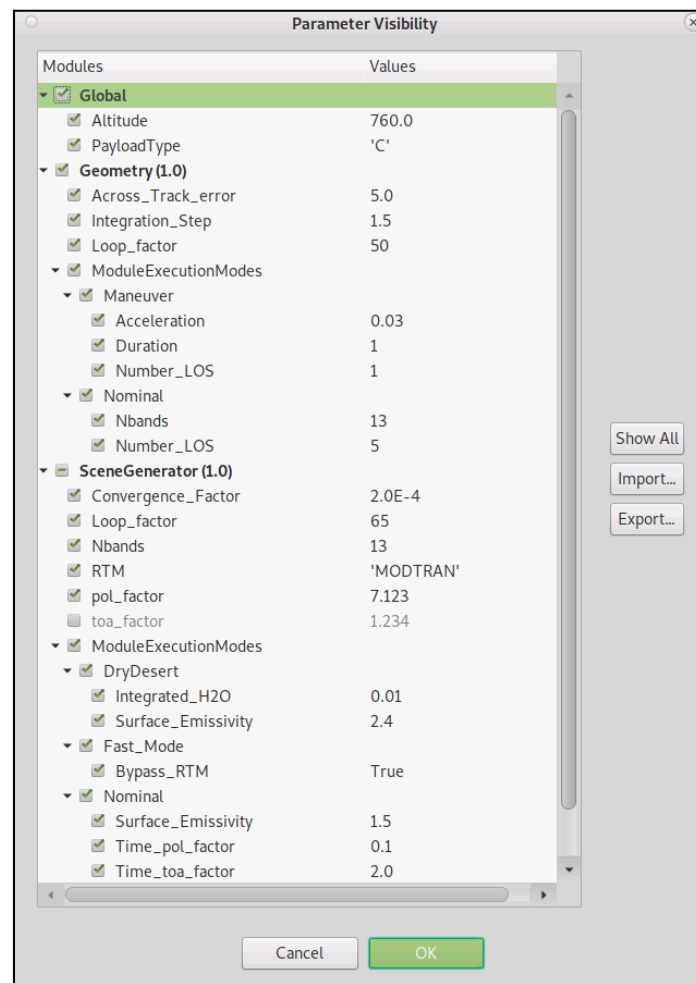
Figure 4-58: Parameter visibility view

Users can show/hide a parameter or a group of parameters with the checkboxes next to their names, use the Show All button to reset every parameter in the simulation to the visible status, load the parameter visibility definition specified in .txt format or save the hide/show state of every parameter in the simulation to a text file.

Once the parameter visibility has been set it can be switched on/off through the 'eye' icon in the simulation → Parameters tab (see Figure 4-55).

4.3.3.6.7. Parameters visibility file

The specific format of the visibility file used for the configuration parameters is explained in this section. This file will dictate which parameter is visible in the openSF GUI without affecting the module run in any way. The visibility file is a text file that follows the following rules:

- ❑ An empty line, or one starting with "#" is a comment and is ignored
- ❑ A line starting with "[" opens a new section, replacing the current one. A section header must be formatted either [MODNAME] or [MODNAME=VERSION].
  - o The first form matches any version of the named module.
  - o When MODNAME is GLOBAL, the section contents only apply to the GCF
- ❑ Wildcards are used in the import. Only * and ? characters are allowed: the first stands in for any number of characters including none, while the second represents a single character.
  - o Note that wildcards do not cross name boundaries: the pattern "a.b*c" matches "a.bxc" but not "a.bx.c" due to the presence of the period.
- ❑ Every line containing a parameter or group name hides that element and its descendants.
- ❑ Every line containing a parameter or group name starting with "!" makes the element visible
- ❑ The lines are executed sequentially from the first/top to the last/bottom line in the file; lines appearing later in the file (i.e. closer to the bottom of the file) override preceding ones.

For example, on a filter file containing "group" followed by "!group.parameter.a", the first entry hides all parameters related to "group", while the second entry overrides the first by making the parameter "group.parameter.a" visible. Another parameter "group.parameter.b" would remain hidden, due to the first filter rule.

4.3.4. Results

A   E

Once a simulation has run, its results and settings are stored in the file system. A series of operations can now be performed, ranging from reviewing the simulations to post-processing the results. These operations are presented in this section.

4.3.4.1. Result view

Users can inspect the results view of any executed simulation by opening it and selecting the "Results" tab. This view is similar to the simulation editor but including more information (see Figure 4-60). Accessing this functionality, users can consult the result of a simulation execution.

In the "Definition" tab the user can see which modules run to completion and which files were generated (see Figure 4-59). The completed modules are highlighted in green, while the uncompleted ones are displayed in blue. Moreover, generated files are represented by black connectors, while missing ones are indicated as blue connectors.
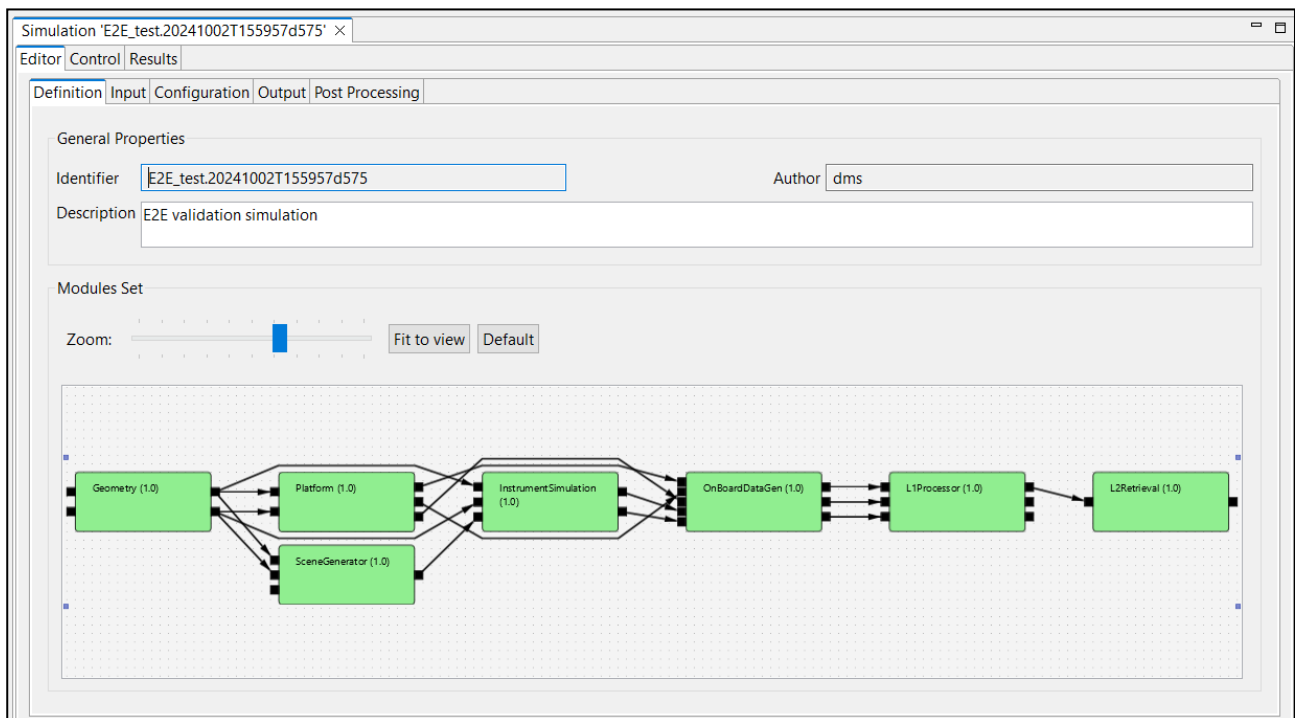
Figure 4-59 Execution results, definition tab

Some simulation data is presented in a "general properties" area (upper part of the tab) showing these attributes:

- Date / Time – this is the local computer date and time of when the execution began. This date and time can also be part of the simulation identifier to distinguish this simulation execution from others;

- Duration – the time (in minutes and seconds) elapsed from the starting time until the execution was finished or interrupted;

- Status – the overall status of the execution. The possible values are "Failed, "Successful" and "Aborted";

Below this area, the Results tab reports the log messages generated by the simulation execution. Users can access all these messages to check its performance.
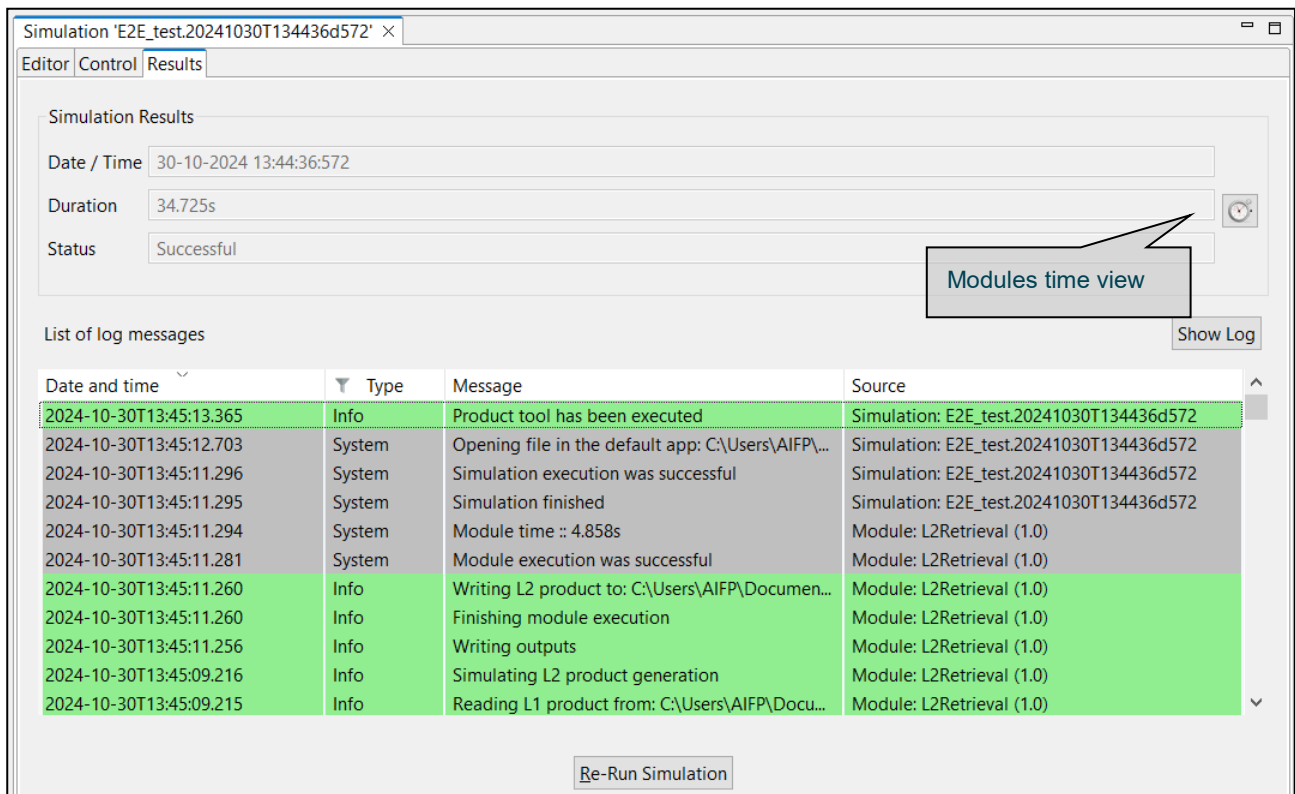
Figure 4-60: Execution results, results tab

Executions represent the dynamic view of the system. Here the executed simulations are stored with their input and output data. Users can consult their results and log messages generated, as well as re-run simulations as needed.

The list of all the executed simulations stored in the database can be accessed through the executions view of the side bar or via the "Executions → Manage" menu (Figure 4-61) from the main menu.
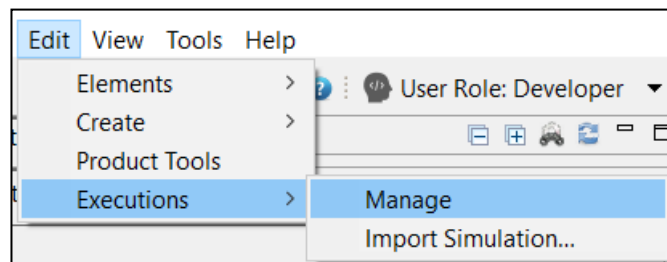
Figure 4-61: Results menu
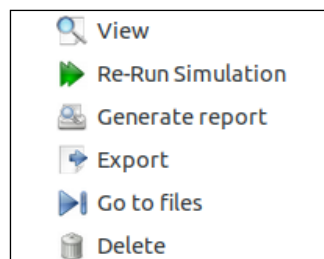
Figure 4-62: Results pop-up menu

The Executions tab of the side view uses a colour code to provide information about the differences between the connected database and the File System. If a result is present in the system, the execution is displayed in black,

![INDRA logo]

openSF

System User Manual

OPENSF-DMS-TEC-SUM01

4.9

18/06/2025

whereas if the result is present in the database but not in the system the item is displayed in light grey (Figure 4-63).
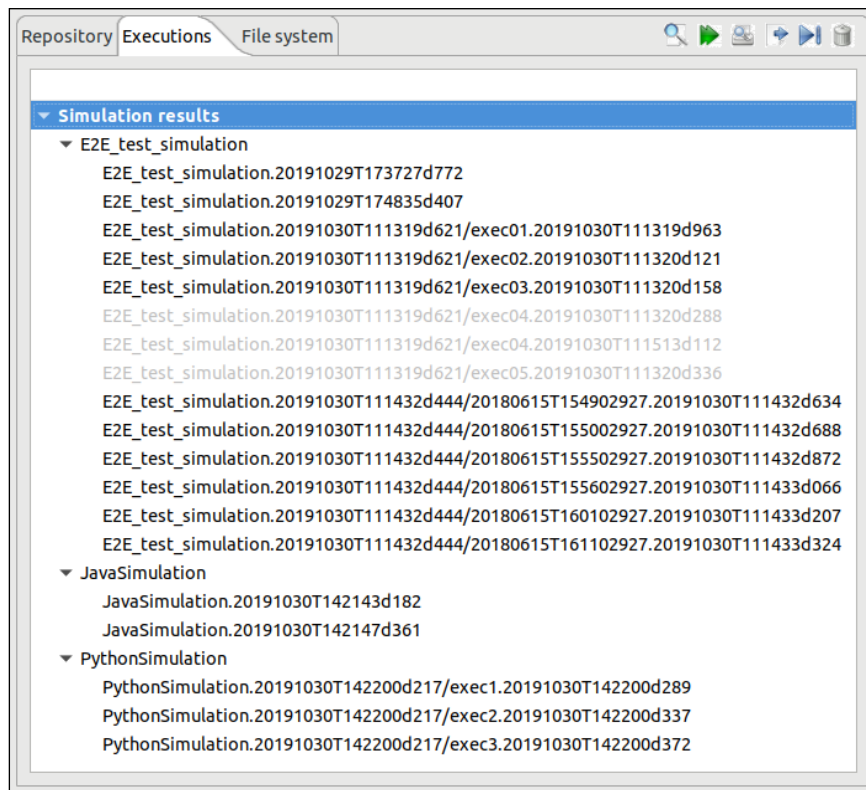
Figure 4-63: Executions view in the side bar

For each of the results of the "Executions" tree, a number of operations can be performed. These operations can be performed for a single result or for multiple ones simultaneously, by selecting more than one result with the Ctrl or Shift keys. Operations involving results include the following:

- *List* – present the list of existing execution results;

- *View* – consult the data of an existing execution result;

- *Re-run* – starts a new simulation execution. This new simulation is a replica of the former, but it is created in a new simulation folder.

- *Report generation* – shows a text report describing the execution.

- *Exportation* - exports the entire execution definition;

- *Go to files* – show the files of the selected execution in the File System tree of the side bar.

- *Deletion* – delete an execution result from the system. A dialog is prompt to the user to confirm the execution deletion from the database and, if desired, also from the file system.

4.3.4.1.1. Modules execution time

In Figure 4-60 the simulation results view is presented. Within this panel there is a clock icon button in the mid-right side that presents the time consumed by each module involved in a simulation run.

Module execution time is presented in a new window with three tabs:

- *Module Times* tab: presents the module execution time in a bar chart graph10. See Figure 4-64.

- *Time Statistics* tab: shows a pie with the simulation time percentage consumed by each module. In case there is more than one Simulation involving one Module, time is divided by the number of module repetitions. See Figure 4-65.

---

[10] Users can zoom within bar chart graph in order to better visualize module times

- *Module Times* Table tab: shows the same information that the bar chart graph but in a table. See Figure 4-66.
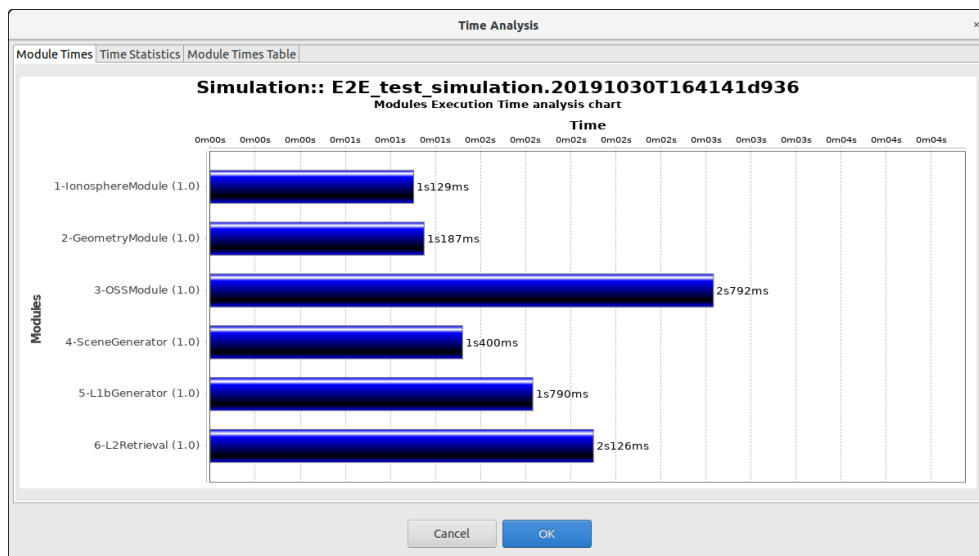
Figure 4-64: Bar graph showing module times
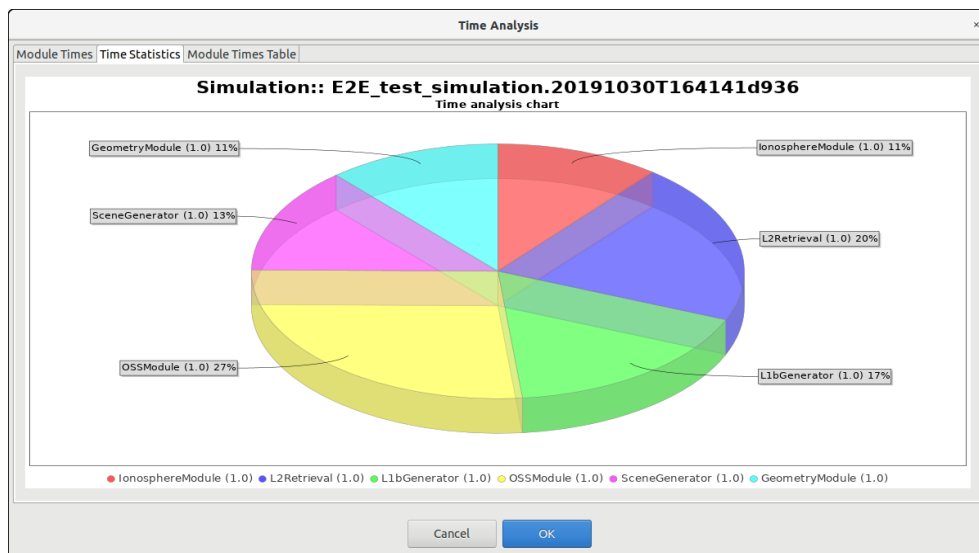
Figure 4-65: Pie chart showing the percentage of time

Figure 4-66: Table showing module times

4.3.4.2. Continuing or repeating the execution of an existing simulation

Accessing the "re-run" functionality, users can repeat the execution of a previously executed simulation. If the simulation execution was successful, the system just creates another execution (changing the starting date and time) but, if the previous execution was aborted or failed, the system will inform the user with the dialog shown below.
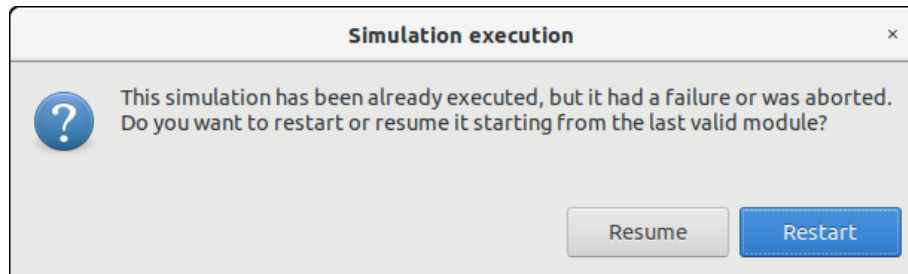
Figure 4-67: Result. Re-run

Users can now choose to restart the execution from the beginning or try to resume the execution, that is, to continue the execution from the last valid module. So, the execution will continue provided that the outer conditions that made the previous run fail have been corrected.

4.3.4.3. Report generation

Clicking on the "Generate report" option from the "Executions" context menu accesses this functionality. A window similar to the one shown in Figure 4-68 is presented to the user.
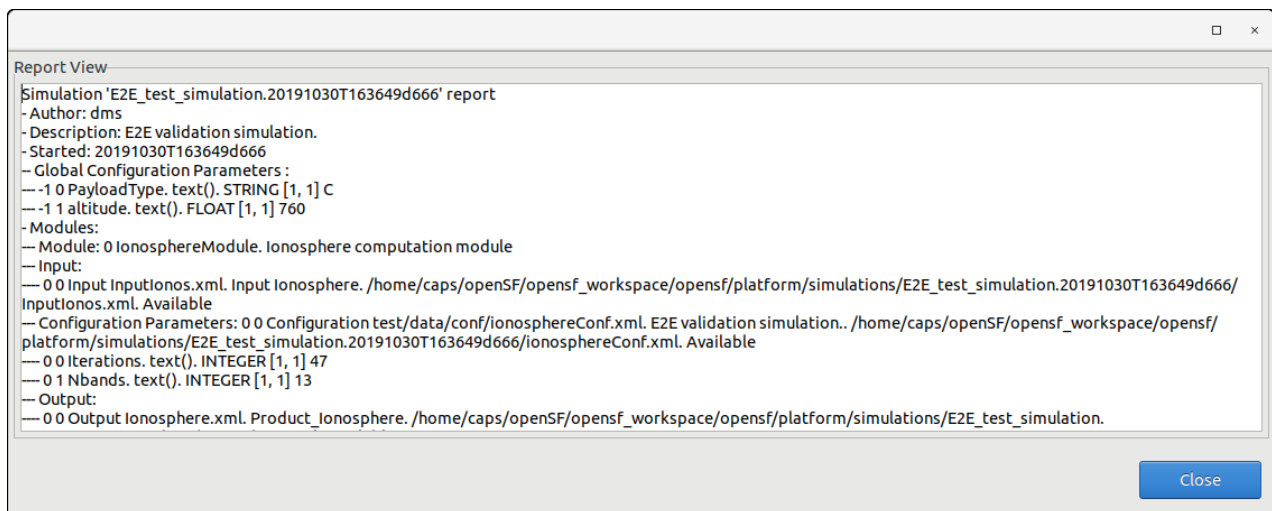
Figure 4-68: Execution report

This execution report consists in a textual description of the same data that users can access with the "Result View" functionality. The only difference resides in that this textual information can be copied and pasted into another application outside the openSF system.

4.3.4.4. Result deletion

Users can select a certain execution result and choose the option to delete it. Once users confirm the operation the execution result is erased from the repository and the file system. Log messages associated with this simulation execution result will also be erased.
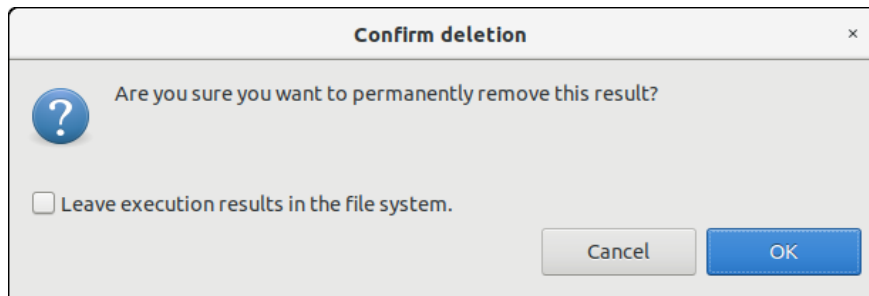
Figure 4-69: Confirmation dialog to delete execution(s) from database and file system

4.3.5. Product tools

E

As explained in section 1.4, a tool is an external program that performs a given action to a certain group of files. Used as part of the openSF framework and associated to a certain file extension, these tools can be called to perform a posteriori operations to products involved in simulations.

Tools are classified as "internal", if they are part of the openSF distribution and are located in the tools' directory, or "external" in other cases. Currently the openSF distribution package does not include any tool executables. However, there is a built-in tool named "OS default app" that is available for all file types and simply asks the OS to open the file with the default program. Built-in tools are not editable, or erasable.

To access this functionality, the user can use the toolbar or the menu bar (under Edit → Product Tools).
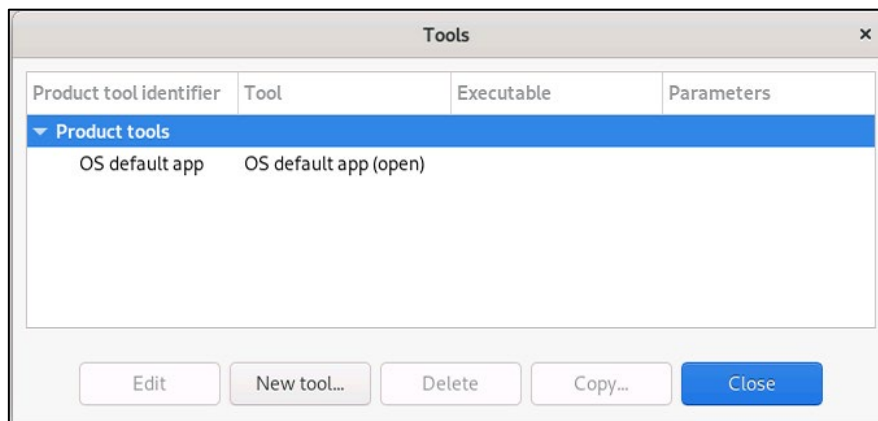
Figure 4-70: Tools list view

A list of tools, showing its identifier, action, executable and parameters is given. Tools are definable by the user. Thus, new tools can be added by clicking the "New tool" button.

4.3.5.1. New tool

Accessing to this functionality, a new window appears to let the user create a new tool, see Figure 4-71.

Users can define the following attributes:

Attribute name	Format	Purpose	Sample
Identifier	Normal string	This is a unique identifier of the tool.	post-pro tool
Action	Normal string	What the tool is going to do. It is expected to be an encoded representation of what the parameters do.	save to PNG
Extension	Short string	The type of files that this tool is going to be applied to.	.nc, .jpg

Executable	Medium string	Location of the executable file that is going to be called to execute the product.	python3
Parameters	Medium string	The list of parameters that will follow the executable. No variables can be passed from the HMI at this point.	-m repe_postpro –f $1 –o . --format=png
Description	Medium string	A brief description of what this tool will do and need. It is a longer explanation than the "Action" representation.	Save data and launches plot directly to PNG files in the same folder

Tools have to be a unique combination of "Identifier", "Action" and "Extension(s)" fields. Therefore, no two tools may have the same values for all those fields, even if they have a different description or binary file. For example, a different tool could have the same ID but a different "Action", with different parameters to perform the described action accordingly.

The "Extension" field states the type of files fed to the tool. Several extensions can be defined, in such a case they must be separated by comma.

Figure 4-71: Tool editor view

4.3.5.2. Edit tool

Selecting this functionality, the user can access and edit all the attributes of the selected tool.

4.3.5.3. Delete tool

Selecting this functionality, the user can delete the selected tool.

4.3.5.4. Tool execution

The process to execute an external data exploitation application can be triggered in three different ways:

- Tool execution from the file system view (see below).
- Tool execution from the simulation edit view and execution results view (see Figure 4-72). The execution view is the same as the simulation view with two particularities, 'Results' tab is enabled and the status is completed (successfully or not). Note that in the execution view the status of the output files is shown as *Available* if the simulation chain has been successfully executed while in the simulation editing the status is shown as *Pending* (see section 4.3.3.6.3).
- Scheduled execution over simulation data products, section 4.3.5.6.
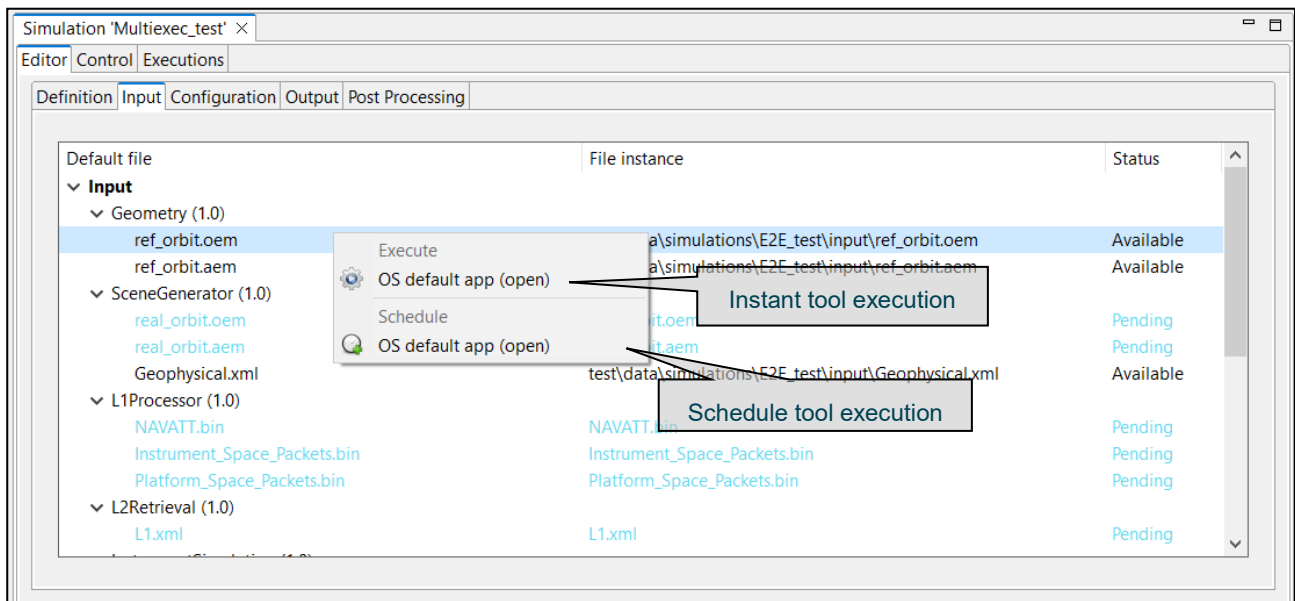
Figure 4-72: Tool Execution/Schedule from Simulation Edition View

To execute a tool from the file system view, the user has to right-click over a file name whose extension is associated to one or several product tools, a menu showing some actions will pop-up.
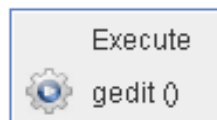
Figure 4-73: IO file pop-up menu

Once the desired action is selected, a dialog will show up asking the user for completing the executable command line. By default, openSF presents the location of the selected file.

Users can accept the default parameters or can add extra ones. Users can also make use of the environment variables supported by openSF (described in section 4.5.1), writing the dollar symbol and its name.

Once accepted, the parameters, the tool program will be executed in a separate thread (so the openSF operations are not interrupted).

For a real example of product tool execution see section 8.3.

4.3.5.5. Popular product tools

During the integration of openSF in E2E simulation projects the development team has identified a set of product tools widely used and that are freely available on the web. For every listed tool the operating system compatibility is also specified (Linux, Multi-platform...).

Image processing tools

Below these lines are listed a set of tools for viewing and editing image files. The applications listed support a large number of image formats.

Image Viewers

- ❑ IrfanView – Multiformat Image Viewer – Microsoft Windows
- ❑ Eye of Gnome – Gnome Image Viewer – Linux (GNOME)
- ❑ Gwenview – KDE Image Viewer – Linux (KDE)
- ❑ Okular – KDE Document and Image Viewer – Linux (KDE)

Image Editors

- ❑ GIMP – The free Adobe Photoshop alternative – Multi-platform
- ❑ Inkscape – Image editor with vector graphics support – Multi-platform

Text editors

- ❑ Notepad
- ❑ Emacs – GNU Editor – Multi-platform
- ❑ Notepad++ - Full-featured – Microsoft Windows

Scientific data formats

NetCDF

Network Common Data Form is a set of interfaces for array-oriented data access and a freely-distributed collection of data access libraries for C, Fortran, C++, Java, and other languages.

- ❑ Panoply – NetCDF data plotting – Multi-platform
- ❑ ncBrowse – NetCDF file browser – Multi-platform

HDF

Hierarchical Data Format, commonly abbreviated HDF, HDF4, or HDF5 is the name of a set of file formats and libraries designed to store and organize large amounts of numerical data.

- ❑ HDFView – HDF File viewer (images, tables–) - Multi-platform

Browsers

A web browser is a software application for retrieving, presenting, and traversing information resources on the World Wide Web.
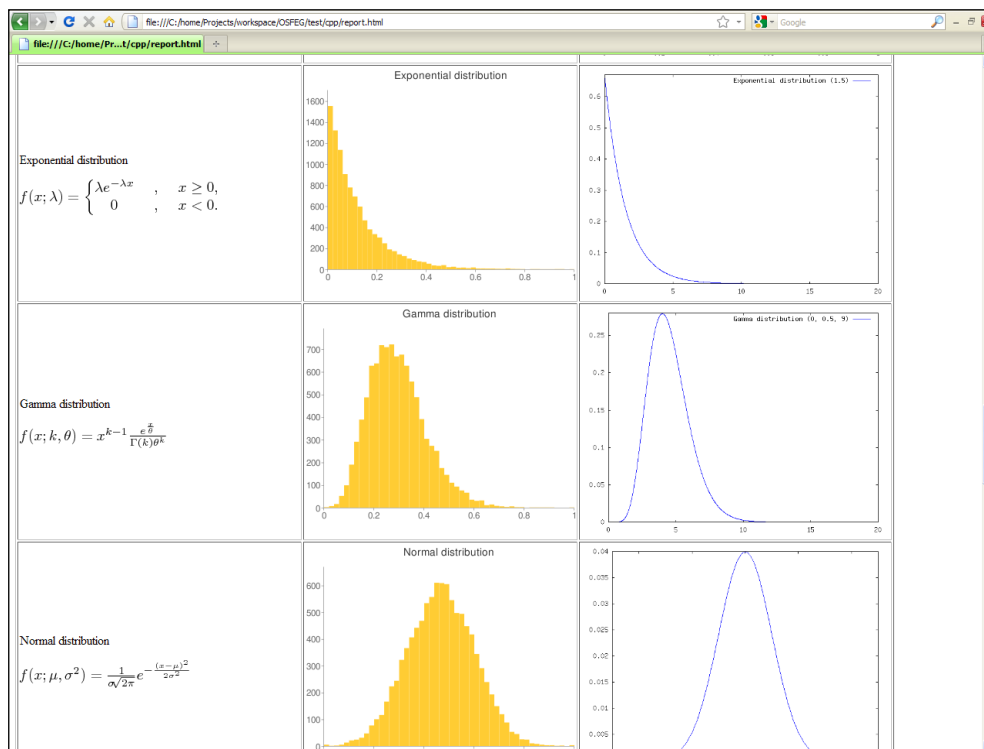
Figure 4-74: Web browser as openSF product tool

Example: Figure 4-74 shows the use of an Internet browser (Mozilla Firefox) for graphing the results of testing a random number generator. It uses Octave graphing capabilities and Google Charts API.

- ❑ Mozilla Firefox – Multi-platform
- ❑ Google Chrome – Multi-platform
- ❑ Opera – Multi-platform

Other tools

- • GNU Octave - If GIMP is the free photoshop-like choice, this is the equivalent for MATLAB - Multi-platform

- GNU Plot - GNU software that gives plotting capabilities through a command line interface - Multi-platform

4.3.5.6. *Specification of final product tools*

It is possible to add a list of product tools as post-processing operations, that is, a series of executables to be called upon the execution completion. This is done while creating a simulation (Figure 4-75).

There are two ways to add tools to this list. The first is by selecting a file from the input, configuration, or output files list. On any file marked as *Available* or *Pending*, the context menu of Figure 4-76 will show all tools that can be applied to that type of file[11]. Selecting the version of that tool under "schedule" adds the tool to the list. Alternatively, the user can click on the "+" button, which presents the list of defined tools for the user to choose one.
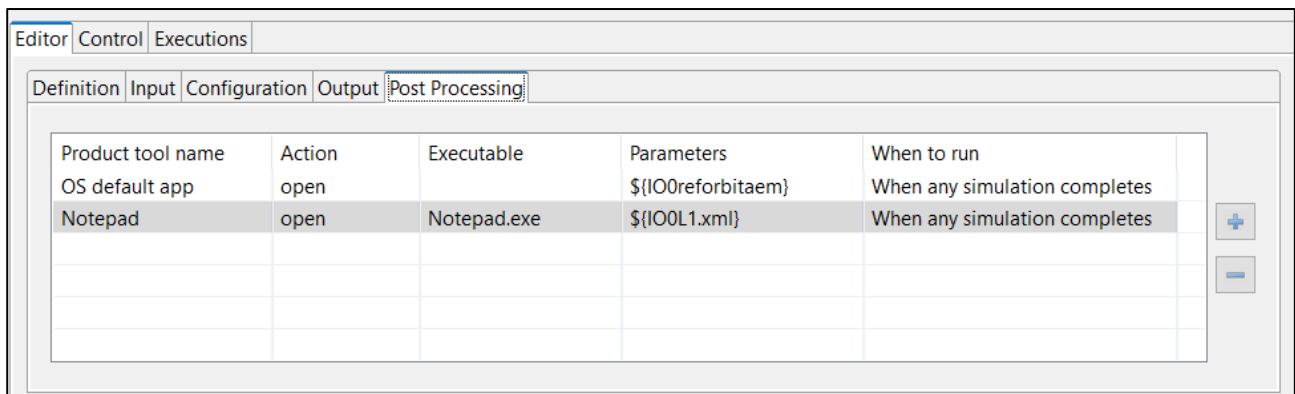
Figure 4-75: Product tools specification
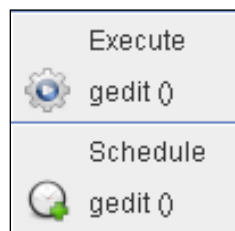
Figure 4-76: File contextual menu.

In either case, the user can change the parameter to be sent to the tool binary. When scheduling actions to certain files, openSF uses a placeholder reference that is later replaced with the actual location of the file. These placeholders are named starting with the dollar symbol, then "IO", then a number and the file name with no blanks, underscores or dots.

For example, ${IO0orbitxml} denotes the orbital file to be generated in the proper folder by the execution process. In the same way, users can include references to the rest of E2E-ICD-mandated environment variables like E2E_HOME. Note that the syntax is always ${NAME}, even in Windows.

Each tool in the post-processing list includes a mode that specifies when and how the tool should be executed. The possible values for this execution behaviour are:

- Disabled. The tool is defined but not launched by openSF automatically. This allows the user to schedule tools manually at a later time or from scripts.
- Whenever a simulation completes execution. The tool is launched immediately after each individual simulation finishes, regardless of whether it is part of a multi-execution or a standalone run. This was the only behaviour supported prior to openSF 4.6.
- Only when a full multi-execution completes. The tool is launched only once, after all simulations in a multi-execution set have finished. If the tool's parameter includes a placeholder (e.g., $IO0orbitxml), that placeholder will be replaced with one value for each simulation. Each value is passed to the tool binary as a separate argument, so the tool must be able to deal with an arbitrary number of arguments.

---

[11] Tools defined in the system have an action associated to a file extension.

Tools are only launched if the execution completes successfully. This applies to both individual simulations and multi-execution sets: if a single simulation fails—whether it is a standalone run or part of a set—no post-processing tools will be executed. In the case of multi-execution, tools configured with the "When a multi-execution completes" mode will not run if even one simulation in the set fails. This ensures that post-processing tools operate only on complete and consistent data.

All scheduled tools are launched simultaneously, and the simulation process waits for a few seconds for them to either complete or fail. If the wait expires, the tool is assumed to be running a GUI or a long process, and the simulation execution processes wind down e.g., the log file is closed.

Usage Example:

A simple simulation involving only one module execution. The scenario is composed by:

- Simulation name: *simulationTest*
- Module name: *moduleTest*
- Input for the module: *inputTest.txt* located in $E2E_HOME folder
- Output generated: *outputTest.txt*
- Module configuration: *globalConfig.xml* and *localConfig.xml*
- $E2E_HOME variable points to */home/tester/openSF/*
- The simulation folder is */home/tester/openSF/simulations/*
- Tool defined associated to txt extension: *meld*

It is desired to compare the input and output files with a visual *diff* like application named meld (http://meldmerge.org/). In this case the syntax for the tool would be:

- For the input file user can use the original location or the foreseen location where openSF copies that file: */home/tester/openSF/inputTest.txt* or *$IO0inputTesttxt*
- Foreseen location for output file: *$IO0outputTesttxt*
- As explained before there are different mechanisms to schedule the execution of the tool. The one recommended in this case is to right click on the output file whose status is pending and click on the "*meld*" tool under the *schedule* title. A pop-up window appears with a text field presenting the variable for the output file location *$IO0outputTesttxt*. The user shall complete the syntax for the tool appending the location of the input file [*/home/tester/openSF/inputTest.txt* or *$IO0inputTesttxt*].
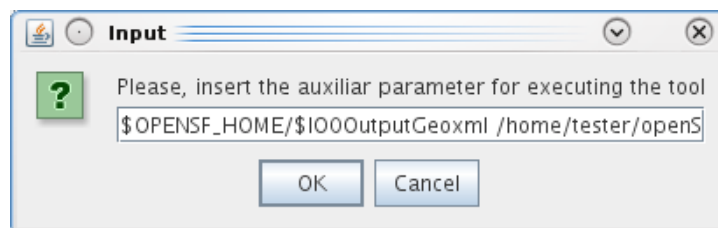
Figure 4-77: Tool parameters specification

- The value of the other simulation related variables would be:
- $E2E_EXECUTION_HOME = /home/tester/openSF/simulations/simulationTest\<DATE\>/, where \<DATE\> represents the date when the simulation was executed.

4.4. Executing a Simulation

M	I	A	E

Once a simulation has been defined, it is time to run it. This section describes this phase.

4.4.1. Execution settings

I	E

This section describes all the execution settings available in openSF.

4.4.1.1. Switch module version

This functionality allows the selection of a specific module version for a simulation execution (Figure 4-78).
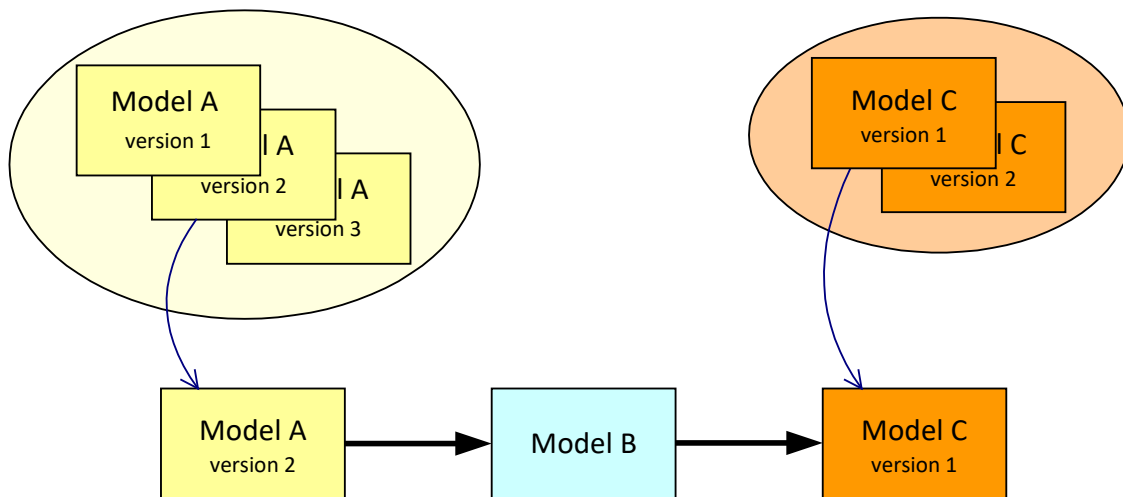
Figure 4-78: Module chain with different module versions

From the openSF HMI, the switch module version operation can be invoked from the simulation execution window by navigating down to the module the user wishes to alter the version of. Next, right-clicking over it. Whenever the module has more than one version available the "Switch module version" option appears for selection. This is illustrated in Figure 4-79 below.
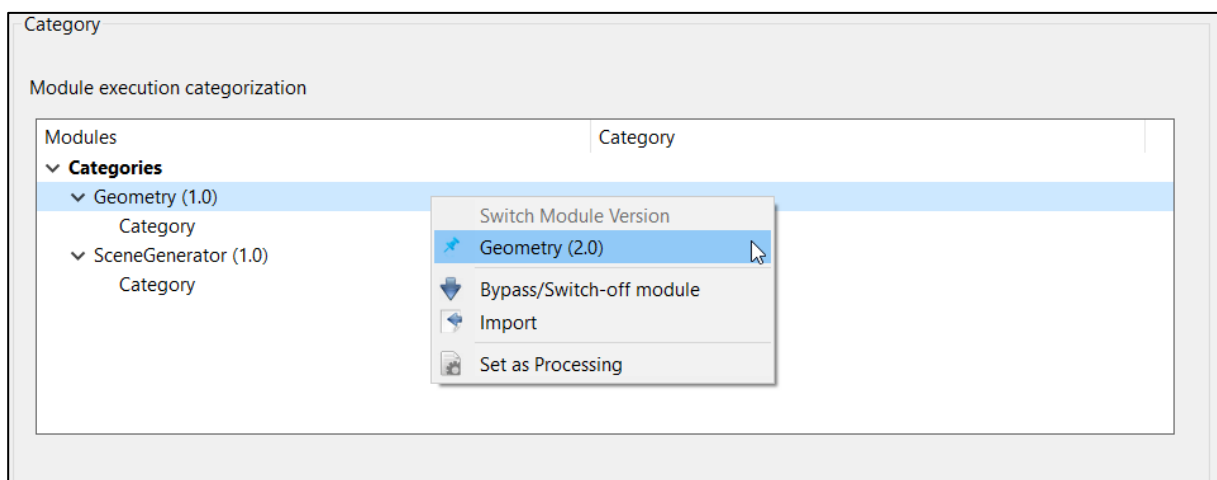
Figure 4-79: Switch module version

When a module version is changed, the system automatically compares the input and output descriptors between the old and the new version of the module. This comparison makes it possible to identify files that are kept, new files that are added and files that have been deleted in the new version.

The process follows the following rules:

- If a file exists in both versions, its current connection status is retained.
- If a file is new in the selected version, it is added to the module marked as disconnected, so that the user can configure it later.
- If a file that existed in the previous version is no longer present in the new version, it is safely disconnected to avoid inconsistencies.

In the event of major discrepancies between descriptors, such as the presence or absence of files that affect the configuration, a warning is displayed to the user explaining that certain files will be lost or disconnected and asking if they wish to proceed with the change anyway.
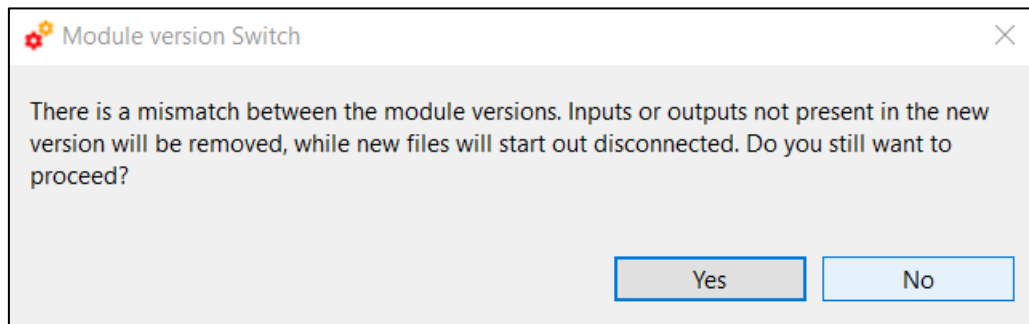
Figure 4-80: switch module version dialogue

After completing the change, a summary message is generated indicating which ports have been added and which have been removed.
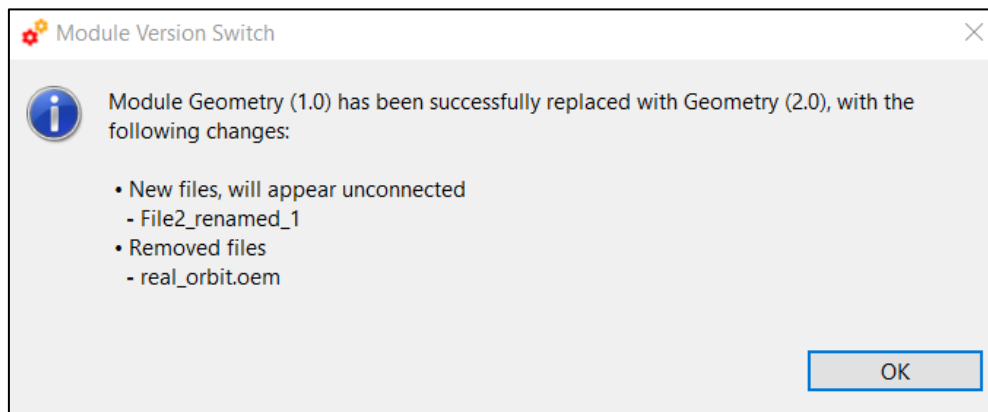
Figure 4-81: Information dialogue indicating which ports have been added and disconnected

4.4.1.2. *Bypass/Switch-off module execution*

This functionality enables users to switch off certain modules when running simulations.

From the openSF HMI, the bypass/switch-off module operation can be invoked from the simulation execution window by navigating down to the module that the user wishes to bypass. Next, right-clicking over the "Bypass/Switch-off module" option, as illustrated in Figure 4-82 below. This choice is persistent, if the simulation is saved to the database.
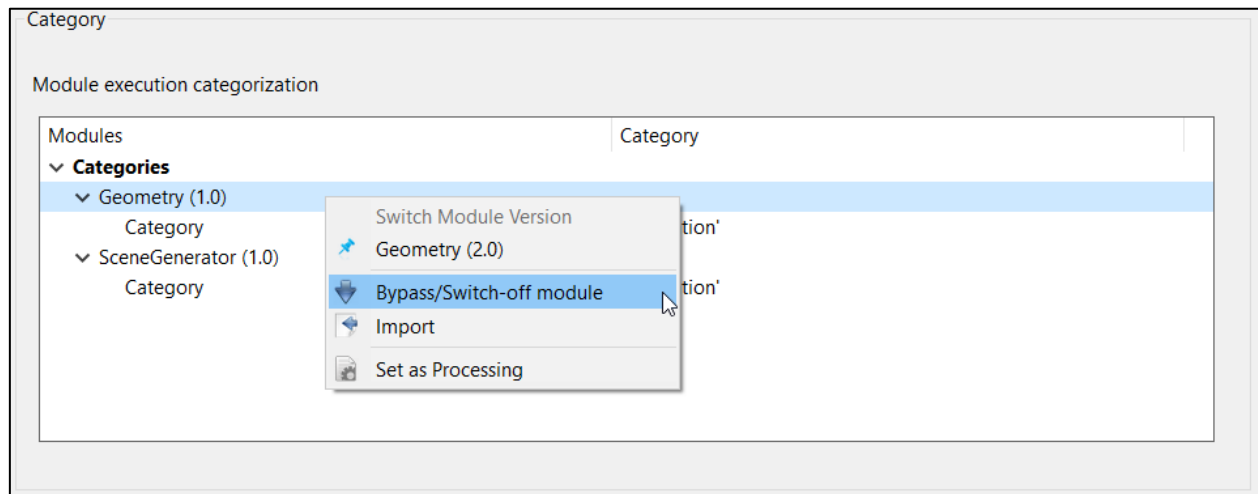
Figure 4-82: Bypass/Switch-off module

As a result, openSF will inform the user of the change in the list of data files needed to be provided due to the omission of modules and their corresponding outputs. Some inputs will no longer be needed, but other files may become "missing" instead of "pending" if a module that was going to generate them as its output becomes inactive.
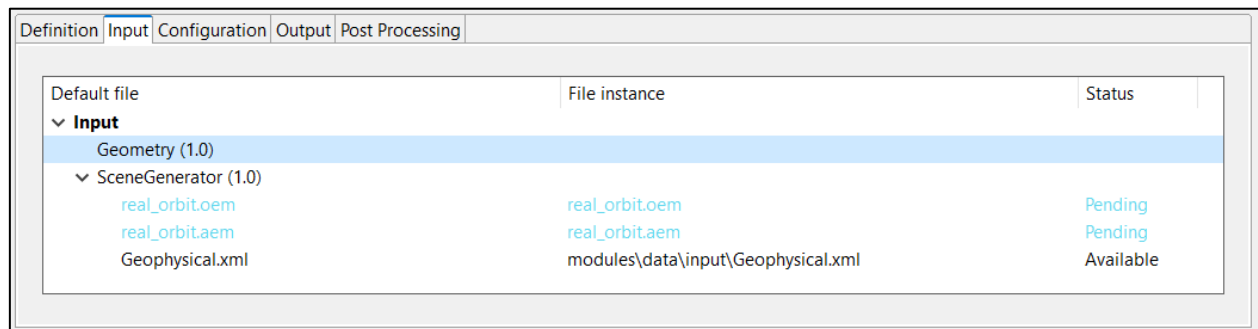
Figure 4-83: Bypass/Switch-off module missing files

The user can revert the bypassed module and switch it back on by navigating down to the module that the user wishes to re-activate. Next, right-clicking over the "Switch-on module" option that appears for selection.
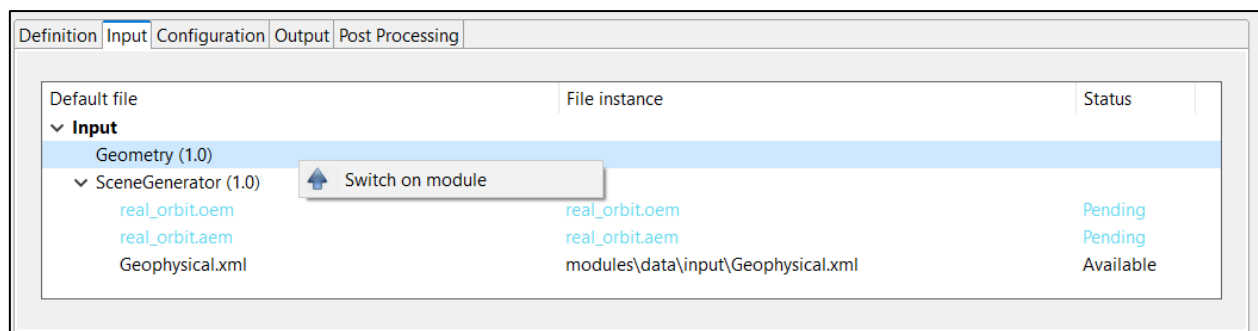
Figure 4-84: Switch-on module

4.4.1.3. Run using previous data

When re-running a simulation execution, the user can use the input data from the original simulation execution or use the input data produced in the simulation execution.

The user can revert the input and configuration data to the one originally used in the simulation definition by navigating down, either in the "Editor/Input" tab or in the "Control/Execution/Category" one, to the module that

the user wishes to reset the IO descriptors. Next, after right-clicking over it, the "Reset setup" option appears in a context menu for selection.
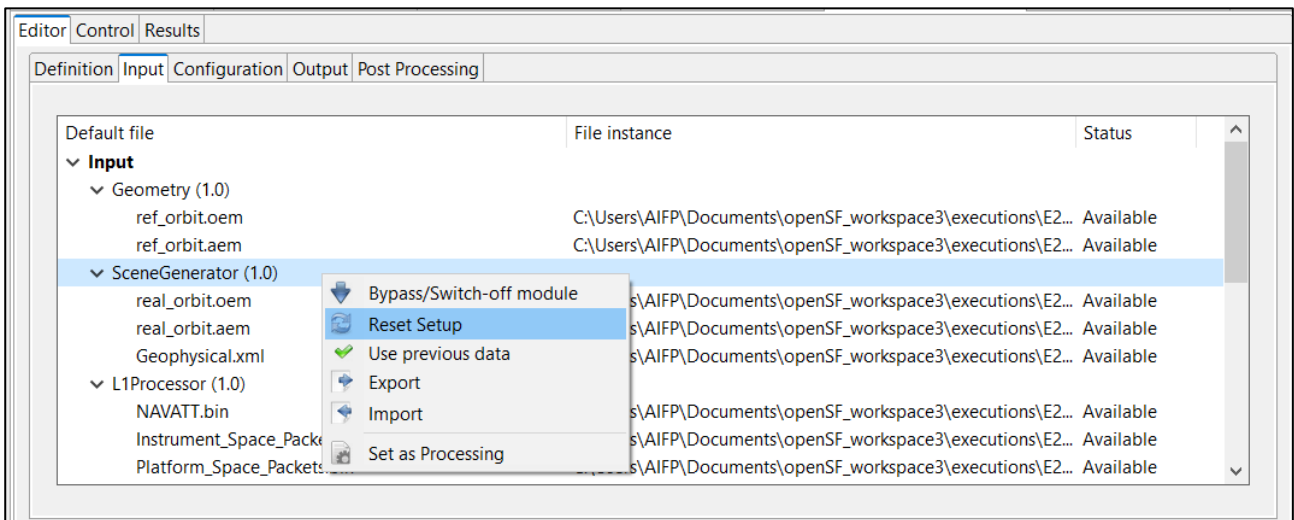
Figure 4-85: Reset IO descriptor option

As a result, the simulation's original definitions are now re-established.
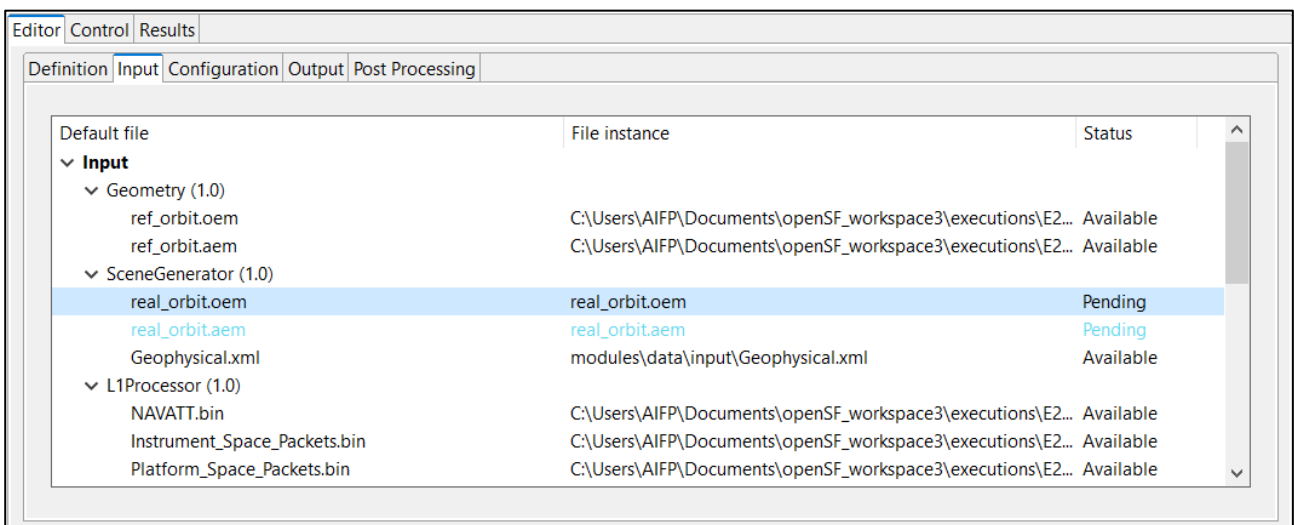
Figure 4-86: Reset IO descriptor setup

Furthermore, the user can revert to the data produced in the simulation execution for an individual IO descriptor by navigating down to the descriptor that the user wishes to restore. Next, right-clicking over it the "Use previous data" option that appears for selection.
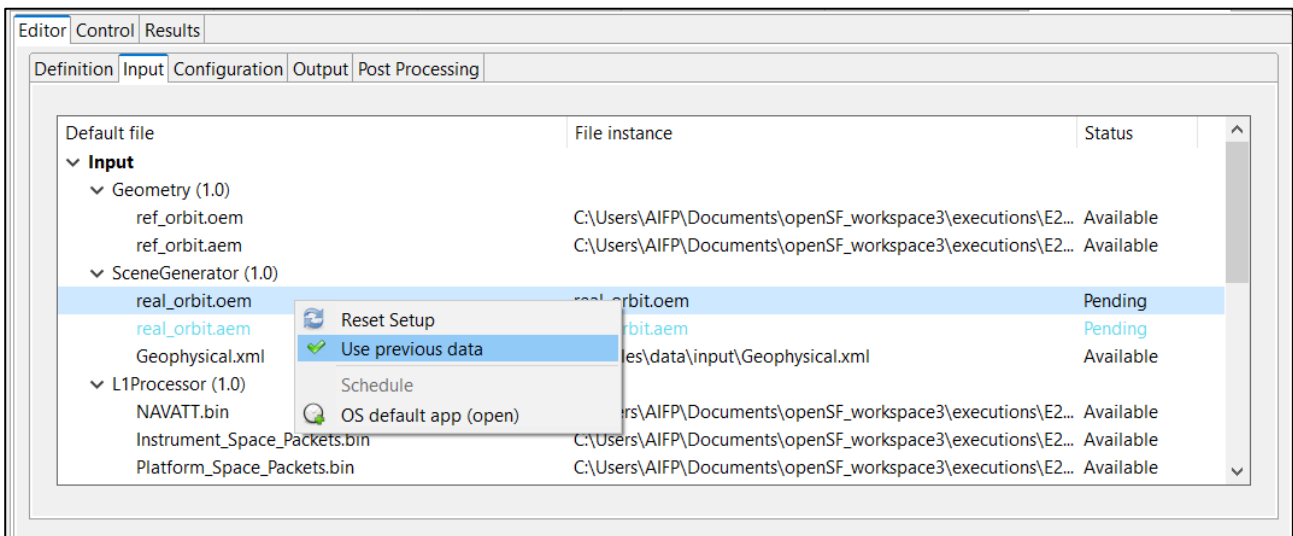
Figure 4-87: Use previous setup IO descriptor options

4.4.1.4. Removal of intermediate output files

As shown in Figure 4-88, the user has the option of removing intermediate output files. By activating this option, a simulation executing will remove from the simulation directory any output file not generated by the last module of a simulation execution and that correspond to intermediate data of a step in the module chain.

4.4.1.5. Breakpoint scheduling

Users can schedule breakpoints during the simulation execution. A breakpoint is a point where the simulation execution shall stop in a controlled manner, due to system architecture constraints it is only possible to interrupt the execution when a determined module has finished the computation and has written the corresponding output.

The user interface for breakpoints addition can be found in the "Control/Execution" tab.
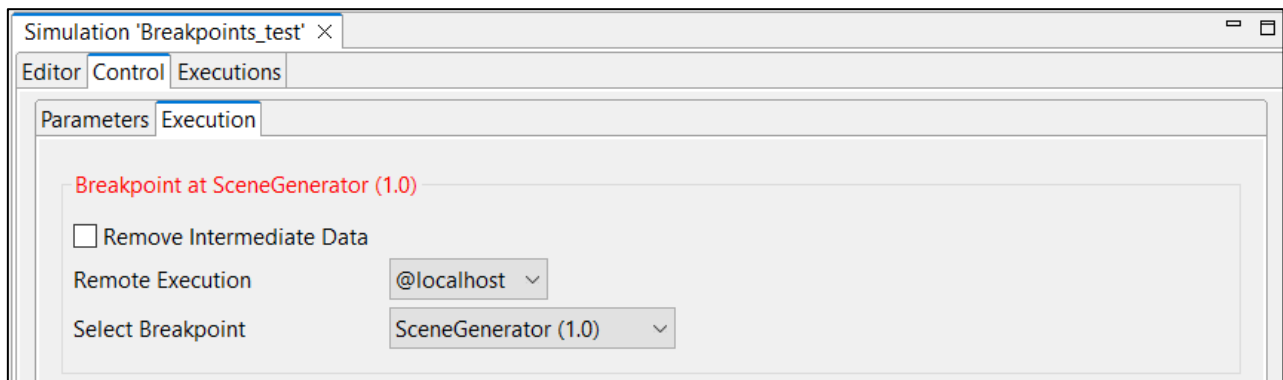
Figure 4-88: Breakpoint scheduling interface

In order to schedule a breakpoint before a module, users shall select its identifier in the "Select Breakpoint" drop-down list box. To remove previously defined breakpoints the user shall select the option "Remove Breakpoint" from the same widget.

Once a simulation execution is interrupted with the breakpoint scheduling system it is possible to resume the paused simulation and continue with the simulation chain keeping the same settings as the previous run (see section 4.4.3.2).

4.4.1.6. Remote execution

When executing a simulation, the user can select a remote machine where to execute it. This configuration can be applied:

(a) to the whole simulation: selecting a remote machine (previously configured) in the "Remote Machine" drop down list box of the simulation execution window;

(b) on a module by module case: selecting for each module a remote machine from the contextual menu obtained when right-clicking over the module listed in the simulation setup pane of the simulation execution window.

If both mechanisms are used, the system assumes that the last configuration selected overrides the previous one. Therefore, selecting a machine for the whole simulation overrides previously configured machines per module; as well, configuring a remote machine module by module after configuring the whole simulation may allow a finer configuration with possibly less configuration effort.

See section 4.4.6 for remote machine configuration details.

4.4.2. Series of simulations with parameters variation

A	E

From the parameter configuration tab, a series of simulations with varying values of the module parameters can be implicitly defined. openSF provides four different mechanisms to do this:

- Iterations
- Batch
- Perturbation
- Timeline

All configured modes, active or not, are saved to the database together with the simulation. However, they are mutually exclusive on execution: simulations may have multiple modes configured, but only one can be used to launch a simulation at a time.

When running multiple executions, the settings for the specific configuration mode that has been selected for launching the simulation will be saved in the multi-configuration folder.

As shown in Figure 4-89, by default the "Basic" mode is selected for a simulation. This means that the current simulation does not have any parameter variation, and a single simulation with the parameters specified in the screen will be executed. In this mode it is not possible for the user to modify, import or export the parameters of the multi-execution mode.

Otherwise, if the user selects any other simulation mode, for example when defining an iterative configuration, the configuration buttons become available and it is possible to save this definition to a file. This is achieved by using the "Export" button. In addition, the user can load a previously defined multi-execution configuration from a file (using the "Import" button).
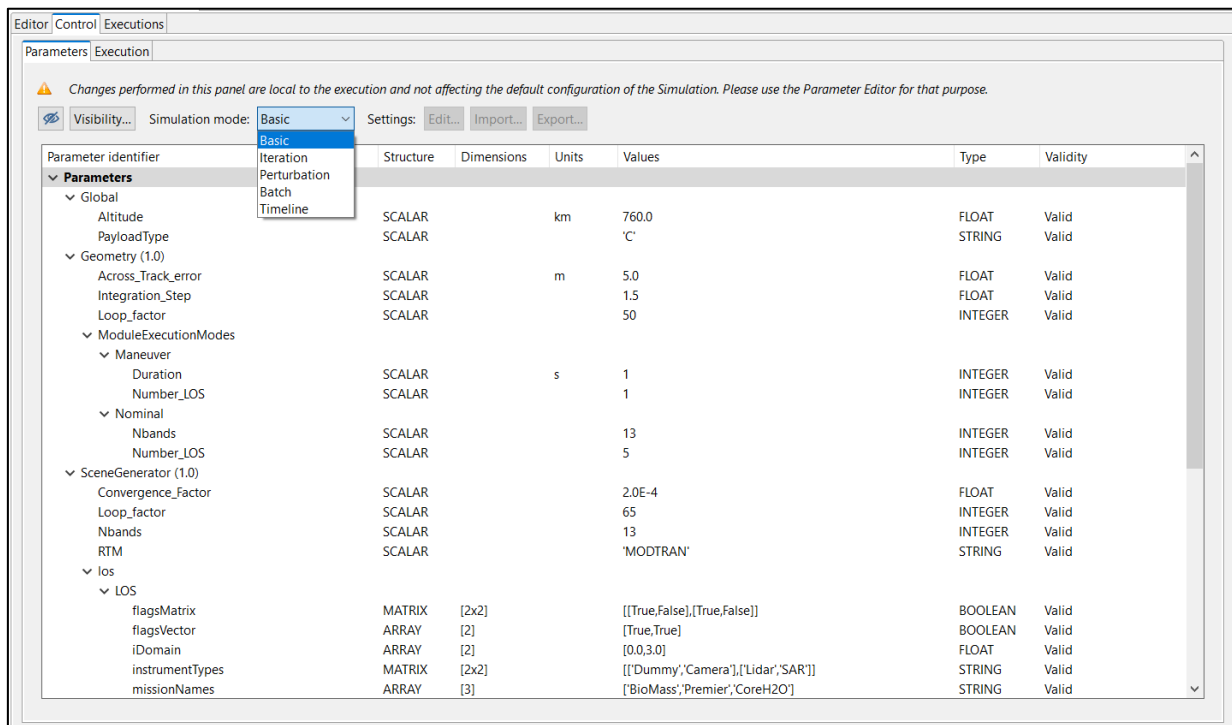
Figure 4-89: Parameters view

4.4.2.1. *Parameters iteration*

Users can assemble iterative simulations. This is a powerful feature that helps to run a large number of simulations by changing values of the parameters. Users can alter any parameter's value to fine-tune the behaviour of a module for a particular simulation run.

From the "Parameters" tab of a simulation view, selecting one or many parameters and choosing the "Iteration" option as simulation mode will open the dialog shown in Figure 4-90. In this example two float parameters and a scalar parameter from three different modules are being iterated.
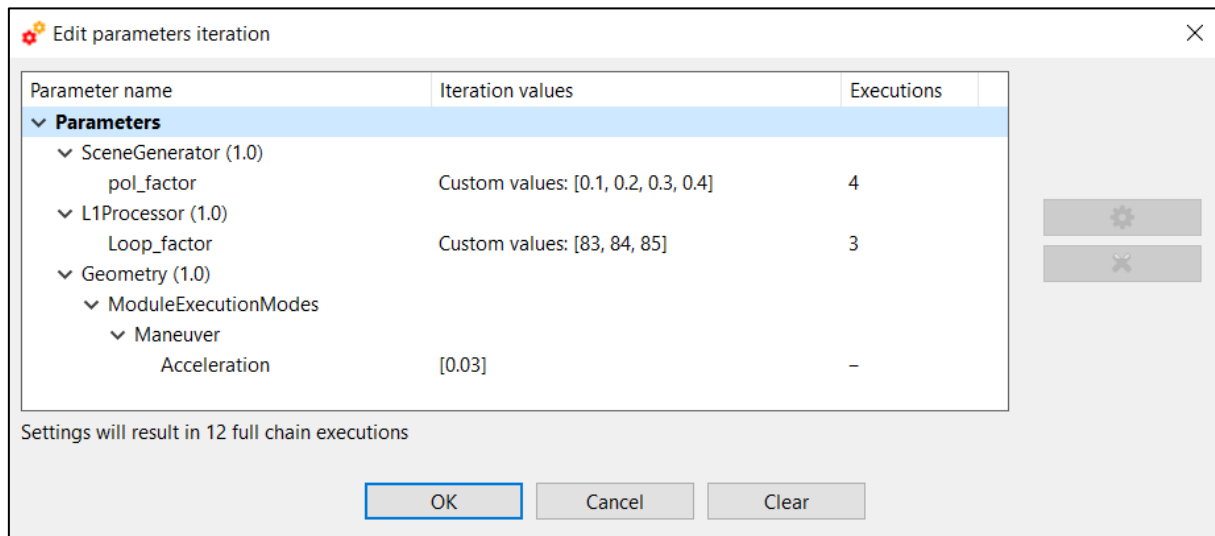
Figure 4-90: Iterating parameters

This figure shows the initial state of the dialog. The list of the selected parameters is shown in a tree configuration. The "executions" column states the number of executions for each parameter, while the label at the bottom states the total number of executions resulting from the configuration. The "*Clear*" button will reset all parameters to their previous values, removing all existing iterations.

The "Iterations values" column of the table represents the numeric sequence if any or the parameter value. This column cannot be edited directly. By selecting a parameter and clicking the cogwheel icon, the user can open the following numeric sequence generator dialog for customization of the iteration values:

Figure 4-91: Editing numeric sequences

When creating a new arithmetic sequence for a parameter value, the start and step fields have a validation error because they are initialised with the same parameter value (see Figure 4-92). This is to indicate to the user that he must edit these values as end has to be greater than start. The start and end fields accept any finite value. On the other hand, the step field only accepts positive integer or positive double values, while the division field only accepts positive integer values.

Figure 4-92: Creating numeric sequence

This dialog lets the user define a numerical sequence of values (of the selected type: FLOAT or INTEGER) in two different ways:

- Numeric sequence by step: defined by the starting (x_1) and ending (x_n) values of the sequence, plus the value of the step (s) entered in the bottom text field. The sequence follows this rule:

$$a_i = x_1 + is \text{, with } i = 0, 1, \dots, \left\lfloor \frac{x_n - x_1}{s} \right\rfloor$$

Values will never be <u>greater</u> than the upper limit. For example, a numeric sequence starting from 1 to 15 with a step of 5 will generate a series of [1, 6, 11], with value 15 not included.

- Numeric sequence by division: defined by the starting (x_1) and ending (x_n) values of the sequence, plus the number of divisions (N) entered in the bottom text field. The generated arithmetic sequence follows this rule:

$$a_i = x_1 + is \text{, with } s = \frac{x_n - x_1}{N-1} \text{ and } i \in [0, N)$$

Thus, the sequence always contains N points that are equispaced between x_1 and x_n. For example, a numeric sequence starting from 0 to 8 with N=5 will generate a series of [0, 2, 4, 6, 8].

Users can now accept or cancel the numerical sequence.

Note that parameters not involved in the iteration will remain fixed to a value but they can be manually changed as seen in section 4.3.3.6.6.

The iterated parameters will be highlighted in the simulation parameters' tab as shown in Figure 4-93 and iterate view can be opened again for more customization with a double-click on an iterated parameter.

Figure 4-93: Simulation with iterated parameters

Once an iteration configuration is established, any changes made to the configuration files (e.g. deleting parameters) may show an error message. For further details, please refer to cection 4.2.2.7.

4.4.2.2. *Batch simulation*

In cases where the user needs to run multiple instances of a simulation with a variation of the parameter values not covered by the mechanism described in the previous section (Sec. 4.4.2.1), openSF enables the definition of a batch execution. This ability aims at providing fine grained control over the number of executed variations of the original simulation and the parameter values that are customized for each of them. In particular, each execution is defined explicitly instead of generating combinations from the values given in an iteration configuration.

The configuration of a batch simulation requires the creation of a batch simulation configuration file. The syntax and format of this type of file is described in [AD-E2E]. This file can be loaded selecting the *"Batch"* option as simulation mode in the simulation's parameters tab.

Upon loading the file, openSF checks the format to guarantee its validity and imports the contents. The simulation will be successfully configured and a confirmation message will be displayed with the total number of simulation executions after the simulation template spawning together with the ID and module names of the overridden parameters (see Figure 4-94).
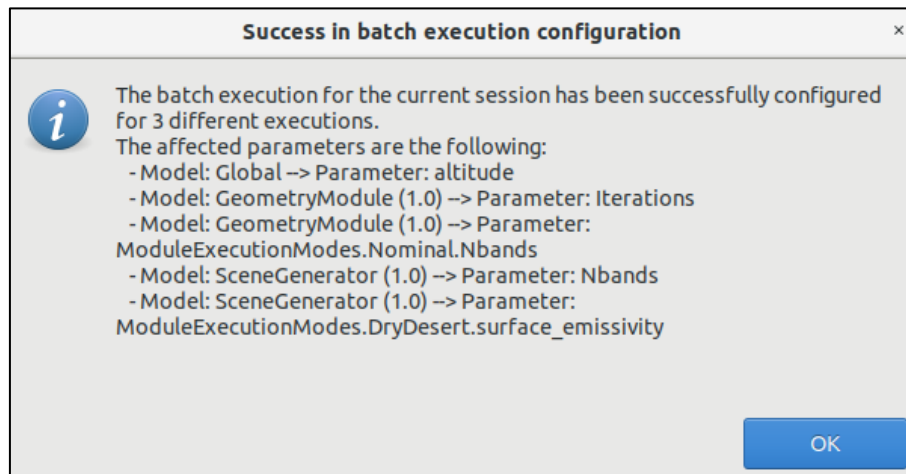
Figure 4-94: Successful batch configurated simulation message

As it was the case with the overridden parameter values configured through the "Iterate" option, the overridden parameters are highlighted in the simulation's parameters tab as shown in Figure 4-95.
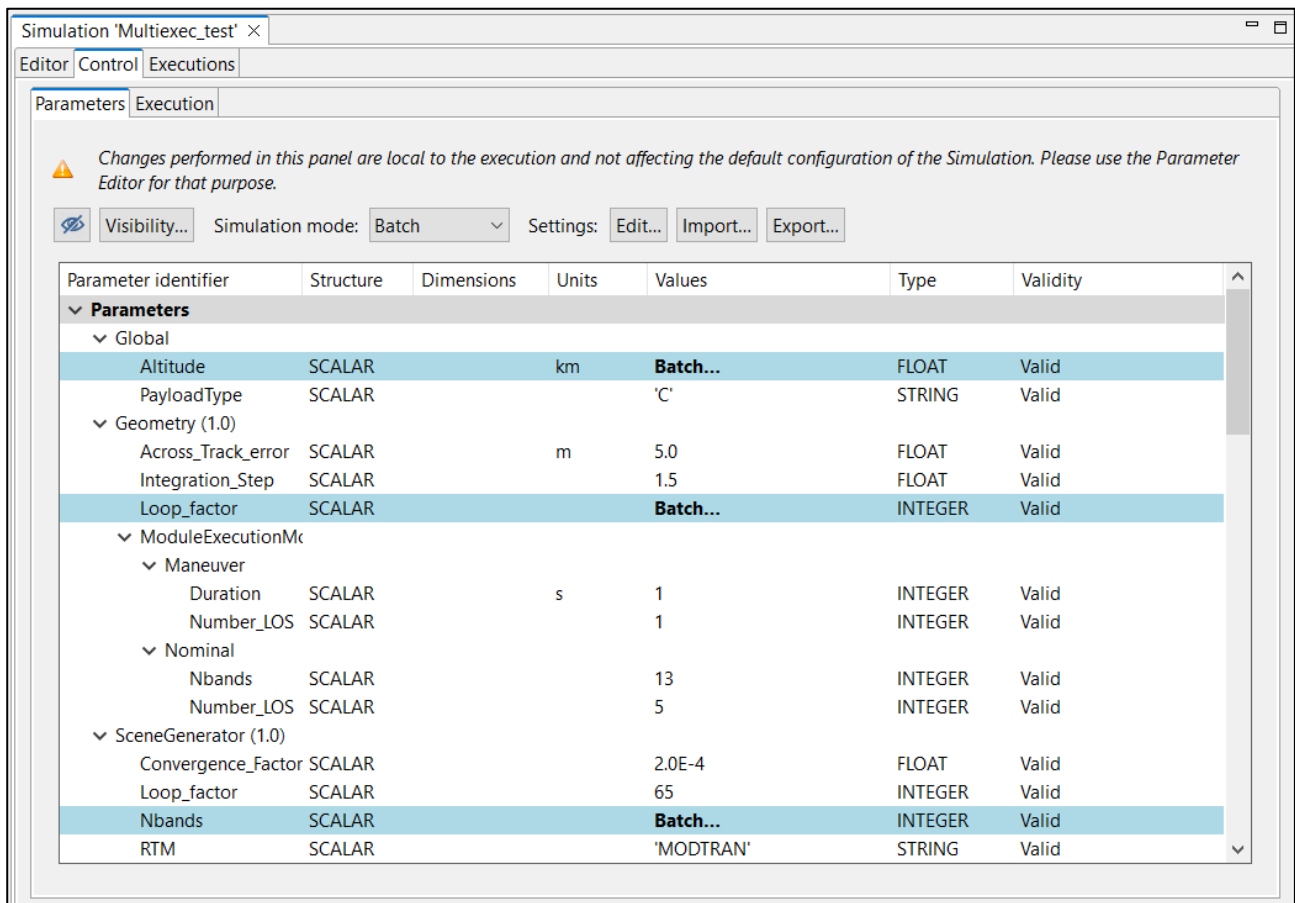
Figure 4-95: Simulation with overridden parameters through the batch option

To ease the use of this feature, together with the examples of the validation database, an example of a *"batch simulation configuration file"* for the *"E2E_test_simulation"* is provided and can be found inside the example simulator package at the following path: *$<OPENSF_INSTDIR>/resources/simulators*.

If the user tries to edit such simulation, the dialog shown in Figure 4-96 will appear. Users can import a new configuration, clear the parameters restoring their previous value or cancel the edition.
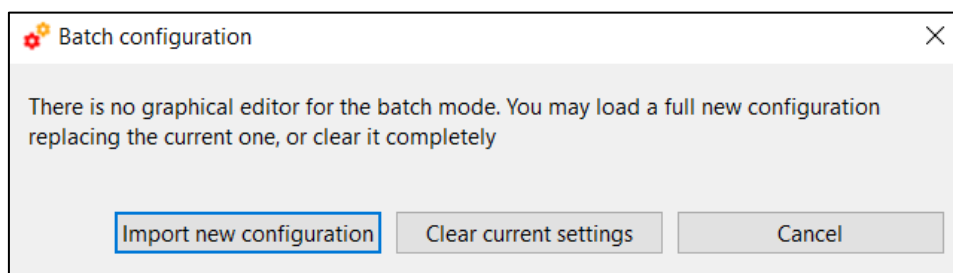
Figure 4-96: Editing a batch simulation

Once the batch configuration is established, any changes made to the configuration files (e.g. deleting parameters) may show an error message. For further details, please refer to section 4.2.2.7.

4.4.2.3. *Parameter perturbations*

The simulation perturbation system brings to the users the following functionalities:

- Independently define perturbation functions for configuration parameters, for each module involved in a simulation.

- Combine different functions for generating parameter values. An example would be a parameter taking values drawn from $A \sin(\omega t)$ where A is itself a random variable with a Gaussian distribution, while ω is a constant and t is an independent variable running linearly between two values.

- Two different execution schemes, Statistical and Combined mode

4.4.2.3.1. Parameter perturbation interface

The parameter perturbation interface is divided into left and right panels. The left panel represents the simulations elements to be perturbed: root, module, parameter and perturbation node. For the latter, a description made up of the role of the node in its immediate parent is used, followed by the node type.

The right panel defines the properties at simulation, module, parameter and perturbation node level. When nothing is selected in the left panel, just an explanatory label.

- Simulation level: allows choosing between the two execution modes and, for the full simulation chain mode, editing the number of executions of the simulation perturbation configuration.
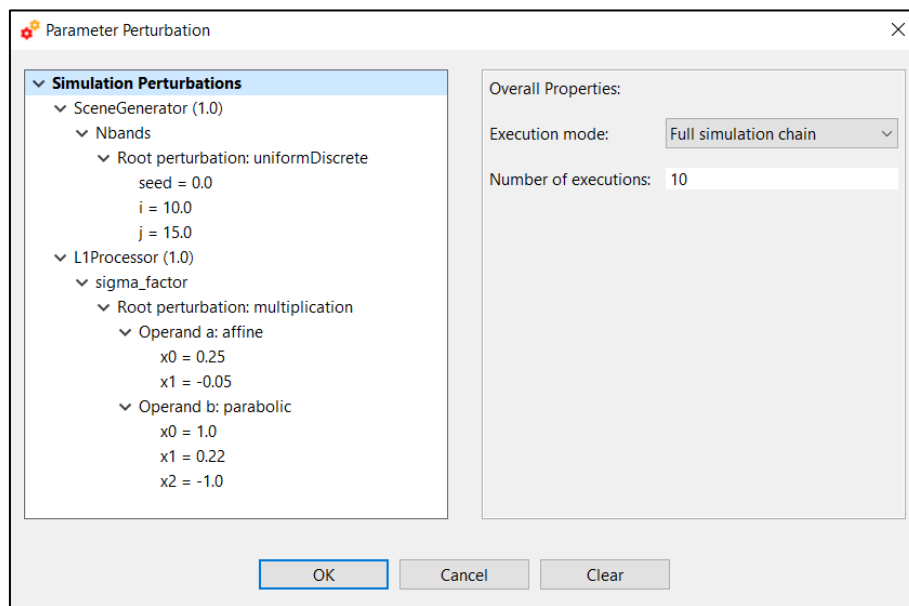
Figure 4-97: Perturbation system main window

- Module level. The properties panel shows the module name followed by the number of executions according to the currently enabled configuration. Note that the number of executions is only editable in per-module execution mode. In this panel it is possible to edit the bounds of the independent variable min and max for analytical perturbation. Values of this variable will be drawn linearly in this [min, max] interval according to the number of shots. For example, with N=5 and an interval [0, 1], the five values would be 0, 0.25, 0.5, 0.75 and 1.
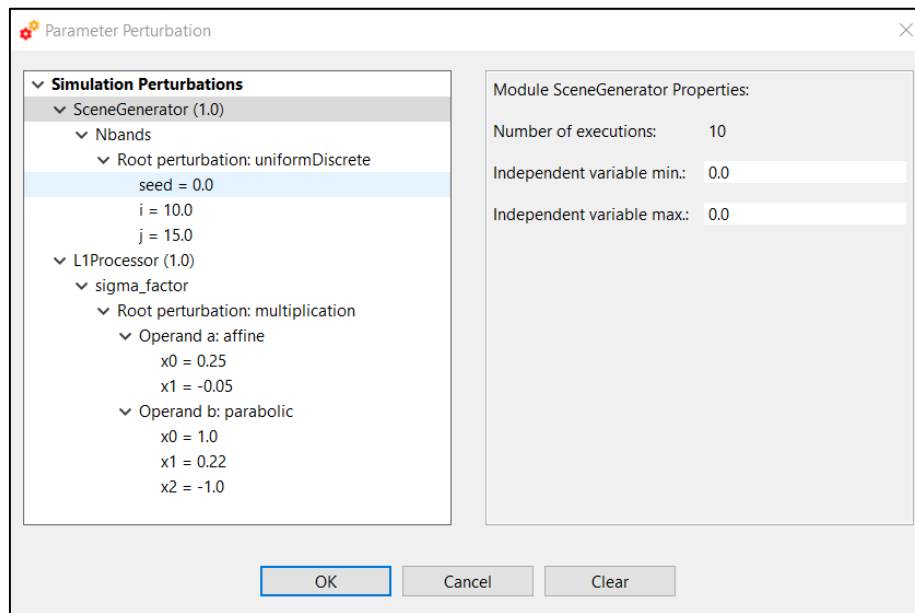
Figure 4-98: specific view of perturbations at module level

- Parameter level. At the top a description with the name of the parametr in the module. If the parameter is not curently perturbed, a button to add a perturbation node is availabele. Otherwise, a button to clear the perturbation/function on the selected item restoring the original single value.
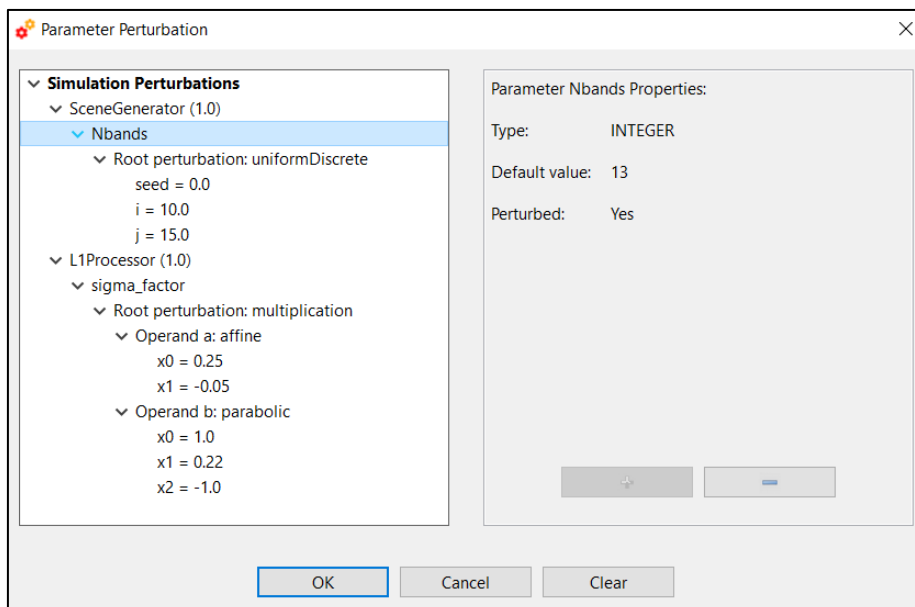
Figure 4-99: specific view of perturbations at parameter level

- Perturbation node level. At the top the type and subtype of perturbation. Then a table if it is a function with a variable number of properties or a single row if the function type is a single constant value. Double clicking a line edits it if it represents a constant value. At the bottom of the table "+/-" buttons in nodes that can take a variable number of entries (e.g., interpolation tables), add a new node before the selected row, or delete the selected row.
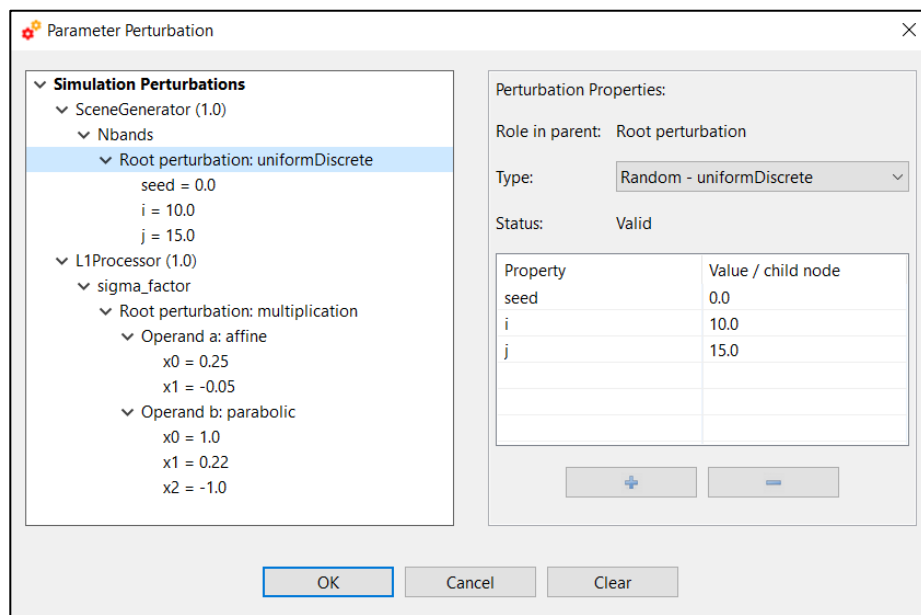
Figure 4-100: specific view of perturbations at perturbation node level

4.4.2.3.2. Defining a new perturbation

This section describes the steps that an user shall follow in order to add a new perturbation to a simulation.

1. Select the "Perturbation" option from the simulation execution mode drop-down box.

2. Select the desired INTEGER/FLOAT parameters and click on "Edit". The user may select multiple parameters using the Ctrl key. Global configuration parameters cannot be perturbed. If no valid parameters are selected, the following message will appear.
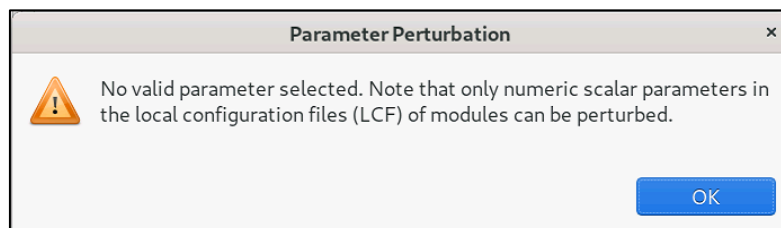
Figure 4-101: No valid parameters selected

3. Select a Parameter node and click on the "+" button from the parameter properties panel. This action will add the corresponding perturbation to the tree.

Figure 4-102: Adding a perturbation function to a module parameter

4. In the perturbation node properties panel insert the function parameters. At the end of this operation the status should be valid. Note that perturbations themselves can be nested.

5. Change number of executions, independent variable min/max as desired

6. Accept the simulation perturbation, adding it to the simulation

Just like in other modes, the "Clear" button at the bottom of the window restores all parameters to their unperturbed status, removing the configuration if the dialog is stored immediately after.

Additionally, perturbations can be graphed by right clicking the perturbation root and selecting "Plot". A new window will appear showing the time series and histogram for each perturbed parameter. Following figures Figure 4-103 and Figure 4-104 show how a parameter perturbation can be plotted from HMI.
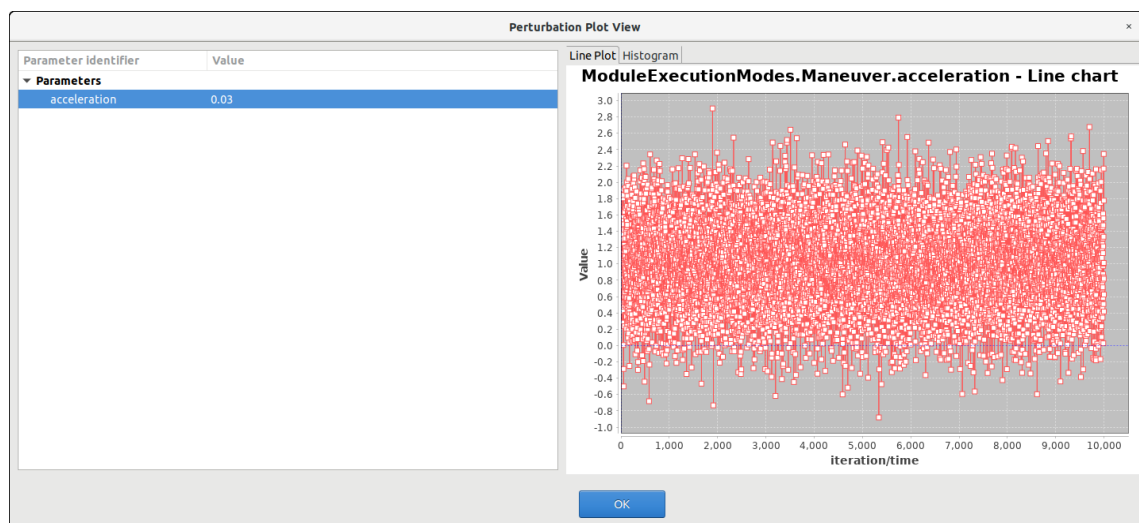
Figure 4-103: Time series line for a parameter perturbation
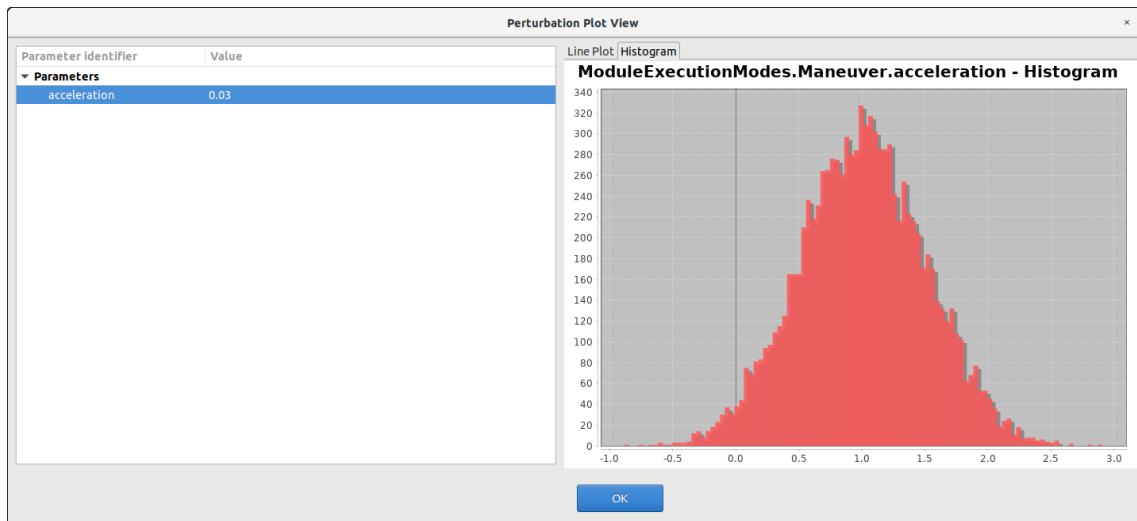
Figure 4-104: Histogram chart for a random parameter perturbation

4.4.2.3.3. Statistical and combined perturbed execution modes

openSF iteration of parameters (Sec. 4.4.2.1) is based on the combination of all possible values that the affected parameters can assume. This approach results in an exponential increase of the executions number, depending on the number of parameters being iterated and the number of values. For example, the iteration of two parameters with 10 different values each one ends up to 100 executions.

The above-mentioned approach can be not the most efficient one in some specific cases, e.g. for statistical modules, for modules that need to run in different modes for each simulation, or for modules that need to be executed more than one time depending on a parameter value. In order to cope with the variable needs of such casuistry, two additional mechanisms have been implemented to handle the iteration/perturbation of parameters or batch runs.
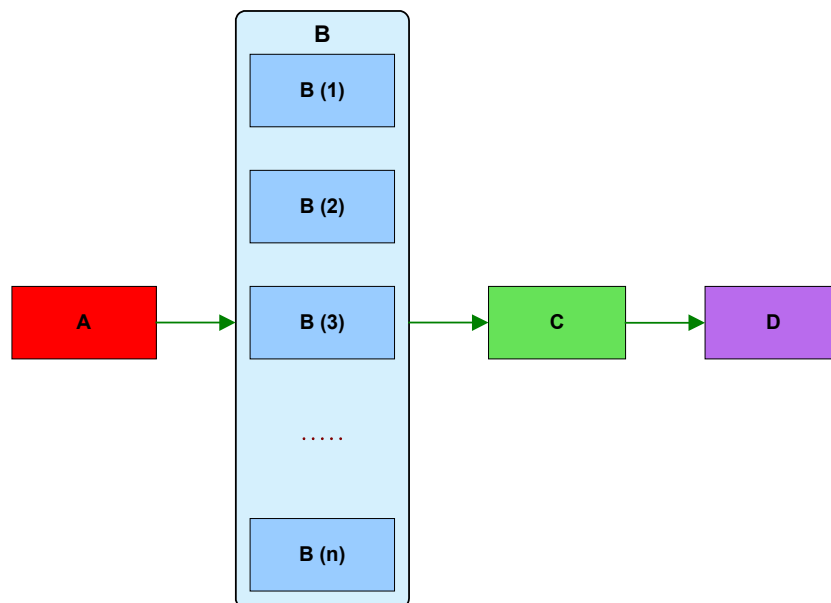
Figure 4-105: Statistical mode execution scheme

Thanks to the first additional mode of execution (Statistical mode), any module can be configured with different parameter perturbations/iterations and, without the necessity to setup any extra simulation configuration, the simulation framework introduces a loop executing *just the selected module "N"* times.

Note that in the "statistical" mode, the output files/folders of each module will be named in principle in the same way at each execution, and consequently **it is the responsibility of the module developer to handle this issue to avoid overwriting the output data**.

The other execution mode (Combined mode) provides the possibility of setting up a number of *simulations* equal to the number of shots configured for the perturbations of each module. This approach imposes the constraint that all modules shall be set-up with the same number of shots, thus every module including the unperturbed ones will be re-run with the same parameters.
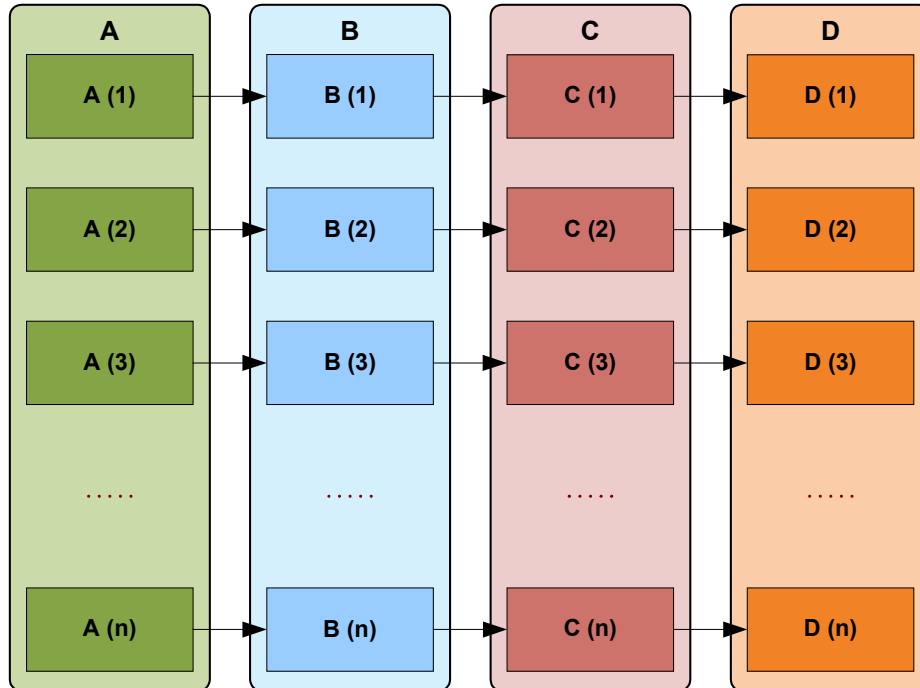
Figure 4-106: Combined mode execution scheme
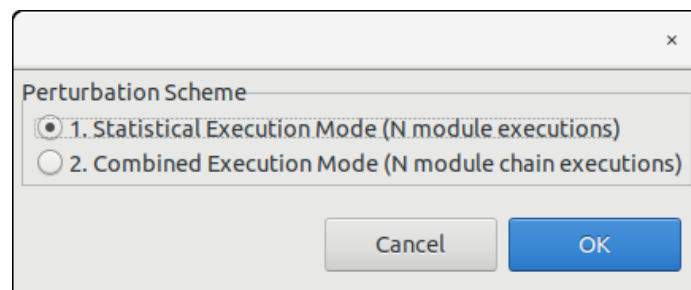
Figure 4-107: Execution mode selector

Some important considerations about this two additional execution modes are the following:

- In Statistical mode, it is modules responsibility to handle output files as openSF specifies always the same filename for each shot.
- In order to make the Statistical mode repeatable by script, openSF creates a set of configuration files, one per shot specified, using as name convention "File_N.xml" where N is the iteration number. This is not needed in Combined mode as configuration files are stored in different simulation folders.

4.4.2.3.4. Perturbations functions

In this section all the available perturbation functions are presented. The independent variable is represented by *t*.

4.4.2.3.4.1. Deterministic functions

Deterministic functions are those whose value is known in all the time domain.

❑ **Affine**

Calculates the perturbation as an affine value. An affine transformation consists in a linear transformation and a translation.

- error = $a_1 + a_0 * t$

❑ **Bias**

Calculates the perturbation as a constant value.

❑ **Linear**

Calculates the perturbation as a linear value:

- $p = a * t$

This is a particular case of affine transformation when translation variable is equals to 0.

❑ **Parabolic**

Calculates the perturbation as a parabolic value.

- $p = a_0 + a_1 * t + a_2 * t^2$

❑ **Polynomial**

Calculates the perturbation as a generic polynomial value. This function has as many float parameters as degrees of the desired polynomial plus one.

❑ **Step**

Calculates the perturbation as step function.

- if $t < T_{step}$ → $p = a_0$
- if $t > T_{step}$ → $p = a_1$

❑ **Sinusoidal**

Calculates the perturbation as sinusoidal function

- $p = a * sin(2 * pi * f * t + phi)$
- f(Hz)
- phi(deg)
- t(secs)

❑ **Tangent**

Calculates the perturbation as tangent function

- $p = a * tan(2 * pi * f * t + phi)$
- f(Hz)
- phi(deg)
- t(secs)

Remember that the tangent function has singularities when the angle evaluated is ± n*pi/2.

4.4.2.3.4.2. Sampling functions

The openSF error generation plugin implements three interpolation methods, linear, polynomial and spline sampling.

In order to define the points of the interpolation there is a common set of variables that are listed below.

❑ **Linear Sampling**

This function makes an interpolation with the given points assuming it follows a linear rule.

❑ **Polynomial Sampling**

This interpolation method builds a polynomial grade n, n being the number of specified points. This interpolation minimizes the Least Square Error. Ref: Neville Method.

❑ **Spline Sampling**

Interpolate the given "n" points with Cubic Splines Method.

How to use the sampling functions

The sampling functions are useful for cases where the perturbation is a function known at discrete instants. That is, F = {yj, xj}, j=1, ...n. In such a case, openSF provides with the functionality of interpolating according different methods: for a given time xt calculate the corresponding perturbation in the discrete series {yj, xj} such that yt = F(xt)

The xj vector assumes equidistant point and as such it is only defined using by

- xMin: Min value of abscise axis
- xMax: Max value of abscise axis
- step: Increment between abscise values

The number of points must be:

$$\frac{xMax - xMin}{step} = nValues$$

The sampling functions configuration has to include *nValues* y values to match the number of x values.

- The "linear sampling" method uses a linear interpolation between points
- The "polynomial sampling" method interpolates using a Neville polynomial
- The "spline sampling" method interpolates with splines

Summarizing, these functions are useful in those cases where the perturbation values for example come from measurements whose underlying module is not fully known or cannot be represented by an analytical equation (a gauss distribution, a beta distribution, a combination of gaussian and linear function and so on).

4.4.2.3.4.3. Non-deterministic functions

Common random function implementation with seed management for testing purposes. If seed is set to zero openSF initializes pseudo-randomly the seed (e.g. used for non-repeatable executions). All the functions are common statistical probability density functions:

❑ **Beta Distribution**
❑ **Gamma Distribution**
❑ **Exponential Distribution**
❑ **Normal Distribution**
❑ **Uniform Distribution**
❑ **Poisson Distribution**
❑ **Truncated Gaussian Distribution**
❑ **Uniform Discrete Distribution**
❑ **Distribution with custom Probability Density Function**

The latter returns the value of a random variable generated with a custom pdf given. It is only recommended to use it by expert developers/scientists.

4.4.2.3.4.4. Binary and composite operations

The simulation perturbation system implements some basic mathematical operations in binary mode. The operations implemented are:

❑ <u>Addition</u>
❑ <u>Subtraction</u>
❑ <u>Multiplication</u>
❑ <u>Division</u>
❑ <u>Exponentiation</u>
❑ <u>Root</u>

4.4.2.4. <u>Time-based scenario orchestration</u>

This section provides details regarding openSF time-based scenario orchestration, which allows the user to launch a series of simulations varying the groups of parameters by using the "simulation time" as the trigger. The modules may have different execution modes that are triggered by the simulation time, and in each module mode the modules parameters can be initialised with different values. An example of a scenario of instrument operational modes is shown in Figure 4-108.

Time

Module A	Module A	Module A
• Mode **On**	• Mode **Off**	• Mode **On**

Module B	Module B	Module B
• Mode **Summer**	• Mode **Autumn**	• Mode **Winter**

Module C	Module C	Module C
• Mode **Nominal**	• Mode **Maneuver**	• Mode **Maneuver**

Simulation 1   **Simulation 2**   **Simulation 3**
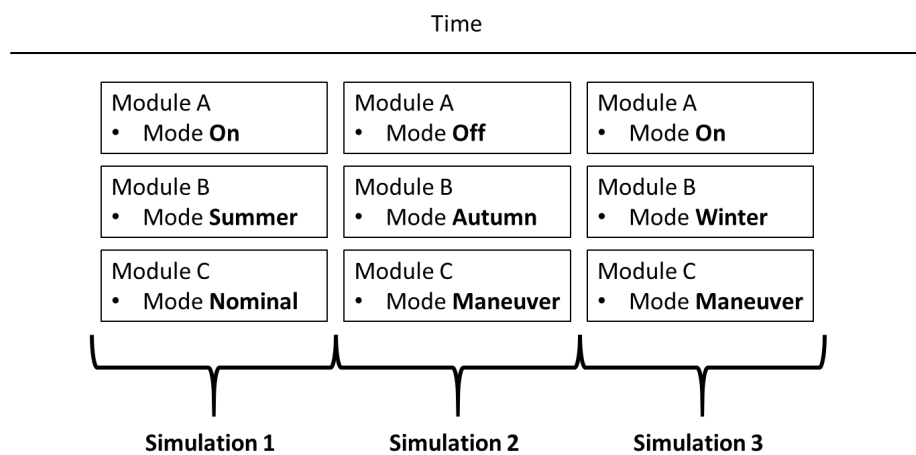
Figure 4-108: Example instrument operational mode scenario

The different values of the parameters associated to a given module mode are stored in the module configuration files. Every module can have multiple execution modes, and for each mode not only the standard parameters values can be customised, but also additional parameters can be added.

Once the configuration is established, any changes made to the module configuration files may show an error message. For further details, please refer to section 4.2.2.7.

The simulations are organized in a time sequence that represents the mission sequence. The time sequence does not admit gaps or overlaps.

Every simulation has its starting time and duration, which must correspond to the mission time segment that the simulation is simulating. For every time segment, for every module that has execution modes configured, a mode can be selected.

The default initial epoch and duration of a time-based scenario are stored in the openSF preferences. Once a new timeline is created, the initial epoch is stored in the timeline file. When a timeline is created, the initial epoch is used to set the starting time of the first time-segment (however this can be edited). When a new time segment is created, its initial time is set to the initial time of the previous time segment plus its duration. The default duration of a time segment is taken from the value set in the openSF preferences. The starting time of each segment is stored in the timeline file.

When a simulation is launched, the initial epoch is copied from the timeline file to the global configuration file. The initial time of every time segment is copied from the timeline file to every module (that has execution modes) configuration file of every time segment.

Note that the initial epoch stored in the global configuration file might hence not correspond to the initial time of the first time-segment.

The modules within a simulation can be set in Processing and Simulation mode. Modules in Processing mode are bypassed when a simulation is run in Time-driven mode.

Note that the time segments can be possibly run in parallel, so the order of execution does not necessarily follow the timeline sequence.

For more details on the concepts and definitions supporting the time-based orchestration, please refer to [AD-E2E].

4.4.2.4.1. Time-based orchestration interface

In a simulation editing window, each specific module time mode is rendered as a node in the tree of parameters – see Figure 4-109. This is only for display purposes, i.e. when rendering the parameters in the simulation editing window parameters tree, so it is not reflected in the repository.

Figure 4-109: Module parameters folder organization on a per-mode basis

In the execution tab (Figure 4-110), classification of each module in the processing chain according to Simulation/Processing categories is done (using the context menu) through the Execution pane in the simulation control window.
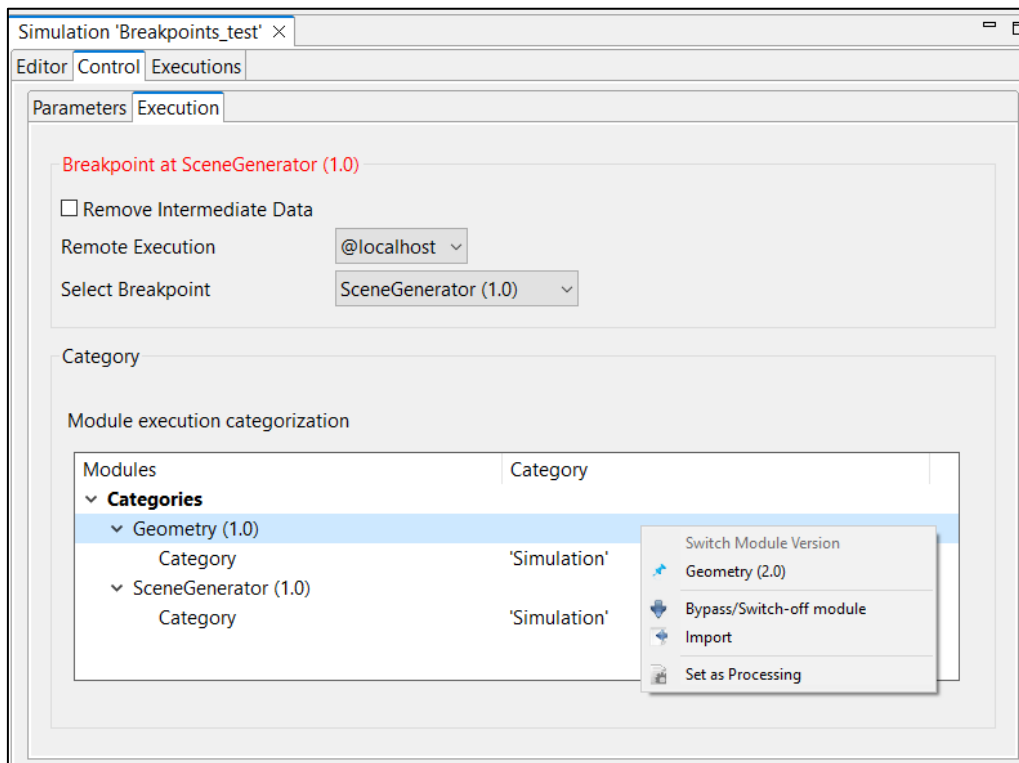
Figure 4-110: Module categorization by Mode

The interface for editing the timeline definition is available upon selecting the 'Timeline' option in the "Parameters" tab – see Figure 4-111. This panel allows to define and enable the global timeline parameters and the actual list of time segments to be executed.
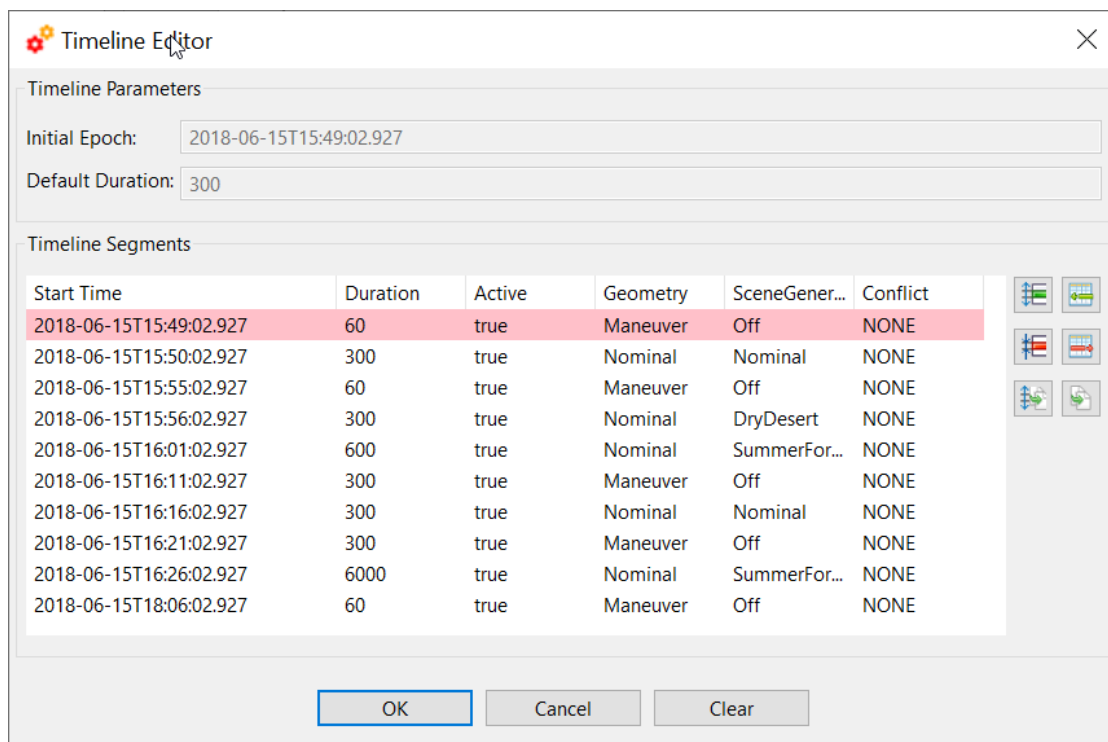
Figure 4-111: Timeline management view

![INDRA logo]

openSF

OPENSF-DMS-TEC-SUM01

4.9

System User Manual

18/06/2025

The global timeline parameters are: (a) the Initial Epoch and (b) the default time segment duration. Timeline segments are displayed below as rows in a table where each column corresponds to the time segments attributes.

It should be noted that the initial epoch parameter is stored in the global configuration file, so this file must have been defined prior to the simulation's Timeline definition. If no previous timeline parameters are present on the global configuration file, openSF will initialize them to the default values.

To define a Time-Driven simulation, the user can load a previously configured timeline scenario through the *Import* button in the simulation editor, or manually configure one. The timeline scenario file must be compliant with the structure described in [AD-E2E], and the modes referred by the scenario file must match the ones specified in the local configuration files of the relevant modules. The user can define as many timeline segments as desired, saving the modifications when satisfied with the result.

For editing the timeline segments the interface has a set of buttons to add, remove or duplicate a time segment row. These three actions can be performed either accommodating the existing segments (e.g. add-shift) or without affecting the existing segments. Accommodation is accomplished by adjusting the start times of the segments so that there are neither time segment overlaps nor gaps in the timeline. The buttons have been arranged in a rectangle, where the left column buttons shift the time segments, while the right ones don't.

The addition of each time segment uses default values from the global timeline parameters defined above. Within the timeline table on each row, it is possible to select the mode (active true or false) for each module to be executed during each time segment.

The default representation of time segments in the timeline table can be configured via the *System preferences* → *Application Setting* → *Timeline visual*. This allows using: *duration*, *number of steps* or *end epoch* as time segment identifier (first column).
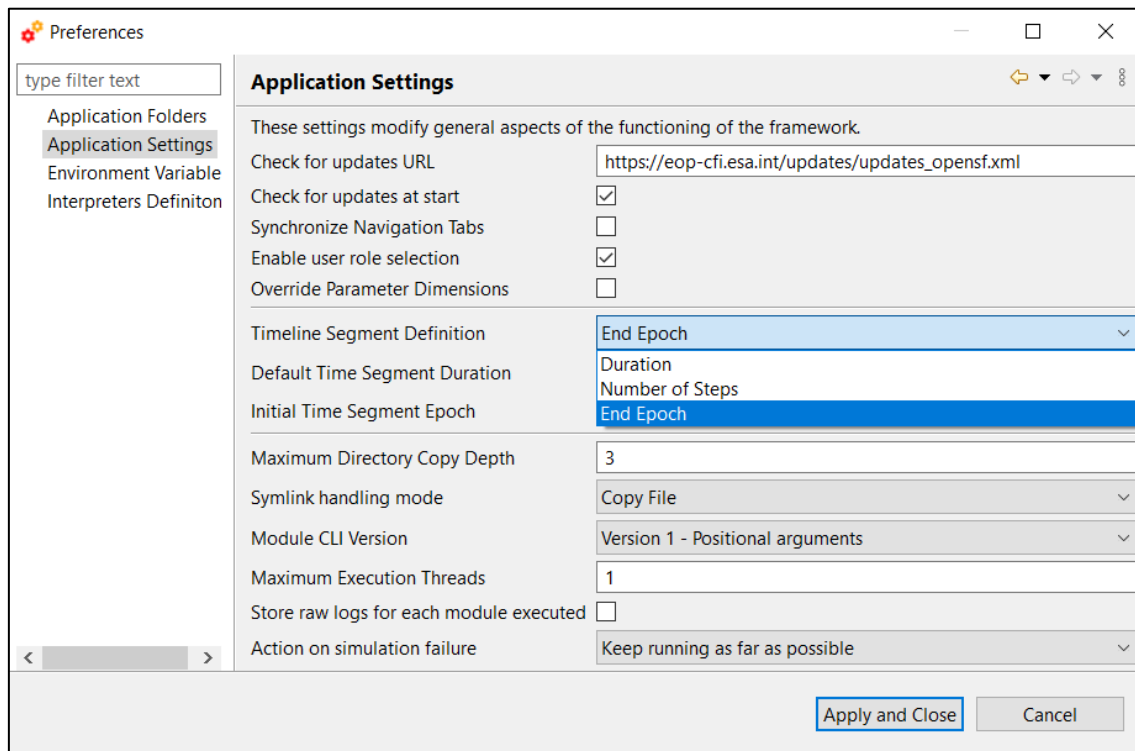
Figure 4-112: Timeline preferences

Each timeline segment time definition shall then be introduced by the user with (a) a start time, and (b) one of the following: duration, number of steps or end epoch. If then the user inputs duration, the system can compute the other two alternative values from the start time (and so on for any other selection). Then in the timeline configuration file all the four time-related values will be written for each segment. Therefore, when the user switches preference in the global definitions it's simply changing the "view" over the time data. Being a global configuration, it actually represents the specific user preference for viewing one of the three alternative values. If the simulation is changed or if a given timeline configuration file is supplied (which shall be self-contained) to another user then the displayed column shall be the one corresponding to the users' preference.

The user can press on the "Clear" button to remove all segments. Saving the dialog then will remove the timeline configuration from the simulation.

4.4.2.5. *Monte Carlo simulations*

This section provides guidelines to implement a Monte Carlo (MC) simulation in openSF, taking into account current openSF limitations.

Full support for MC studies is intended in the future, as the current implementation is known to have limitations. With the current status of openSF, MC analysis can be simplified by taking some precautions when developing simulation modules.

Three different approaches are suggested to implement a MC analysis. The best approach depends on the specific constraints of the project and the implementation of the module on which it is desired to perform the MC analysis. These different approaches originate from the fact that, although openSF can perform iterations (parametric analyses) both in local and global parameters, it can only introduce perturbations (Monte Carlo analyses) in the LCFs.

4.4.2.5.1. *One module MC with local parameter*

The first option to implement a MC analysis in openSF considers that only one module needs to be executed multiple times. The parameter to be perturbed only needs to be injected in a single MC module. The layout of this approach is shown in Figure 4-113.

Figure 4-113: Monte Carlo chain in statistical mode

In the most generic version of this approach, the module on which the Monte Carlo will be performed is preceded by a chain of modules and generates results that will feed into another set of modules. These post-processing modules can be used to merge the multiple results of the MC.

The parameter(s) on which the MC is applied must be defined as part of the module's local configuration, with the perturbation specified in openSF through the "Perturbation" menu – see details on how to configure the desired number of shots and use the **Statistical** execution mode at section 4.4.2.3.

In the chain shown in Figure 4-113, the modules A through B are only executed once, and the same happens with the modules D through E.

Notice that openSF only initiates the execution of a module when all expected inputs are available. Therefore, to build a sequential execution the chain needs to define the inputs of a module as the outputs of the previous one.

To use this approach, the developer must consider the following points. Alternative approaches should be considered in case the following points don't apply to the problem at hand.

- ❑ The value of the varying parameter can only be defined in the local configuration file of the MC module.

- ❑ openSF passes the same input and output files/folders to all the executions of the MC module. This implies that the MC module knows the varying parameter and dynamically modifies the name of output files to avoid overwriting them.

- ❑ To correctly process the results of the MC module, downstream modules must be able to read and understand those results. Some degree of agreement is necessary between the MC module and the

downstream modules, either by sharing the naming convention used to generate output files and recognise from the names the varying parameter, or by reading the result files and obtaining that information from its contents.

❑ Considering that openSF initiates the execution of a module when all inputs are available, it is not possible to simply specify a folder as output of the MC module to pass results to downstream modules. After the first MC module terminates, the presence of the folder indicated to openSF that the following module can be started. Possible alternative solutions are listed below, knowing that the most applicable depends on the design of the simulator.

   o Pass the number of shots to the MC module, and at the end of execution generate a "MC_Completed" flag file in case the expected number of output files is available (i.e. all shots have finished). The "MC_Completed" flag file is essentially used to trigger the execution of downstream modules.

   o Pass the number of shots to the first post-processing module. This module is launched as soon as openSF verifies that a results folder exists, and enters an active polling loop until it finds the expected number of files is available.

   o In case the modules design/implementation cannot accommodate the above approaches, divide the whole Monte Carlo study in two completely independent simulations. The first simulation executes the modules up to the MC modules, without any post-processing modules; then the second, executes the remaining post-processing modules, manually configured and launched by the user after the completion of the first one.

The options discussed above do not consider handling of errors in MC modules. In those cases, either the modules need to implement error-checking mechanisms, or the user needs to check the correct execution of all shots to ensure reliable results.

4.4.2.5.2. Multiple modules MC with local parameter(s)

Another approach has to be used when more than one module is executed with perturbed parameters for each of the Monte Carlo shots. The chain presented in Figure 4-114 shows two main disadvantages: all the modules are executed for each of the MC shots, with the consequent increase in the computation time, and it is not possible to combine all results generated by the MC modules.
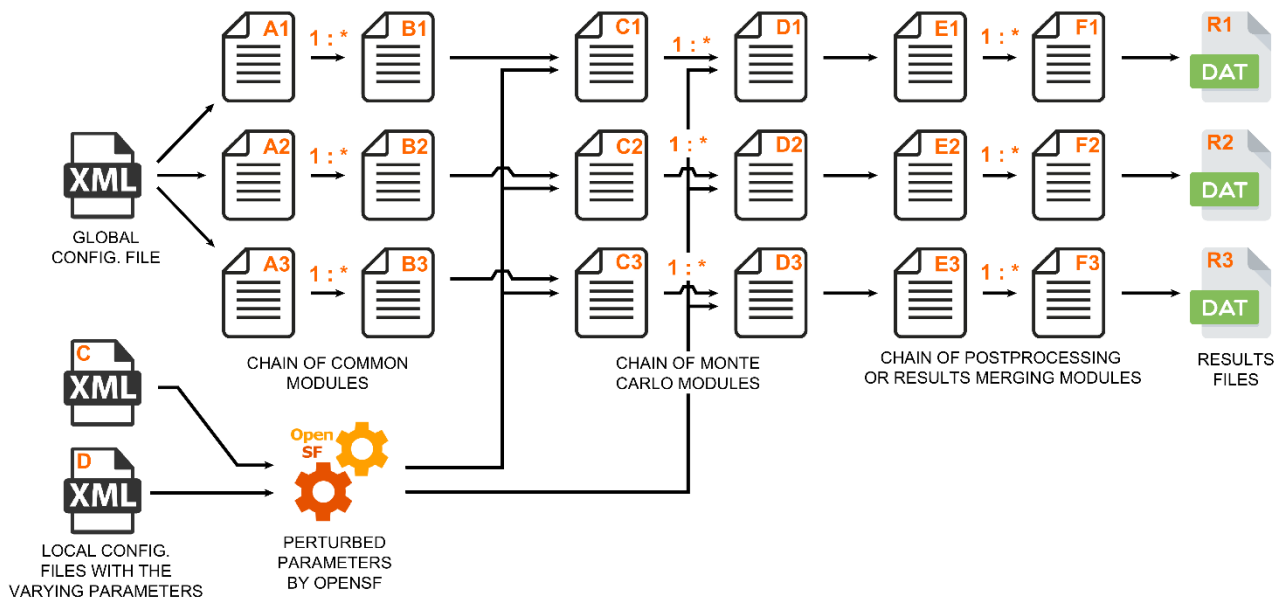
Figure 4-114: Monte Carlo chain in combined mode

In the most generic version of this approach, a chain of modules on which the Monte Carlo will be performed is preceded and followed by a set of normal modules.

As in the previous approach, the parameter(s) affected by the MC must be defined in the local configuration file, with the perturbation specified i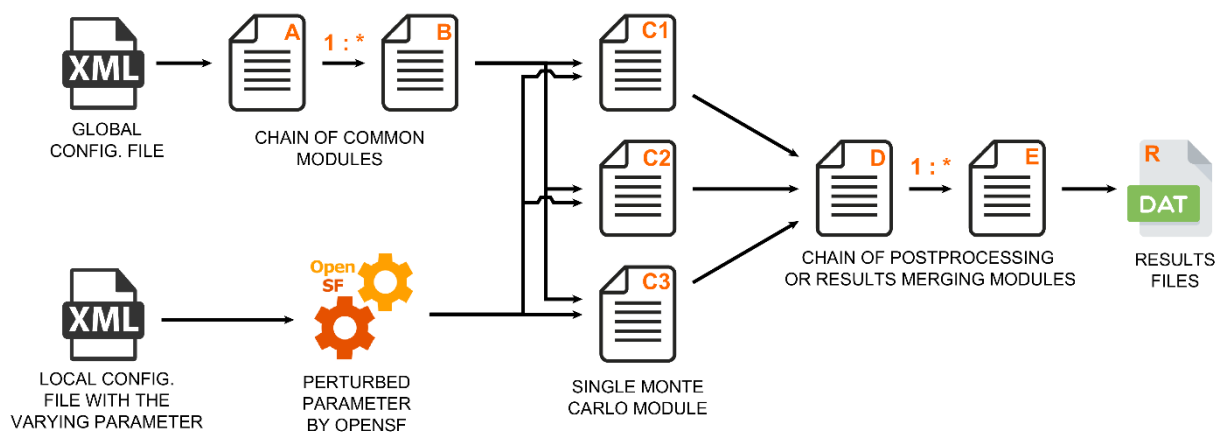n openSF through the "Perturbation" menu – see details in the section 4.4.2.3 on how to configure the number of shots and use the **Combined** execution mode.

This approach relies in assembling an independent simulation for each perturbation shot. Each shot is considered independently, rerunning common modules, and storing the results in a different simulation folder. In this case, the MC module is not obliged to provide a different name for the result files, although it might be desired in order to gather all the results in a common folder.

The use of the same value of a perturbed parameter in more than one module is supported by simply defining the desired parameter in the local configuration files of the affected modules, and in openSF define the exact same perturbation for all of those parameters.

To keep the simulations executions coherent, all parameter perturbations of the MC modules need to have the same number of shots. The non-perturbed modules will be re-run with the same configuration parameters and the inputs produced by the previous modules of its simulation.

4.4.2.5.3. Multiple modules MC with global parameter

In case none of the previous approaches is applicable, or the parameter to vary is not local but global, there is still one additional approach that can be used to implement a Monte Carlo analysis in openSF.

As this approach relies on an external tool to generate the perturbed parameter values, it should only be used when the varying parameter needs to be injected into multiple modules and, either the varying parameter must be defined in the global configuration file, or else no perturbation capabilities are needed.

The typical chain for this approach is shown in Figure 4-115. To setup this approach, create a normal simulation in openSF without taking into account MC. Use the "Iteration" features, as described in section 4.4.2.1, to customize the global parameter to be perturbed. The perturbed parameter values can be introduced manually, or by providing a file containing the values – this file is typically generated by an external tool. Consider also using a "batch simulation configurator file" as explained in section 4.4.2.2. Executing the simulation essentially results in a parallel execution for each of the values of the varying parameter.
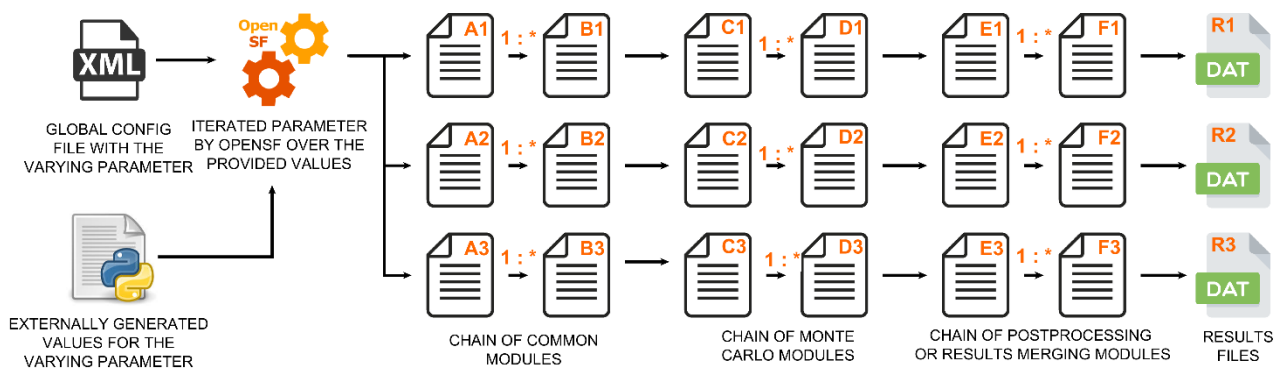
Figure 4-115: MC with a global parameter

As the varying parameter is global, all the modules can access it and decide according to its value. As with the previous approach, each shot of the MC is executed as an independent simulation implying that all the common modules are executed repeatedly and that there is supported way to gather/aggregate all the results.

This last limitation could be bypassed by implementing post-processing modules considering the following:

1. Define the total number of shots of the MC in the global configuration file.
2. Considering that all the simulations files are stored as subfolders within the main simulation folder, at the beginning of the module execution, retrieve the simulation folder path and go one level up.
3. Index all the existing folders and their contents.
4. Count the number of MC results and if it is lower than the expected number of shots, end the execution of the post-processing module.

5. When the number of existing results is the same as the expected shots (i.e. when executing the last simulation) perform the post-processing over all the indexed results.

The choice between using this or a similar solution or to simply defining another processing simulation to aggregate the results and manually execute it depends on design decisions particular to each project.

4.4.2.6. Changes in the configuration files after setting up multi-execution

In a simulation where multi-execution configurations have been established, changes made to the configuration files may produce changes in them, which are made visible to the user via a series of warnings. For example, if a perturbation configuration references a parameter, but that parameter no longer exists in the module LCF, the perturbation for it is removed from the configuration.

Changes to configuration files can happen in two ways. The first is manually changing their paths, either from the Definition tab or Configuration tab and select the configuration files to be used in the modules or as the global configuration file. When a file path is changed, its parameters are reloaded.

The second way may happen either automatically when opening or running a simulation, or through the button to reload the configuration files from the file system placed in Configuration tab. In that case, all parameters are reloaded from all files.

It is important to note that this affects not only the current active simulation mode, but all configurations present in the simulation. The iteration, batch and perturbation configurations are mainly sensitive to parameters being deleted from files, with messages similar to Figure 4-116.
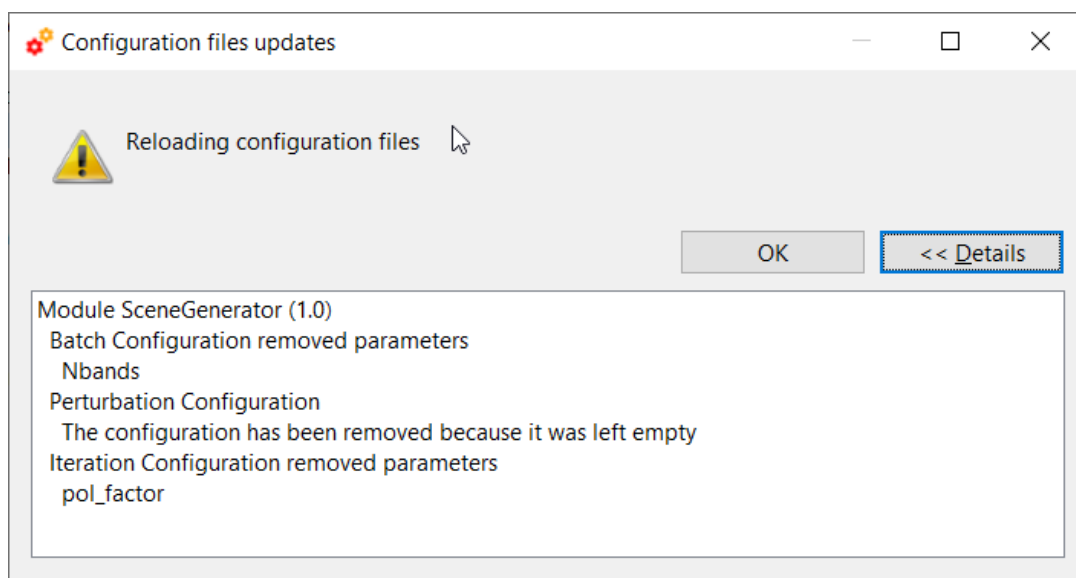
Figure 4-116: Warning due to changes in the configuration files

In a similar way, the Timeline mode has it owns warnings. If a module has defined some executions modes in the timeline and all the parameters in one of them, are deleted a warning as the one in Figure 4-117 is displayed. The user must edit the configuration, select appropriate modes for the missing timeline segments before proceeding. Figure 4-118 illustrates the missing timeline modes.
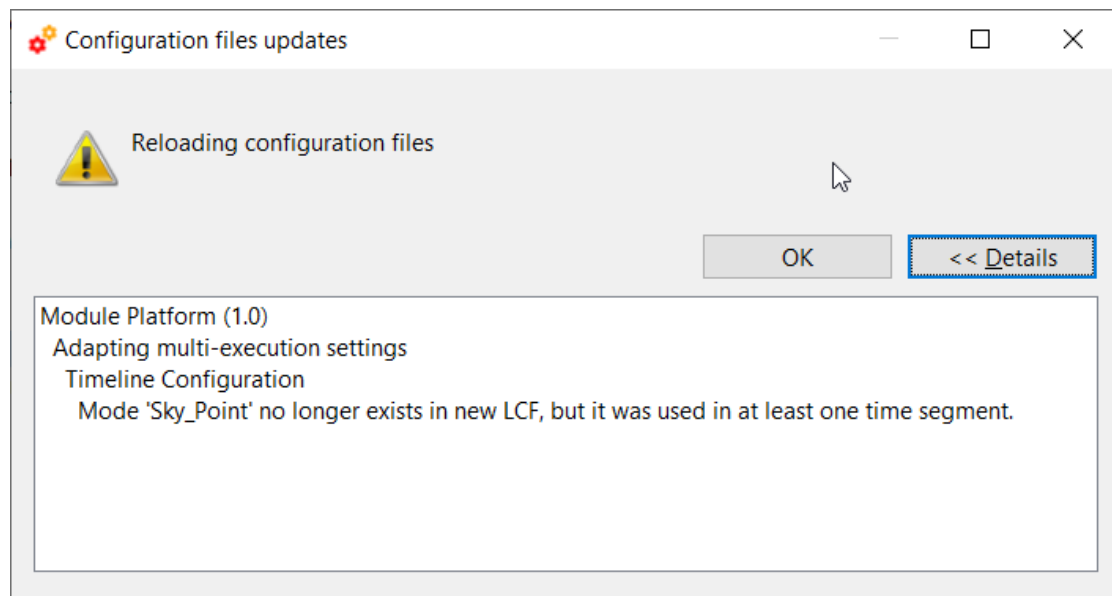
Figure 4-117: Warning due to missing parameters in the timeline configuration files
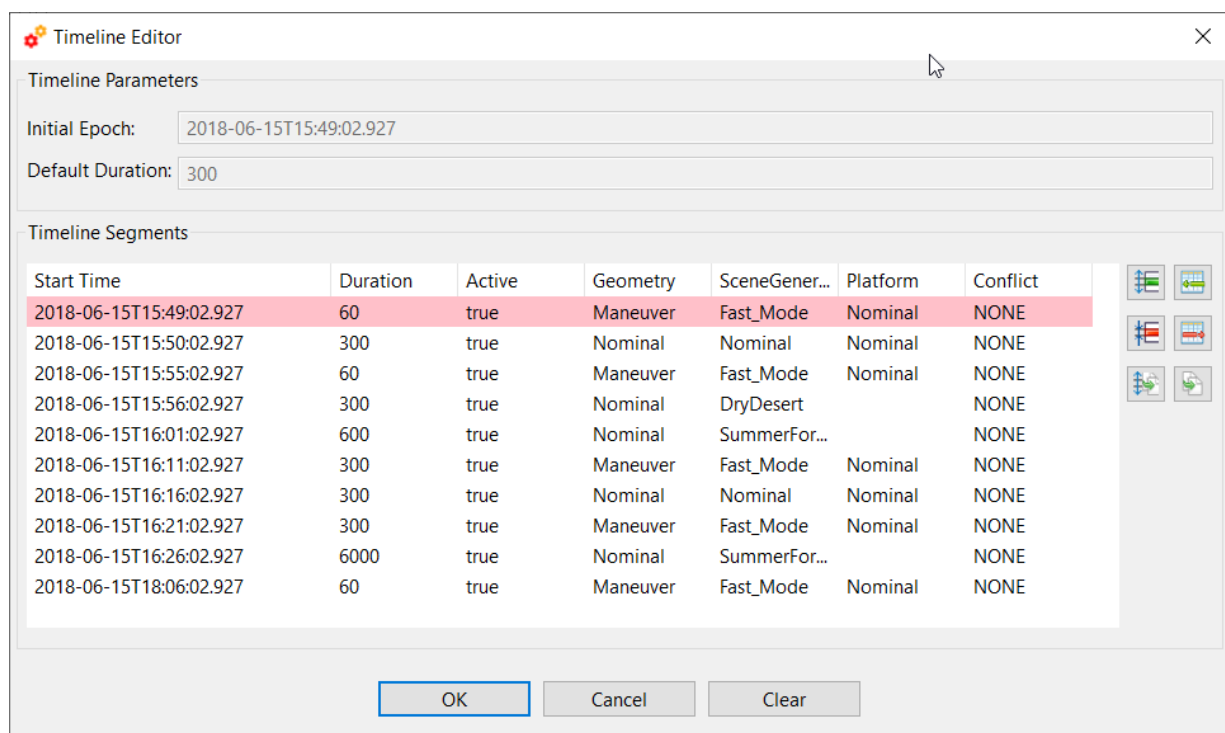
Figure 4-118: Missing modes for a module

The same happens when changes in a module LCF *add* execution modes to a module that did not have any before. This is reflected in a message similar to Figure 4-119 and the user must configure every mode for each time segment, otherwise the timeline and simulation cannot be saved.
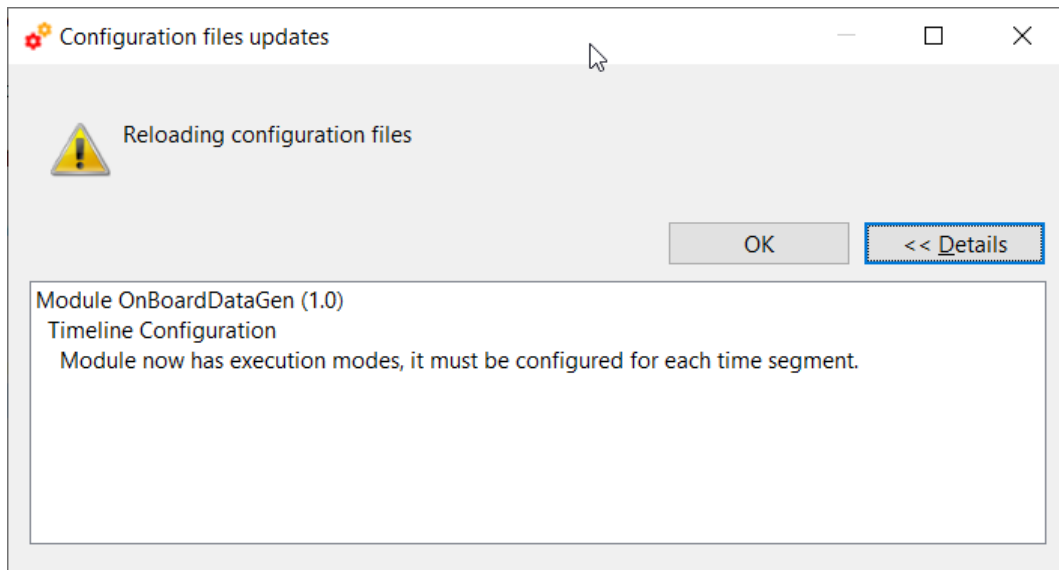
Figure 4-119: New executions modes for a module warning
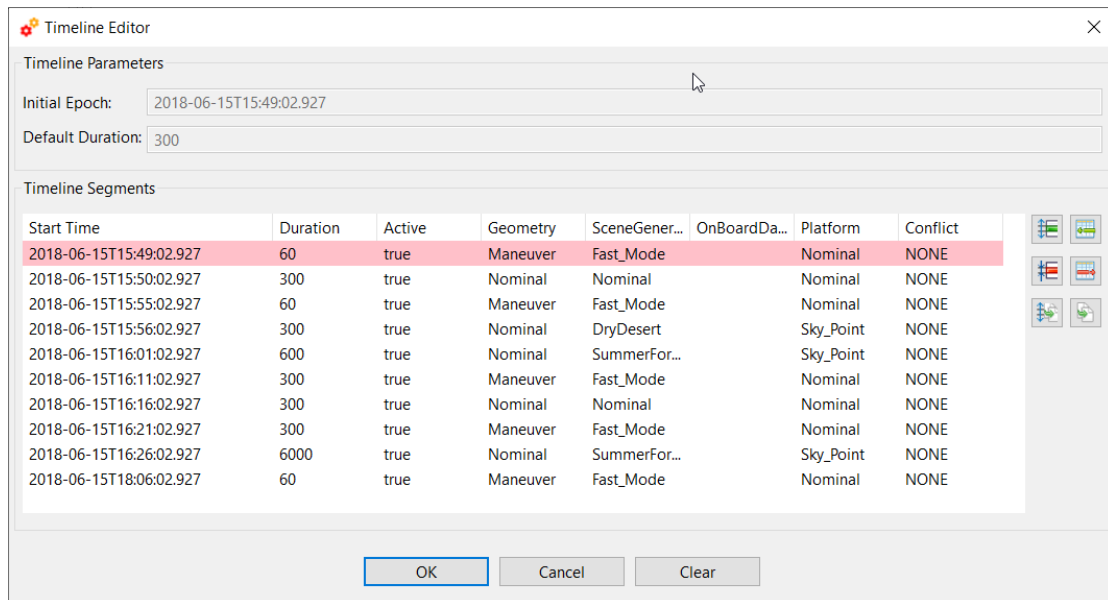
Figure 4-120: Configuring empty execution modes

4.4.3. Launching a simulation

| M | I | A | E |

Once a simulation definition is ready, users can execute it.

The effective execution order of the simulation modules is determined by openSF, based on the input/output dependencies – see section 4.4.3.1.1 for further details.

Upon the activation of the "run" command, the system performs a series of checks to ensure the validity of the simulation:

- If the global configuration file for the simulation, or the local configuration file of any active module in the simulation is missing, the execution will not proceed, and a dialog will be displayed listing the missing files as shown in Figure 4-121.
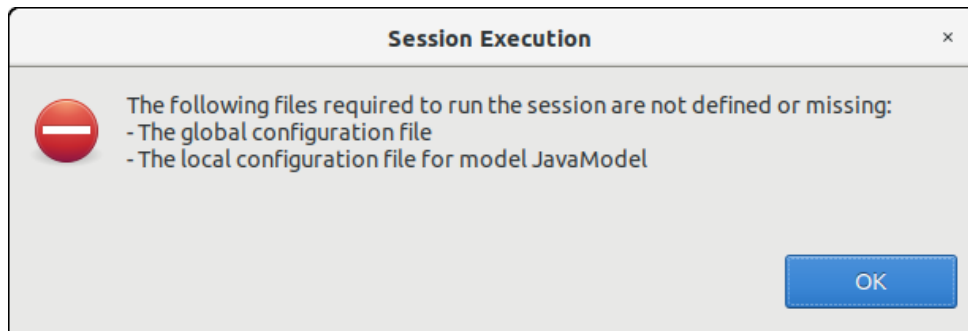
Figure 4-121: Execution prevented due to missing configuration files

- If there is any other file with the "missing" status (that is, the system is unable to find it in the given location), underlined openSF makes the assumption that this file will be in the right place when needed, so it leaves the responsibility of placing it in the correct place to the user or to process outside the system. A fatal error is likely to be raised by a module if it cannot locate a needed file.

- If any parameter is not valid, openSF will show a message warning the user. The simulation can be run with parameters in a not valid state, but modules may then raise errors and stop the execution if they cannot parse the value, if a parameter points to a non-existing file, etc.

- If any parameter has been added or deleted from the configuration files of any active module involved in the simulation, the execution will not be carried out, and a dialog will be displayed listing the configuration issues as shown in Figure 4-122. See section 4.4.2.6 for details.
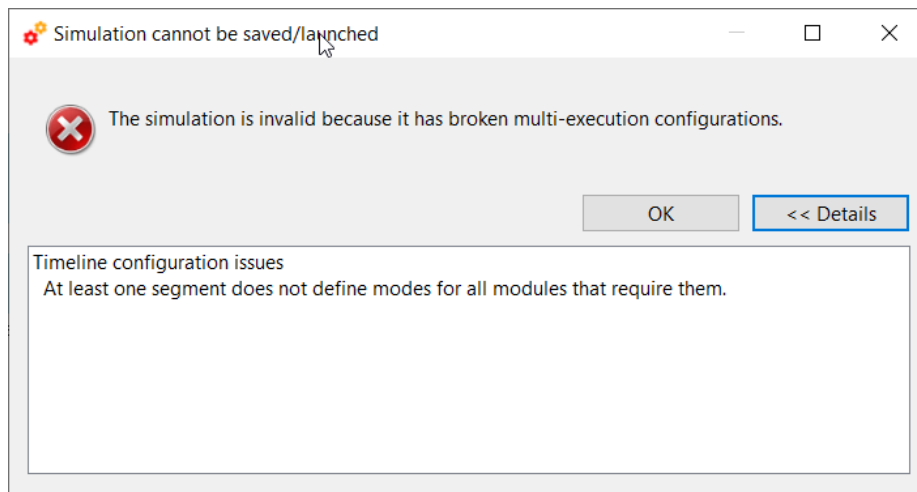
Figure 4-122: Execution prevented due to configuration issues

Once every validity check is fulfilled, openSF will run the simulation (Figure 4-123) and information about its status will be given to the user in three different ways. The first is the simulation progress section, where the overall simulation progress is displayed by a progress bar. The second is the modules progress tab, where each module is coloured according to its state and a small progress bar displays the progress of the single module. The third is the logs tab, which displays the logs generated by the system and the modules.

The Executions View provides a structured and intuitive interface for managing and monitoring simulations, whether single or multi-execution scenarios. The design focuses on organizing executions in a hierarchical manner on the left-hand side while maintaining a detailed view of individual execution progress on the right-hand side.

On the left side, the executions are displayed in a tree structure, offering a clear and organized representation of all simulations. When a single simulation is run, it appears as a top-level row in the tree. For cases where a multi-execution is launched, the multi-execution is added as a top-level row that can be expanded to reveal all its sub-executions as second-level rows. Expanding the row reveals all sub-executions, displayed as second-level rows. These sub-executions provide detailed statuses for each part of the multi-execution process. Clicking on any row selects the corresponding execution.

The right panel provides detailed information about the selected simulation(s) and adapts dynamically to the user's selection. When the user selects a single execution or sub-execution from the tree, the panel displays

detailed progress and results for that specific execution, resembling the current panel tab. The display includes essential information such as simulation progress, module status, and log messages, allowing users to analyze the execution comprehensively. If no rows or multiple rows are selected in the table, the panel does not display detailed execution information. Instead, the contents remain "empty" but visible, signaling to the user that no specific execution is currently selected and that needs to select a specific row to view details.

When a module execution ends in failure, the simulation is immediately displayed as "failed," but already launched modules continue to run to completion. Depending on the application settings, additional modules may be started too, see section 4.5.2. In particular, the module itself is immediately turned red (failed) as soon as a sub-module fails (see Figure 4-124).

In the modules progress tab, the colour scheme of the modules is the following:

- ❑ Green: Completed successfully
- ❑ Yellow: In progress
- ❑ Blue: Pending
- ❑ Red: Failed

The single files (represented by the little squares next to the modules) are also coloured from black to green as the modules produce them.

During the simulation execution, all events raised by the modules are collected and displayed in the Log Messages table. By default, these messages are processed and displayed/coloured based on their type, but openSF allows to inspect the original messages by checking the "Show non-formatted messages" option. In this table, if the user brings the scroll all the way to the bottom, the table will keep auto-scrolling. The event messages displayed in the Log Messages table are collected and stored in a log file that can be opened for further inspection using the "Show Log" button. The collected events can be of one of the following types:

- *System information* – An event with some information to the user is generated by the platform. This is a harmless event; the execution continues with no interruption. Coloured in dark grey

- *Information* – Some module raises an event. Its message is intercepted and stored by the platform. This is a harmless event; thus, the execution continues with no interruption. Coloured in green

- *Warning* – A module has detected a non-fatal error or situation. From the point of view of openSF, this is a harmless event, so the execution continues with no interruption. Coloured in yellow

- *Debug* – These events are raised when executing the simulation in "debug mode". Some modules optionally use an environment variable to show debugging information (see Sec. 4.5.1). Coloured in grey.

- *Error* – A fatal error has happened in the module execution, so the entire simulation is considered to have failed. There are multiple ways this may happen: if the module execution unexpectedly crashes, if the module itself informs the platform by using an "Error" type event, or if it returns a non-zero code (for the languages that support it). Coloured in red.

- *Exception* – This log type shows the error output stream of a module when the execution crashes. Typically, this kind of messages is produced when an un-controlled exception has occurred (Ex: error in bash script syntax). Coloured in orange.
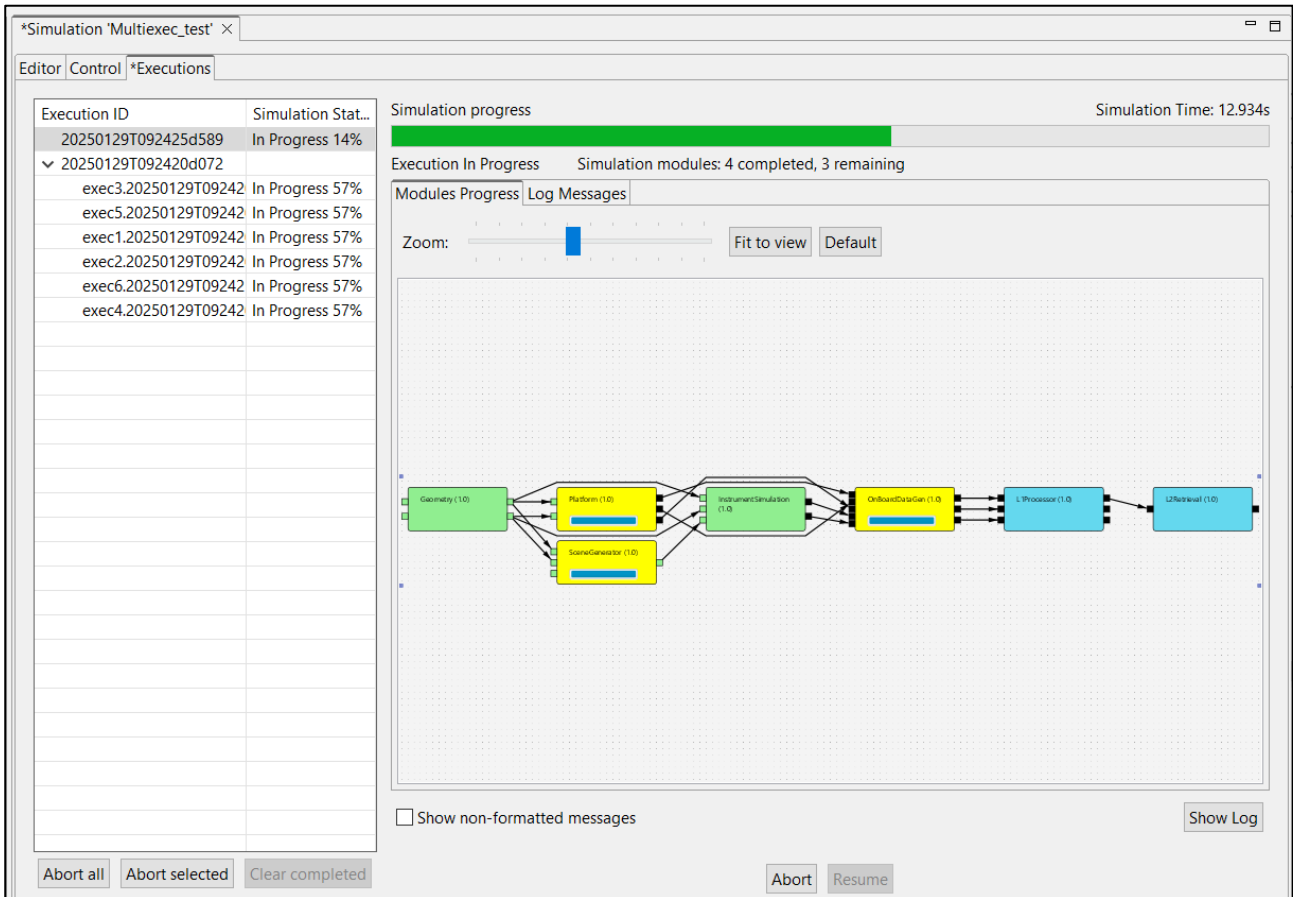
Figure 4-123: Simulation execution progress
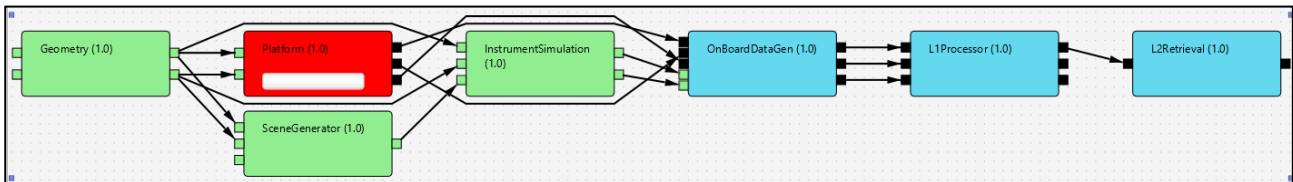
Figure 4-124: Representation of a failed simulation

Figure 4-125: Execution log showing an error message

In the previous screenshot it is also shown how openSF warns the user about output files that have not been created by the module: "*Output file/folder "…" has not been successfully generated by the module*", there is a log message for each output item not generated.

Pressing the "abort" button will make the system ask for confirmation. Once granted, the execution will be interrupted with an error event generated by the system. Later on, this simulation execution can be restarted or recovered from the last valid module executed.

In addition to the above, if there are simulations running and the user attempts to close the simulation view for a running simulation, a dialog like the one shown in Figure 4-126 will appear. This message warns that any running simulations will be aborted if the view is closed.
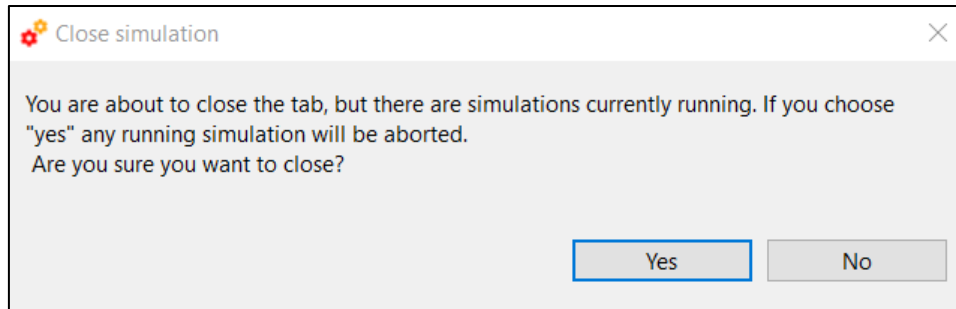
Figure 4-126: Confirmation dialog for closing a simulation in progress.

If there are unsaved changes in the Editor tab in addition to simulations running, a more detailed dialog will appear (Figure 4-127). This message warns that if you close the tab, both the unsaved changes and the running simulations will be affected.
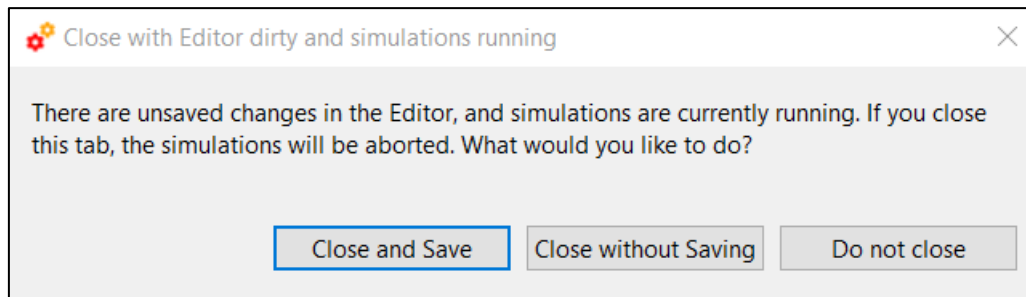
Figure 4-127: Confirmation dialog for closing a simulation in progress and unsaved changes

4.4.3.1. *Parallelisation of module execution*

| M | I | E |

Parallel execution of modules in openSF is based on Multicore programming for parallel computing.

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 parallelised to run on multiple cores simultaneously (Amdahl's law).

openSF target system is typically a workstation or server. These systems will often have multicore processors.

Regarding the parallel execution the approach is based on performing parallelisation at module level: each module acquires a core resource thread and uses it, then releases it when finished. This approach is also generic enough to cover parallelisation at module chain level as well.

Parallelisation is implemented in openSF according to the principle that a module starts its execution as soon as its inputs are available. At the E2E simulation start-up, the only modules that can be launched in parallel are those having external inputs (i.e. inputs not generated as outputs by other modules), while during the simulation execution the system allocates each available core to the first module in the pipeline ready to be executed (i.e. the first in the queue with available inputs).

4.4.3.1.1. Parallel execution

The parallel execution of simulation modules is activated and controlled by openSF based on the configured maximum number of distinct processes used for execution (see option Maximum Execution Threads in the preferences, section 4.5).

The process of scheduling the simulation modules execution can be decomposed in two distinct steps:

1. Taking into account the input/output dependencies between all simulation modules (as specified via the descriptors), openSF calculates a serializable execution schedule of all modules

2. Then, at the start of the execution and whenever a simulation module terminates, openSF launches in parallel as many modules as possible, considering that:

    a. a module can only be launched if all modules providing its inputs have already completed their execution

    b. the number of running processes must always be less or equal to the maximum number of processes allowed

It is important to notice that currently openSF relies only on process termination and the return code to determine that a simulation module has completed, without any other additional check – i.e. openSF detects if the intended outputs have not been generated but it will not consider that cause for termination nor will it verify the validity of generated outputs.

The parallelisation of the simulation modules is essentially transparent to the user, i.e., the parallelisation is performed without requiring feedback from the user, both at E2E simulation chain level and module level. Figure 4-128 shows an example of a simulation execution where two modules are executed in parallel as can be seen by log messages originating from the two modules interleaved with each other.

Figure 4-128: Simulation execution showing parallel module execution

In case two modules are executing in parallel, the log messages are shown in the order of arrival to openSF. They will appear mixed in the simulation log. By looking at the Source column the user can identify which module produced each message. Nevertheless, the user can access the "*Execution/Logs*" option and select/filter whichever log messages according to given criteria (see Sec. 4.4.3.3). Notice that the writing accesses to the log file itself is protected for concurrency issues.

The openSF mechanism on whether to parallelise module execution is based on the simulation module's IO descriptors dependencies. Only modules without such dependencies are considered for parallel execution. This means that during the simulation execution the consistency of the data flow constituents is granted.

In case parallelisation is active for a simulation execution with parameter perturbation a choice is given to the user whether parameter perturbation can be parallelisable or if it should be serialized.
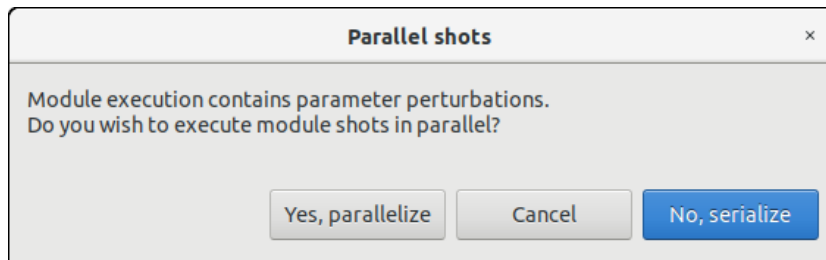
Figure 4-129: Parallelization option dialogue

4.4.3.2. Simulation Resuming

Users can also resume aborted or failed simulations taking into account that the simulation chain will start in the last successfully run module.

4.4.3.3. Logs

One of the products of a simulation execution are a set of events are produced as messages and stored by the system.

Log messages are stored in the file <simulations_folder>/<sim_id>/openSF.log for global events, and in a file named log/simulation.log under each execution folder for events related with a specific simulation. Also, if a certain application setting is enabled (see section 4.5.2), log files are stored for each module in a simulation, in the same folder and named after the related module.

Users can access to the complete set of logs stored by the system in the "Logs" menu (Figure 4-130) from the main menu.
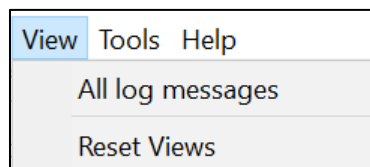
Figure 4-130: Logs menu

Figure 4-131 shows a window with a list of log messages stored by the system. As it can be seen, the table shows the computer date and time when the platform intercepted the event, the type of the event, a message describing the event, the identifier of the simulation associated with the event and its detailed source (module, simulation or system).

Indra Deimos

IND-CMS-SUPTR09-SUM-10-E
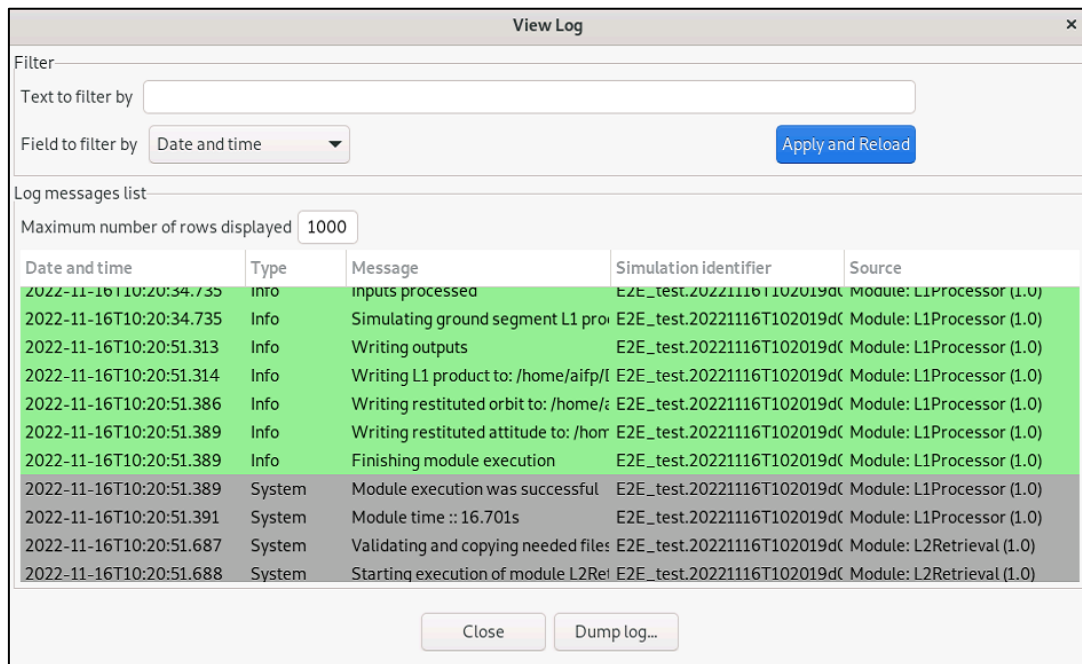
Figure 4-131: Logs list view

This list of events is sorted (by default) in increasing time order until filling the "Maximum number of rows displayed" field. Users can change the number of log messages to be displayed. For example, if the "Maximum number of rows displayed" is set to 10, the list displays the last 10 messages

Users can also filter this list. Users can select a field, input a string that must contain this field and press the "filter" button. A search is performed by the system and the results are shown on screen. Only records fulfilling the filter restriction will be shown. When clearing the filter text, the system shows again the full set of log messages.

Messages can be copied by selecting one or more of them and using the context menu that appears upon right-clicking on one of the selected messages. Moreover, users can access the "dump log" functionality at the bottom of the window. Once selected, users can select the name and location of the log destination file. The list of logs shown in the window will be stored in the file system.

4.4.3.4. *Simulation groups*

A   E

When simulations run in Time-driven or Iteration/Perturbation mode, a considerable number of children simulations can be opened at the same time (see Sec. 4.4.2). To manage this issue, openSF groups all the child sub-simulations into a parent window, as well as in a parent output folder (see Sec. 4.2.3) to ease the user management of the overall simulation results.

Thus, if a simulation is executed in any of these modes, the user will be prompted with a screen collecting all the sub-simulations and each of the execution results, as depicted in Figure 4-132 (for a time-based execution) and in Figure 4-133 (for an execution with iterated or perturbed parameters). If the user closes such a tab, any unfinished simulations will be aborted.
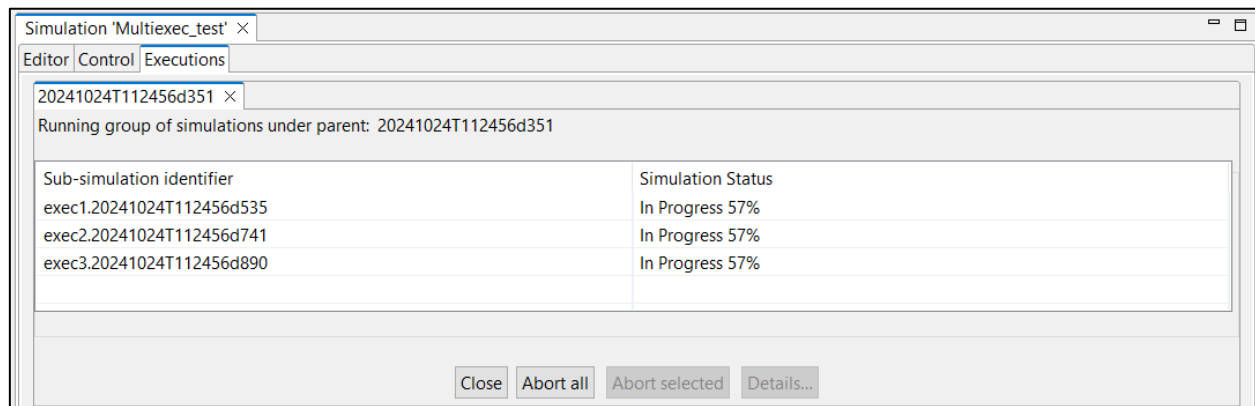
Figure 4-132: Grouping of simulations for the Time-Driven execution

While the simulations are executing, the user has several available options: "Abort all…" which aborts all the sub-simulations, "Abort selected" which aborts only the selected Sub-simulation or "Details…" if the user wants to open one subs-simulation in particular. The latest option can also be accessed if the user double-clicks on top of any simulation.
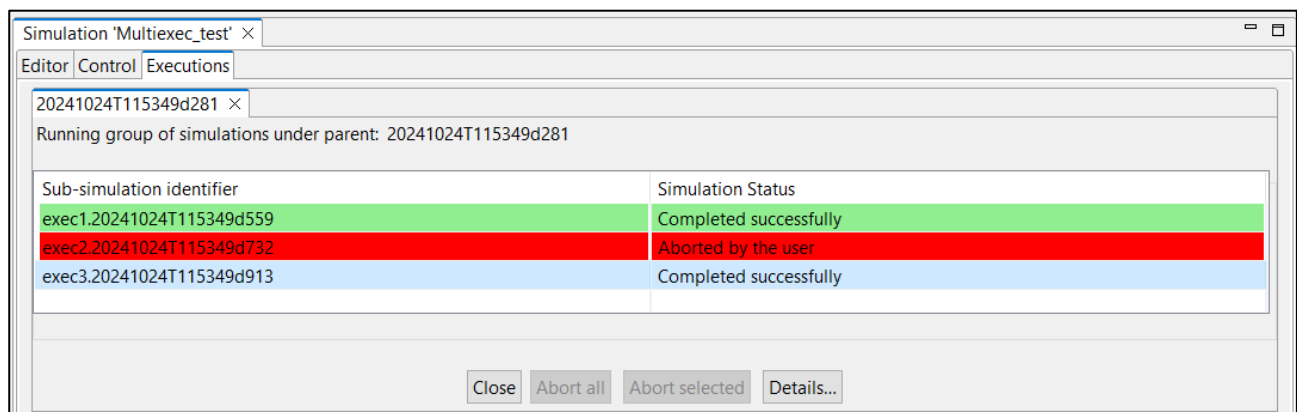
Figure 4-133: Grouping of simulations for the Iteration/Perturbation execution

4.4.4. Import and export simulations

The import/export capability provides the means to share all the information associated to simulations among different openSF instances.

4.4.4.1. Export simulation

The data needed to import a simulation consists in two files, obtained through a previously executed export operation:

- XML file, containing a partial export of the database elements making up the simulation in question.

- ZIP archive, containing the data files needed for the execution of the simulation. Furthermore, in case of an executed simulation, the archive includes the files related to that particular run.

From the openSF HMI, the export operation can be invoked from two different locations:

1. From the **Repository** menu. In this case, we need to navigate from the "Repository" menu down to the simulation that the user wishes to export. Next, right-clicking over it; the "Export" option appears for selection. This is illustrated in Figure 4-134 below.
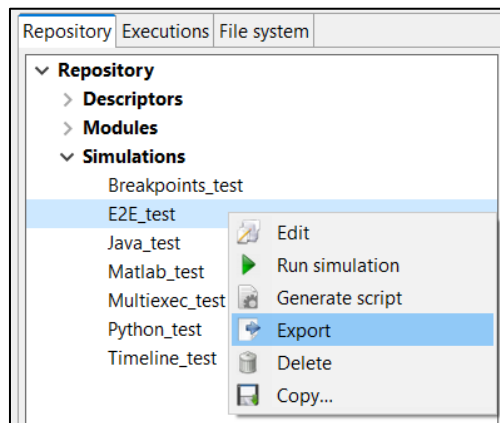
Figure 4-134: Export from the repository menu

1. From the **Executions** menu. If the user wishes to export a simulation that has been run, it can do so by accessing the "Executions" menu and selecting the Export operation upon the desired simulation, similarly to the previous case. This is shown in the next figure.
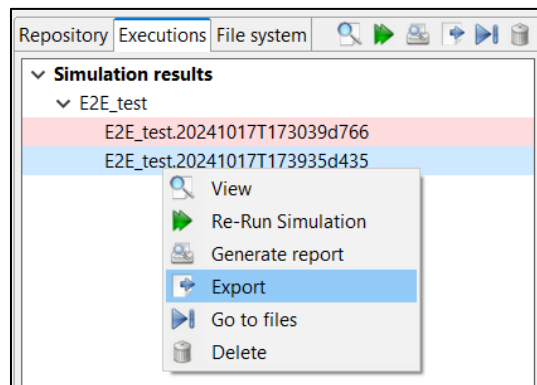
Figure 4-135: Export from the executions menu

In both cases, the output obtained as a result of the Export operation are two files (xml and zip) that are placed in the workspace folder. These files are needed for the import operation, and they ensure the creation in the database of the constituent elements of the simulation (i.e. descriptors, modules, tools, and simulation, as well as the provision of input and configuration files) needed for the simulation's execution. However, it is to be noticed that the executable files corresponding to the modules and tools are not included in the export operation.

The difference between both types of export (from the Repository menu and from the Executions menu) is that the Import of the latter type creates the simulation only in the Executions tab. That is due to its identifier (featuring the execution time stamp) and its state (Successful or Failed), which indicates that it is an executed simulation.

Once the export has been carried out, openSF reports the status in a dialog as the one shown below.
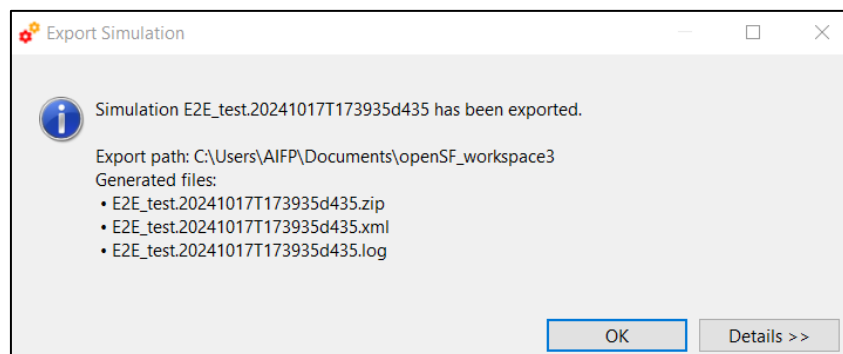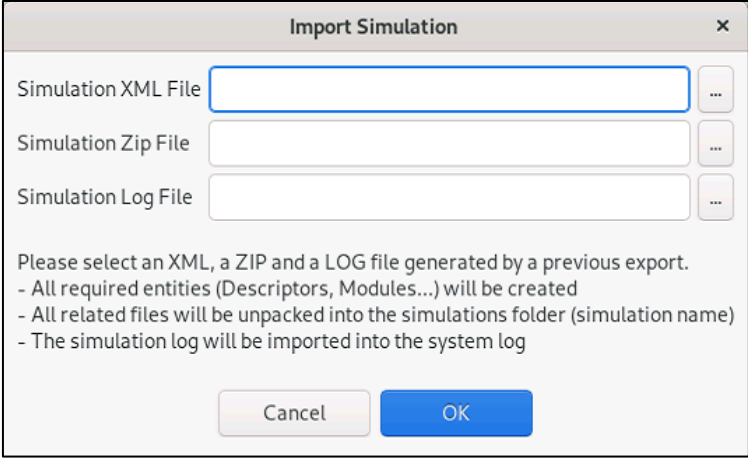
Figure 4-136: Successful execution of the export

4.4.4.2. Import simulation

Users can activate the import operation by accessing the "Repository" menu and clicking on the "Import simulation …" option, showing the dialog of Figure 4-137. The dialog requests the files needed to complete the operation, which are those generated by the export. The log file field is only used for simulations exported by openSF v3.7.1 or earlier, where the ZIP archive did not include the simulation logs.

Figure 4-137: Inputs requested for the import

The user can navigate through the file system to access the files by clicking on the buttons appearing at the right side of each input field. Upon completion, openSF reports on the status of the operation. Once the import has completed, the simulation and all the relevant components (descriptors and modules) will appear in the Repository tab of the Navigation view. The same applies to the imported simulation itself if was not executed. On the other hand, executed simulations will appear in the Executions tab.

4.4.4.3. Export module of a simulation

This capability deals with the possibility of exporting the data associated with a module that has already taken place in a simulation. The export functionality exports the data related to only one module of a given simulation. Thus, the data exported are the module configuration and input files.

From the openSF HMI, the export module operation can be invoked from the simulation execution window by navigating down to the module that the user wishes to export. Next, right-clicking over it; the "Export" option appears for selection. This is illustrated in Figure 4-138 below.
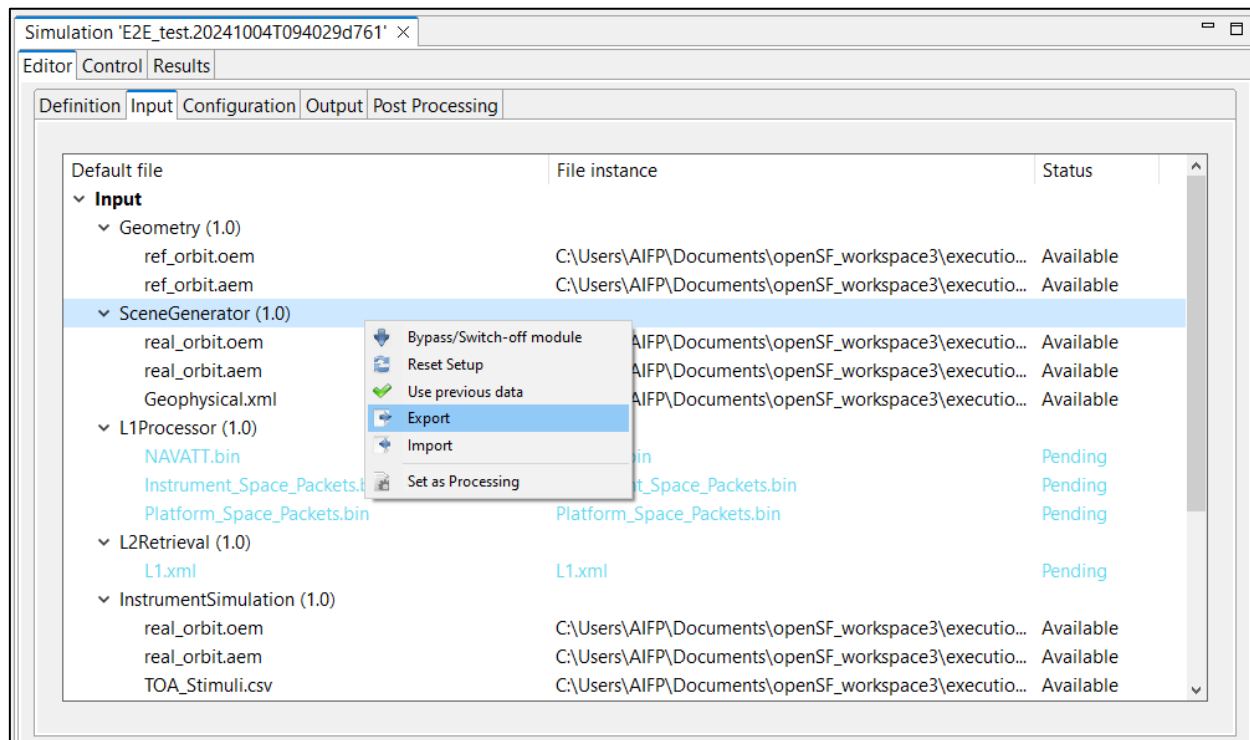
Figure 4-138: Export module from the Simulation Result view

When clicking "Export", the user will be asked for the file where to save the zip. The default option is the workspace folder, but this location can be changed. This file is needed for the import operation providing the input and configuration files needed for the simulation's execution. However, it is to be noticed that the executable files corresponding to the modules and tools are not included in the export.

4.4.4.4. Import module of a simulation

From the openSF HMI, the import module operation can be invoked from the simulation edition window by navigating down to the module that the user wishes to import to. Next, right-clicking over it; the "Import" option appears for selection. This is illustrated in Figure 4-139 below.
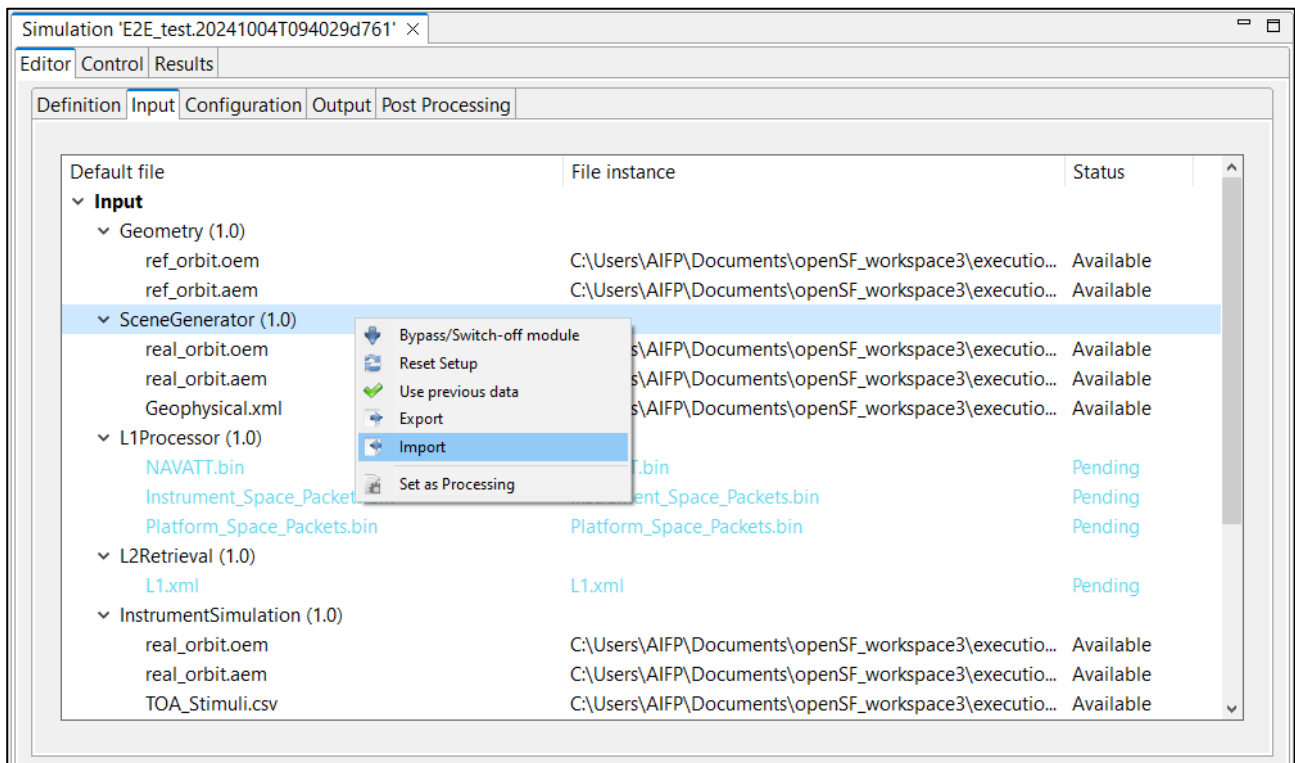
Figure 4-139: Import module from the Simulation edition view

A module data 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 module exists in the target openSF instance.

4.4.5. Simulation script generation

| I | E |

This functionality is provided to create and save a file script to enable the external execution of the simulation. This script file, which e.g. in Linux is named "*<simulation_name>*.sh", will be saved in the simulation folder as every needed input and configuration files. This script contains all the environment variables definitions and calls for modules' executions. The script can be executed from command line and it requires no parameters. Furthermore, scripts contain code to check the exit status of a module and exit with an error in case one module fails.

Note that the script is always generated when executing a simulation. The simulation script generation functionality just generates the script, without having to execute the simulation.

It is important to notice that the execution of this script (outside openSF) will not rely on any of the openSF provided functionalities. In fact, while the execution will be mimicked, the error handling and results storage capabilities provided by openSF will not be in place. Moreover, the simulation will be executed without resorting to the parallelisation capabilities provided by openSF.

4.4.6. Multi-node simulation

| E |

openSF has the ability of orchestrating the remote execution of one (or several) modules in a simulation.

For the sample test simulation scenario shown in Figure 4-140, the user can choose which machine to use for each module to execute. Note that the remote execution in openSF relies on mounting a remote file system shared by all instances of openSF executing modules of a same simulation.

Indra Deimos
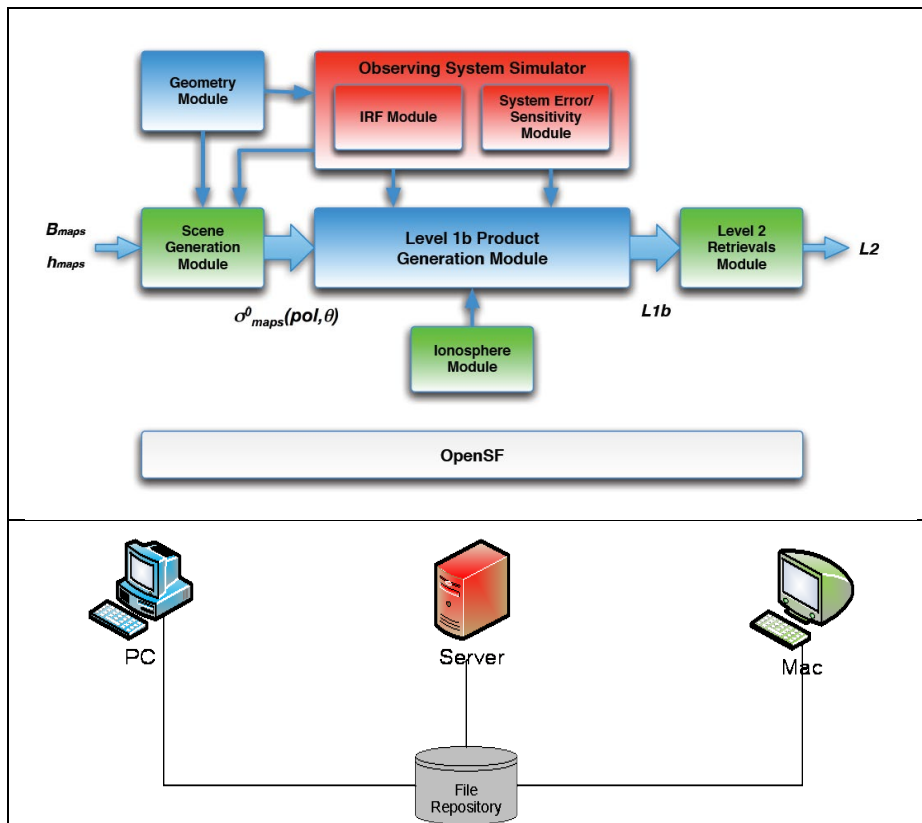
Figure 4-140: Outline of a simulation scenario

4.4.6.1. Remote machine management

For this purpose, a set of remote machines can be configured and managed in the system configuration.

When the user selects the "Remote" option from the menu "System", the window shown in Figure 4-141 will show up.
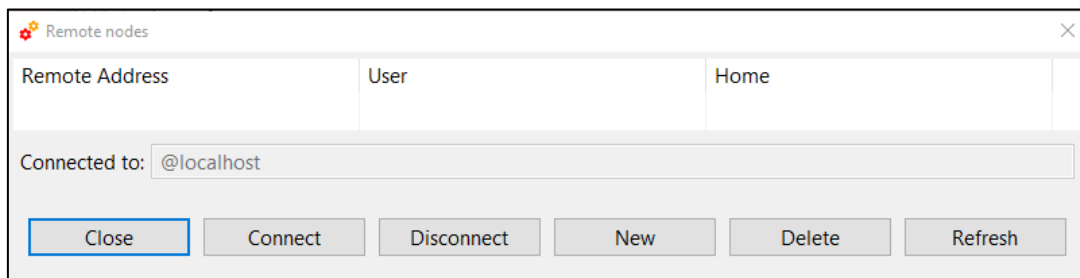
Figure 4-141: Remote machines management window

In the bottom of the window there are five buttons, which allow users to perform different actions over the remote machine configuration. Users can connect or disconnect from a remote machine, create a remote machine reference, delete a remote machine configuration or refresh the list of available remote machines.

The central area of the window shows a list with the remote machine configured in openSF. Users can obtain information about the remote machine address, the user that connects to the machine and the remote path where openSF instance is installed.

The bottom of the window shows a label with the remote machine currently connected. In the case exemplified, the application is not connected to any remote machine so the local installation of openSF is used for storing the execution products (identified by "localhost" label).

4.4.6.2. *Connect to a remote machine*

Users can configure openSF to produce the simulation execution products in the file system of a remote machine. For this, the user has to select a remote machine from the list, and click on the "Connect" button. Automatically the file system is connected to it, and then, the name of the selected machine is shown beside the label "Connected to".

Connecting to a remote machine file system means that the Simulation system folder used by openSF is located in a remote machine instead of in the local one.

4.4.6.3. *Disconnect from a remote machine*

Users can disconnect from a remote machine and rely on the local machine file system for simulation execution. For this, the user has to click on the "Disconnect" button. In case openSF is already connected to a remote machine automatically the file system is disconnected from it, and then the label "localhost" is shown beside the label "Connected to".

Disconnecting from a remote machine file system means that the simulation system folder used by openSF is the one in the local machine.

4.4.6.4. *Configure a new remote machine*

If the user wants to create a new remote machine reference, he has to click on the "New" button, and a dialog will be shown by the application, as it is can see in Figure 4-142.

Figure 4-142: Create new remote machine

A remote machine has four characteristics fields:

- **User**, it is the user who connects to the remote machine

- **Password** (optional), it is the password of the user. In case the ssh key has been setup this field is not required

- **Machine Address**, it is the address (IP or verbose) of the remote machine

- **Home**, it is the location in the remote machine where openSF instance is installed

Fields User and Home have a default value, that are:

- User: "openSF" (created during the installation of the application)

- Home: $E2E_HOME (created during the installation of the application)

It is mandatory that all fields have a correct value with the only exception made for the password field, that can be empty.

When the user enters all the information correctly and clicks on the "OK'" button, openSF attempts to connect to the remote machine using the configuration provided.

In case some field has been entered incorrectly (as for example the user or password to connect with the server are incorrect) openSF shows a message reporting the error (Figure 4-143), and invites the user to enter the correct information or cancel the creation.
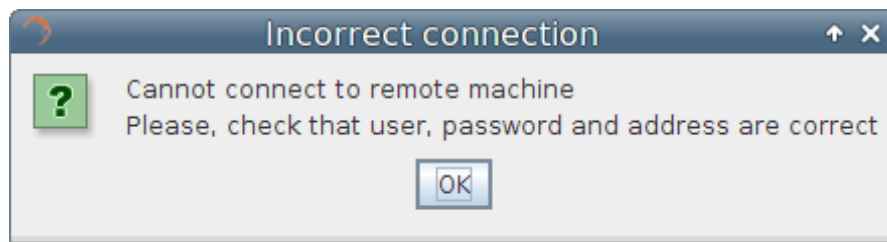
Figure 4-143: Remote machine is unreachable

If the user clicks on the "Cancel" button on the remote machine settings window, no action is performed.

4.4.6.5. *Delete a remote machine*

To remove a remote machine configuration, the user has to select the remote machine to be removed from the list, and click on the "Delete" button (Figure 4-144). A new dialog is shown to confirm the action. If the user clicks on "Yes, delete", the entry is deleted.
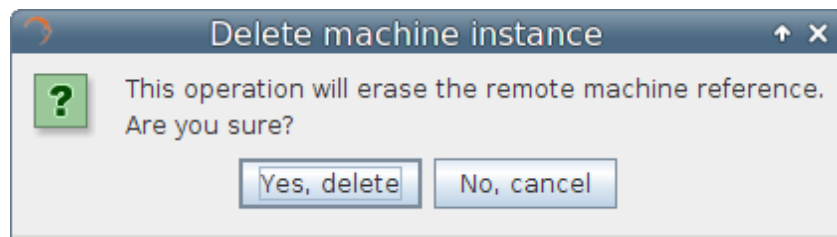
Figure 4-144: Confirm deletion operation

4.4.6.6. *Refresh remote machine list*

The remote machine management dialog provides the capability to refresh the list of remote machines to which openSF can be connected to. This capability is useful e.g. in a situation when the user wants to recover an existing configuration upon upgrading to a new version of openSF.

4.5. Preferences

The "System" menu, shown in Figure 4-145, gives control the general characteristics of the whole openSF system.
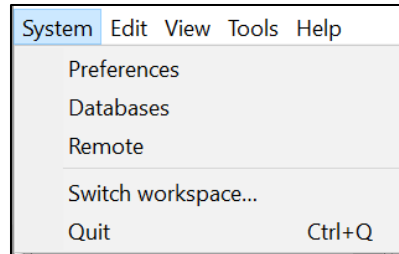
Figure 4-145: System Menu

Selecting this menu option presents the dialog presented in Figure 4-146.
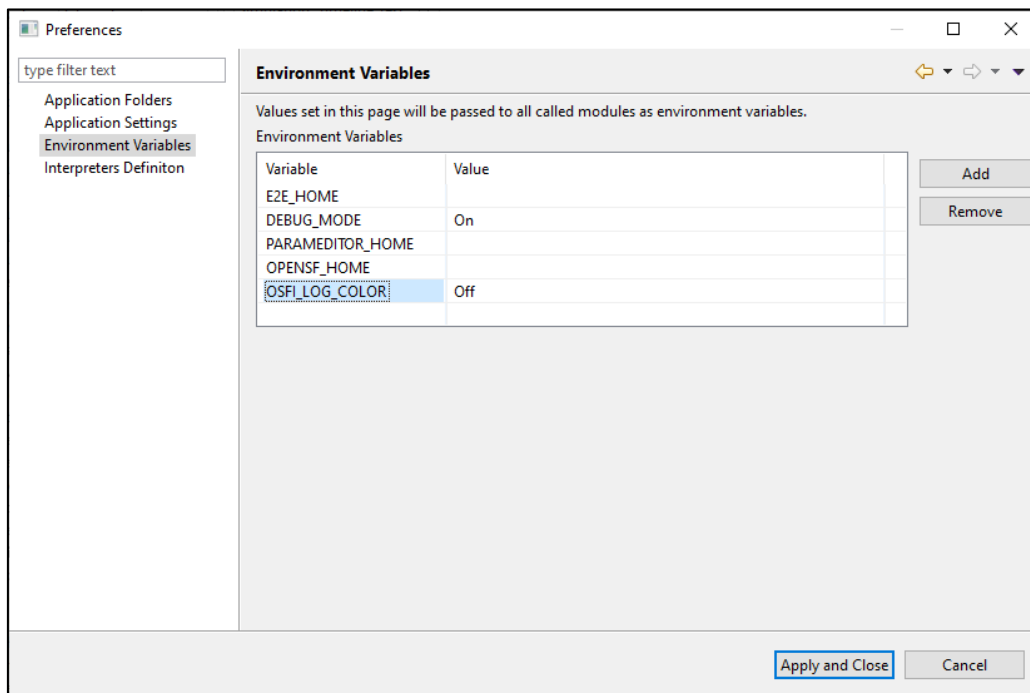
Figure 4-146: Environment variables

In this dialog users can modify some of the characteristics of the system.

4.5.1. Environment variables

A list of environment variables and associated values are shown in a table (Figure 4-146). Once a module or a product tool is being executed, they can access these variables if they need them because the system makes them available to the execution process. Users can "add" or "remove" an environment variable using the given buttons. Double-clicking on an already existing variable the user can edit its name and value.

Note that all other variables that were already present in the system/user environment when openSF was started will also be passed to the modules. In case of conflict, the variables defined in the preferences replace (with a few exceptions, see below) those already in the environment with the same name.

$E2E\_HOME$ must to be defined here so it is exported as an environment variable (see Sec. 3.3.1 for reference). There is an environment variable recognized by OSFI called $DEBUG\_MODE$ that controls the verbosity of some module executions. Setting to "On" or "Off" can enable or disable this output.

OpenSF handles in a special way all environment variables used to customize the search for dynamic libraries. For %PATH% (on Windows), $LD_LIBRARY_PATH (on Linux) and $DYLD_LIBRARY_PATH (on macOS), openSF prepends the value specified in the preferences to the value exported externally in the environment.

macOS users must be aware of the limitations in using environment variables in script execution enforced by Apple with the introduction of the System Integrity Protection (SIP) security feature in OS X 10.11 (El Capitan), which does not allow critical environment variables (such as $DYLD_LIBRARY_PATH) to be passed by openSF.

4.5.2. Application settings
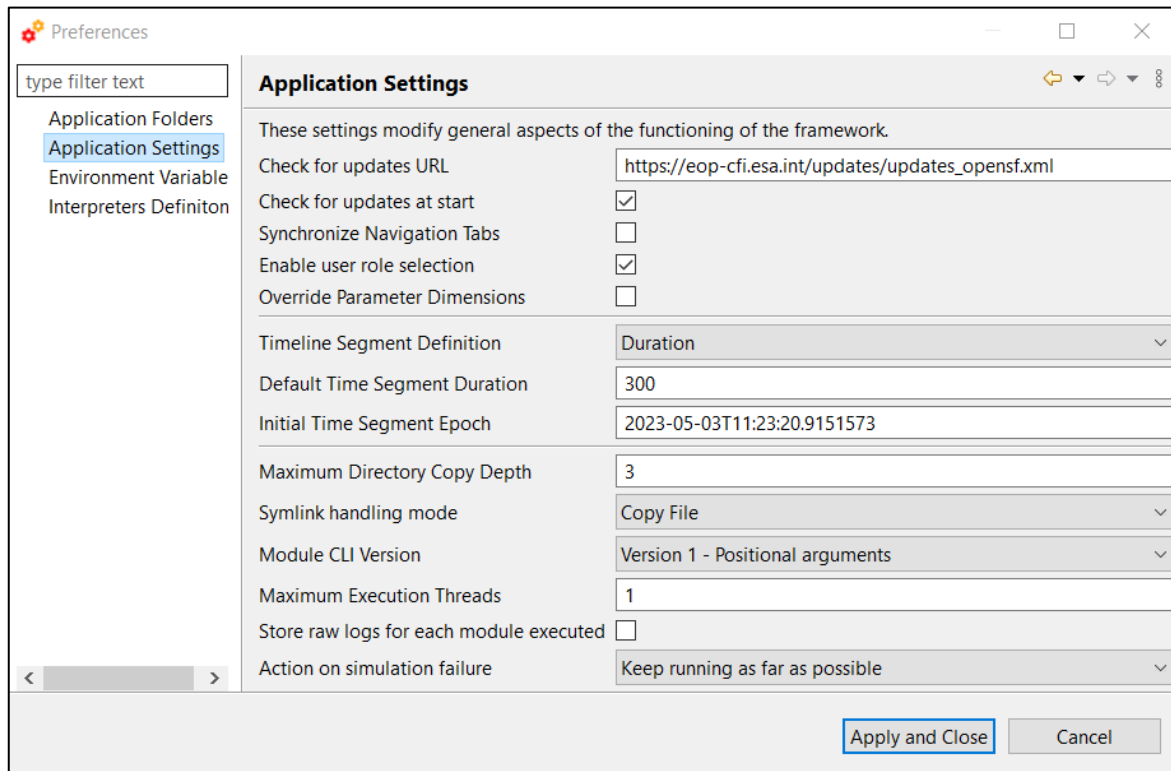
Figure 4-147: System Applications settings

Under this category (Figure 4-147) users can change the following default system parameters:

❑ *Check for Updates URL*. The configurable URL where openSF looks for software updates.

❑ *Synchronize Navigation Tabs*: Shows the appropriate simulation folder in the repository view when a simulation result is opened.

❑ *Enable user role selection*: When enabled, a custom menu is shown in the toolbar to allow switching between "Normal" user role and "Developer" user role (see section 3.2.2).

❑ *Override Parameter Dimensions*: When a parameter is changed, its dimensions may vary. When this option is ticked, this would generate a "Dimensions inconsistent with value" error. When this option is unticked, the parameter dimensions are updated according to the new value.

❑ *Timeline Segment definition:* Specifies how the timeline segment time should be interpreted (section 4.4.2.4).

❑ *Default Time Segment duration:* Default duration of the new Time Segments in seconds.

❑ *Initial Time Segment Epoch:* Initial Epoch for new Timeline Scenarios. This field shall be compliant with the CCSDS ASCII Time Code A format (YYYY-MM-DDThh:mm:ss.ddd).

❑ *Default execution strategy*. The user can choose if the default execution is data-driven or time-driven.

❑ *Maximum Directory Copy Depth*: Defines the maximum depth of the directory tree used when copying files and folders before a simulation is executed. This copy operation typically includes copying input and

configuration files/directories into the simulation folder. The use of this value is widespread and is designed to avoid possible infinite loops caused by symbolic links.

❑ *Symlink handling mode*: By default, when a symbolic link file or folder is used in a simulation (as a configuration or input), the actual file or folder referred to by the source element is copied in the simulation folder ("Copy File") (see section 4.3.4). The user can instead choose to create a symbolic link to the original source ("Copy Link") or ignore symbolic links altogether ("Ignore"). In this latter case, the symbolic links are not considered when creating the execution folder. If some module refers to files/folders containing symbolic links, the user is responsible for manually placing them where the module expects them to be found.

❑ *Module CLI Version*: Version of the Command Line Interface used when running a module. Version 1 uses positional arguments, while version 2 uses flagged options (see [AD 1][AD 1]).

❑ *Maximum Execution Threads*. Sets the maximum number of modules that can be executed in parallel during a simulation execution. The recommended value corresponds to the number of cores of the machine were openSF is installed. Possible values are: 0 (number of cores of the machine), 1 (no parallelisation enable) or N (the number of modules that may be executed in parallel). Keep in mind that it is allowed to insert a maximum number of execution threads higher than the number of cores of the machine and the impact is that a core may have to deal with more than one thread. Nevertheless, a warning message will be shown to the user whenever entering a value higher than the machine's number of cores.

❑ *Store raw logs for each module executed*: if disabled, a single log file will be generated for each executed simulation. The file contains general status messages, plus the messages from all modules formatted according to [[AD-E2E]]. However, if this option is enabled, a log file will be created in the same folder containing the "raw" output (both with and without E2E-ICD format) for each module in the simulation. This may be useful for debugging some modules.

❑ *Action on simulation failure*: the default value is "keep running as far as possible" which means that if a module fails, everything that does not depend on it is still launched and runs to completion. Thus, the execution goes as far as possible without the outputs of the failed module, leaving the simulation in a stable state. The alternative is to "stop launching modules but allow those running to complete" which finishes sooner but results in a non-deterministic final state that depends on what else was running at the same time.

4.5.3. Application folders

Under this category users can inspect and change the locations for the following directories:

❑ *Program*. This is where openSF is installed. Cannot be changed.

❑ *Workspace*. This is the place where openSF stores its configuration and file-based databases. Can be changed from the File menu.

❑ *E2E_HOME*. This is the location to find the simulator data such as modules, input and configuration files. Most paths in openSF are interpreted relative to this folder.

❑ *Executions*. This is the place where all the files associated to simulation executions can be found. Execution scripts, report files and, by default, configuration and output files generated are going to be stored here.
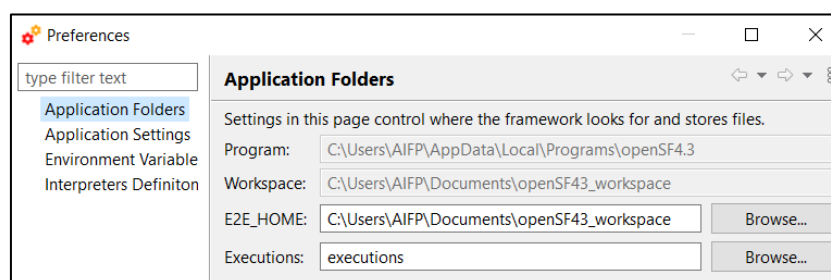
Figure 4-148: Application folders

If the user introduces the name of a folder that is unsuitable e.g. it does not exist ot is not writable when such access is required, warnings will be displayed at the top of the dialog. In addition, after changing the location of E2E_HOME, the user is recomended to restart.
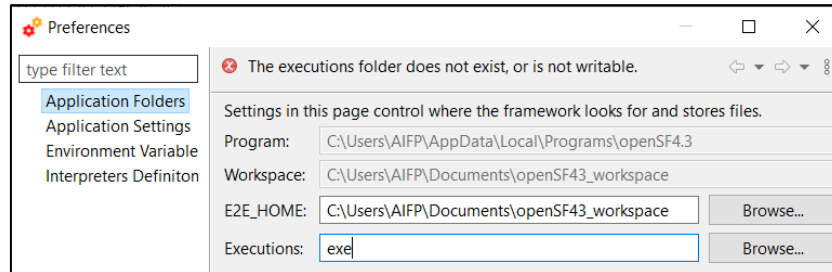
Figure 4-149: Wrong executions folder

4.5.4. Interpreters Definition

OpenSF allows the user to run modules not directly executable from the system command line. Such modules can be executed by means of an interpreter that can be specified in the system preferences (See section 6.3.1 for module's execution details). A number of built-in interpreters are available by default, while further ones can be specified by the user. Figure 4-150 shows a list of available interpreters, accessible by navigating to System → Preferences → Interpreters Definition.
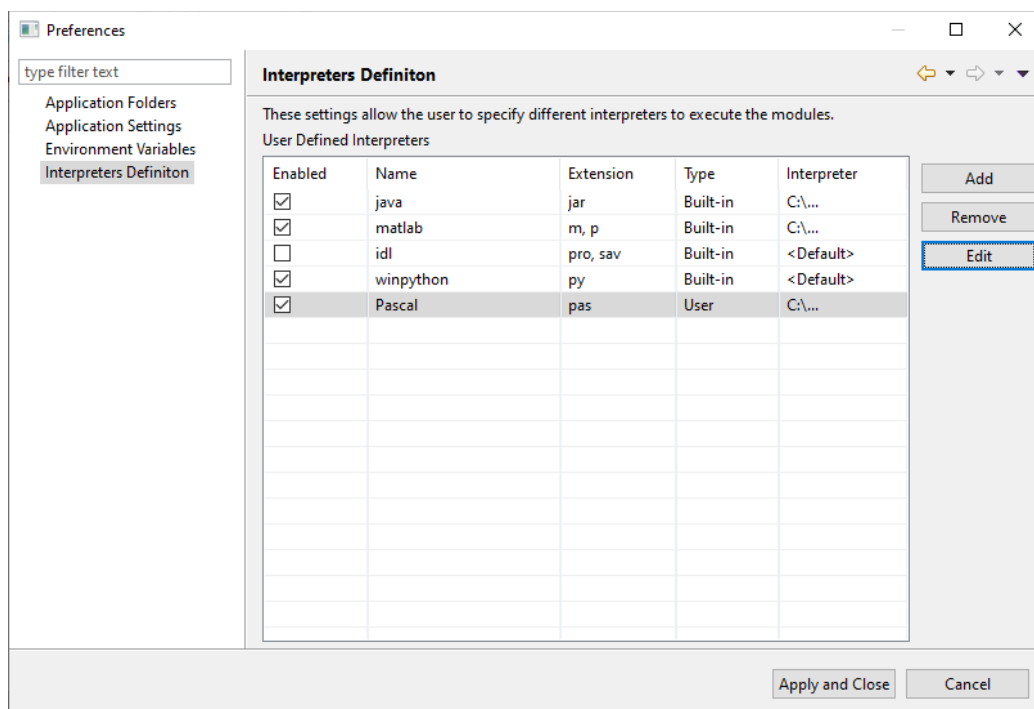
Figure 4-150 Interpreters definition

When a module is executed, openSF checks if its extension is compatible with one of the available active interpreters, giving priority to the built-in ones. Interpreters can be activated and disabled using the first column in Figure 4-150.

Built-in interpreters' path can be modified by selecting the desired interpreter and pressing on the "Edit" button. This will open the pop-up in Figure 4-151, which allows the user to select an interpreter's path.
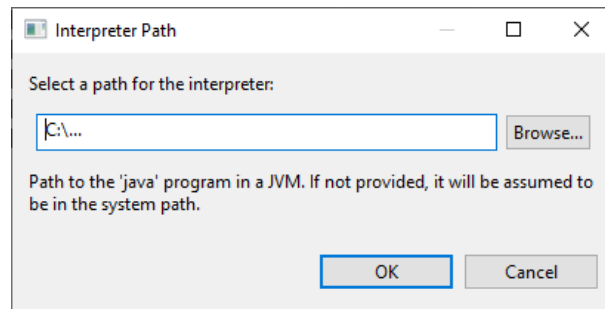
Figure 4-151 Built-in interpreter path definition

In case no path is given, the table in Figure 4-150 will display the placeholder "<Default>", which assumes the interpreter to be in the system path.

User-defined interpreters can be added, removed and edited using the corresponding buttons, as shown in Figure 4-150. When one of the last two options is chosen, the window in Figure 4-152 is shown.
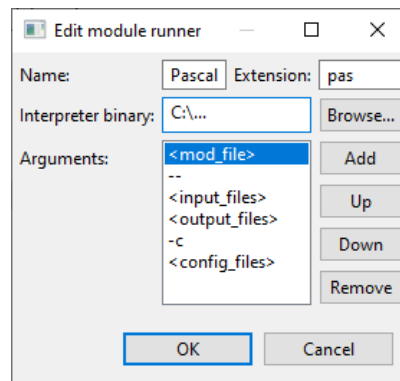
Figure 4-152 User-defined interpreter definition

Here the user can define a name, an extension, a binary path and a list of arguments. Note that only one extension can be specified. Note also that if different interpreters share the same extension, openSF does not resolve the ambiguity and therefore the user is invited to resolve it, e.g. activating only the desired interpreter. To add, shift upwards, shift downwards or remove an argument from the arguments list, the user can use the respective buttons. When creating an argument, the window in Figure 4-153 is shown.
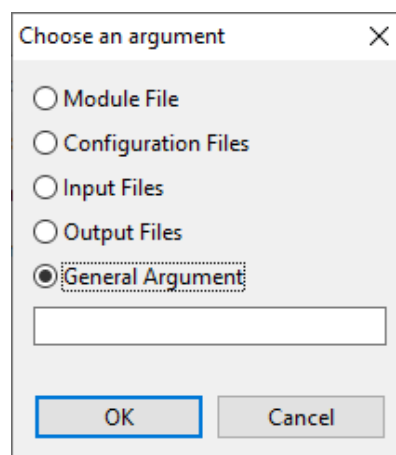
Figure 4-153 Interpreter argument definition

Here, the user can choose to specify a general argument as a literal string or to insert a placeholder to indicate openSF to insert:

Placeholder	Representing
<mod_file>	The module file name that is being executed
<input_files>	The module input file names
<output_files>	The module output file names
<config_files>	The module configuration files (global and module-specific)

The actual output generated by the input, output and configuration files placeholders depend on the global setting for the CLI version to be used: running with CLI version 1 generates a single argument with all files separated by commas, while using CLI version 2 generates multiple arguments with the appropriate flags e.g. --global gcf.xml.

4.6. Miscellaneous

M	I	A	E

A series of documents and utilities that are available from openSF is presented in this section.

4.6.1. About openSF

I

From the "help" tab of the menu bar, the "About openSF" functionality can be accessed. The system will show a dialog with the copyright and license scheme for the openSF platform.
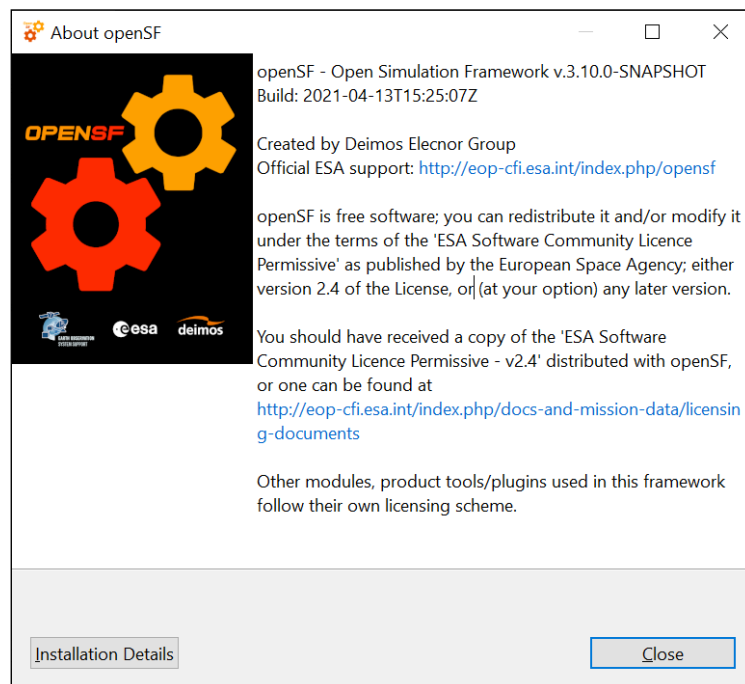
Figure 4-154: openSF About View

4.6.2. Embedded documents

M	I	A	E

A series of documents are embedded and deployed along with openSF. Users are able to view these documents launching the proper software application. The applications for viewing the supported file types are the ones defined as default applications for current OS.
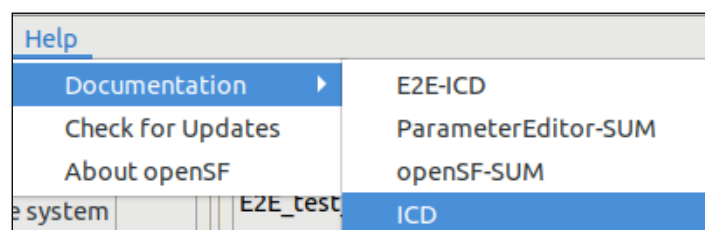
Figure 4-155: Help documents tree view

The supported file types and the typical viewers are the following:

- PDF: "Portable Data Format"; Acrobat Reader

- HTML: "Hypertext Markup Language"; Web Browser

- Plain Text: Notepad, Vim, Emacs …

4.6.3. CPU usage

E

This dialog is helpful to analyse the CPU core use when the module execution is parallelised by openSF.

4.6.3.1. Linux

Accessing this functionality, the system will show a custom dialog with occupation of CPU cores by machine processes.
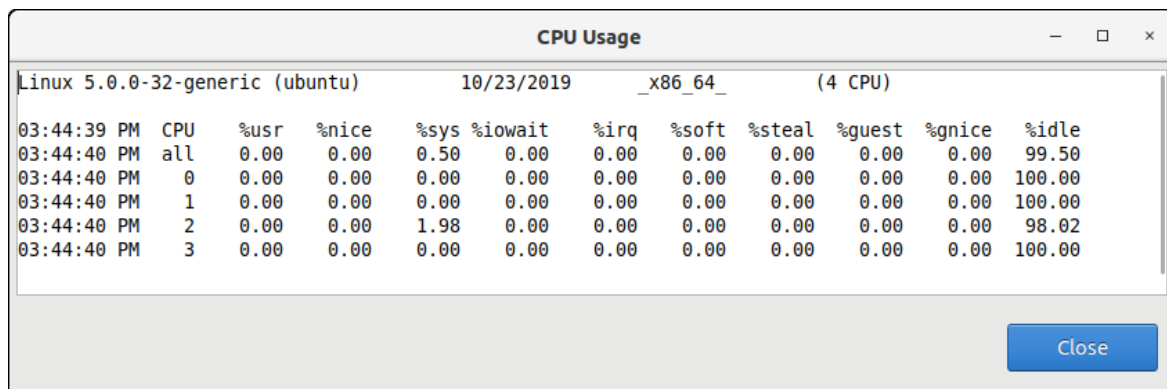
Figure 4-156: CPU Core Usage view

4.6.3.2. macOS

The macOS Operating System already provides a default application to convey information about CPU usage, the 'Activity Monitor'. Accessing the CPU Usage functionality in openSF therefore launches the 'Activity Monitor' external application.

4.6.3.3. Windows

Similarly to macOS, in Windows the system program "Task Manager" is launched.

5. ANNEX A: ERROR MESSAGES

| I | A | E |

openSF platform controls its correct behaviour with an error handling system. Users are informed about the nature of the error and a possible way to correct it.

In general, every time an input value is needed, the platform will perform a validation process. If the input does not comply with the needed format, the user will be informed with a self-explained message.

Errors not shown as part of the graphical interface are not controlled messages. They correspond to messages from the standard output or error stream.

When executing a simulation, modules raise their own error messages and they are intercepted by the system and shown as log messages in the execution view

Here is a list of different kinds of raised errors:

Module	Operation	Error	Comments
System.Configuration	Adding a new variable	Validation error	Follow the instructions to correct the value
System.Tools	Accepting changes	Tool addition failed	The user has chosen a duplicated identifier. Please provide a different identifier
		Validation error	Follow the instructions to correct the value
	Deleting a tool	Database error	Possible database failure. Is the database running?
	Executing a tool	File IO error	Follow the instructions
Repository.Descriptors	Accepting changes	Descriptor modification failed	Possible database failure. Is the database running?
		Descriptor addition failed	Duplicated identifier chosen. Please provide another identifier
		Validation error	Follow the instructions to correct the value
	Adding an IO file	Validation error	Follow the instructions to correct the value
		A descriptor shall not have associated two files with the same id	Please choose another identifier
	Deleting a descriptor	Database error	Possible database failure. Is the database running?
Repository.Modules	Accepting changes	Validation error	Follow the instructions to correct the value
		Module addition failed	Duplicated identifier. Please provide a different identifier
	Deleting a module	Database error	Possible database failure. Is the database running and configured?
	Creating a new version	Database error	Possible database failure. Is the database running and configured?
Repository.Results	Deleting a result	Database error	Possible database failure. Is the database running and configured?
Repository.Simulations	Accepting changes	Validation error	Follow the instructions to correct the value

		Database error	Possible database failure. Is the database running and configured?
		Simulation addition failed	Duplicated identifier. Please provide a different identifier
		Simulation modification failed	Possible database failure. Is the database running and configured?
	Adding a simulation	Simulation identifier cannot be void	Please provide a valid identifier before adding a simulation
	Deleting a simulation	Database error	Possible database failure. Is the database running and configured?
	Generating a script	File IO error	Follow the instructions
	Iterating parameters	Invalid list of values	Please input a comma-separated list of valid values (no blanks)
		Validation error	Follow the instructions to correct the value
	Removing a simulation	There is no simulation selected	Please select a simulation to remove
	Running a simulation	Cannot run an unnamed simulation	Please input a valid identification to the simulation
		Missing configuration files	Provide the missing GCF or LCFs in order to run the simulation
		Validation error	Follow the instructions to correct the value
		File IO Error	Follow the instructions
	Setting limits	Validation error	Follow the instructions to correct the value
Executions.Log	Dumping the log	File IO error	Follow the instructions

6. ANNEX B: DEVELOPING MODULES FOR OPENSF

M

This section is aimed to module developers that are looking for further information about openSF module integration.

openSF can integrate as a module every executable code that follows the requirements described in [AD 1], and execute them within the system.

Nevertheless, module developers must take care of the following points:

- Memory handling is responsibility of the module. openSF does not manage memory assignments and does not destroy any data structure created by the module;

- A module can create child processes, but their management is still on the module developer's side;

- openSF does not detect when a module execution is "halted" or in an infinite loop. It is suggested to periodically send some logging information to openSF (i.e. heartbeat) so that the user can identify a stalled module.

- Execution performance of the module may be slightly slower when orchestrated by openSF, because of the messaging interception/collection mechanism;

- Module developer is responsible of the error and exception handling as explained in [AD 1]. (Error Handling section).

Below there is a list of documents with relevant information on module development and the topics covered by each one.

- openSF Interface Control Document [AD 1]

  ➢ openSF interface specifications

  ➢ Module development guidelines

  ➢ Module development process

  ➢ OSFI Developer's Manual [RD 1]

  ➢ Integration libraries reference manual for each programming language.

- OSFEG Developer's Manual [RD 2]

  ➢ Error Generation Libraries reference manual

- openSF Architecture Design Document [AD 2]

  ➢ openSF architecture

  ➢ Interaction between modules and openSF

6.1. Precautions to ensure safe module parallelization

The functionality of openSF to allow parallel execution brings a certain responsibility to module developers. It is responsibility of the module developers to ensure that the modules are in fact parallelisable, e.g. that their implementation has the proper precautions regarding access to common resources. openSF can only assure synchronization of module execution and it must rely on modules being "well behaved" with respect to parallel execution.

In order to ensure safe module parallelisation, module developers should ensure that modules 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.

![INDRA logo]

openSF
System User Manual

OPENSF-DMS-TEC-SUM01
4.9
18/06/2025

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 completes 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-constant) 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 a data race occurs, 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 it can be guaranteed 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.

Element like mutexes and critical sections can be used 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, the programmer still needs to ensure that the code is safe in the wider context: if more than one variable needs to be modified then the mutex needs to be held across the whole operation rather than for each individual access, otherwise the invariants of the 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 a separate copy of the data is held 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, synchronization is always needed if two or more threads are going to operate on the same data.

6.2. Environment variables

The user can customize the environment variables used by openSF and that are available to the modules and tools during simulation execution, tailoring them to his system. These variables can be set in the system preferences window (see section 4.5). An environment variable that is always exported is $E2E_HOME, described in sec. 3.3.1.

6.3. Module pre-requisites

In most cases, openSF runs modules by invoking them as binaries from the simulations folder. This means that it is responsibility of the module developer/integrator to provide any dependencies (e.g. libraries, interpreters, etc.) and to perform the setup needed for such modules to work correctly. The means are varied and depend on the type of module (compiled binary, script, etc.).

For example, a module written in C++ may link statically against all its dependencies so that the resulting executable does not depend on any dynamic libraries that would have to be found and loaded on start, although this may not always be possible depending on the library and the system. Alternatively, dependencies can be provided in a way that may be located by the module, so e.g. a Python module that makes use of OSFI-Python or other libraries may access them by:

❑ Installing them in the site-packages directory of Python,

❑ Including them in the PYTHONPATH environment variable, or

❑ Distributing them alongside with the module, making the script itself look for them in a known location relative to the module (e.g. `../libs/OSFI/Python`).

For more information on the matter of module development and deployment, look at the documents mentioned in section 6.

6.3.1. Modules not compliant with E2E Generic ICD

OpenSF offers some support to run modules non-strictly adherent with the Generic E2E Interface Control Document [AD 1]. If such modules do not comply with the standard CLI, they can be wrapped and called by means of a specified interpreter. The arguments of the given interpreter can be freely specified and ordered (See section 4.5.4) and they may include:

1. The list of input files
2. The list of output files
3. The list of configuration files
4. The module file

Note that the lists themselves are specified in the same fashion as described in [AD 1].

6.3.2. IDL

Note: IDL module execution is deprecated and not under further development.

To execute modules in IDL with openSF is necessary to have IDL software installed on the computer.

openSF has been tested with the following versions of this software: version 7.1, 8.0 and 8.1.

If the user has a previous version, the application may not work. It is recommended to have installed at least IDL 7.1, and whenever possible version 8.0 or later.

An important requirement for the correct functioning is that IDL is installed in the default path, because if not some features of the OSFI library will not work properly. This problem is related with the ConFM module, which uses some internal classes of IDL that must be in the default path in order for the application to find them. IDL looks in fact for these classes only in the default directory, and if it does not find them generates an error.

For IDL 7.1 the default path is '/usr/local/itt/idl' and for IDL 8.x the default path is '/usr/local/itt/idl/idl'.

Furthermore, IDL provides three types of licenses according to the user needs, as can be seen below:

- IDL development: Full license for IDL that allows to the user to use all its functionalities. Users can access to the IDL Development Environment, the IDL command line, and having the ability of compiling and executing IDL .pro files and executing .sav files.

- IDL runtime: Allows executing IDL programs precompiled and saved as .SAV files, or .pro files without any type of restriction.

- IDL virtual machine: It is a free license that allows to the user to execute IDL programs precompiled and saved as .SAV files, or .pro files. This kind of license has a few restrictions, like displaying a splash screen on start-up, callable IDL applications are not available.

To execute a .sav or a .pro file without any type of restriction it is necessary to have installed the development license or the runtime license on the computer. If user wants to generate .sav files by compiling .pro files, it is mandatory to have the development license. If the user only has the virtual machine license, he can execute .pro and .sav files but with restrictions, as many functionalities are not available for this type of license.

6.3.3. MATLAB

To execute modules in MATLAB with openSF, MATLAB software must be installed on the computer. The only requirement is that MATLAB version must be R2009a or later, with the corresponding license.

Like the example given in section 6.3.4 for Python, the MATLAB module can access any extra libraries it needs by simply adding the necessary directories to the MATLAB path in the first lines of the code. In particular, the libraries that need to be accessed should be placed in a known location relative to the module files. Similarly, any other environmental preparation needs to be performed in terms of modules before actually starting working.

6.3.4. Python and other scripts

The framework also executes script-based modules, such as Python or shell scripts. Currently, they are invoked as normal programs, so they must be marked with execution permissions. The choice of interpreter must be somehow recognizable for the system e.g. with the customary hash-bang line:

```
#!/bin/sh                                          (POSIX shell scripts)
#!/bin/bash                                           (BASH scripts)
#!/usr/bin/env python                   (Python, system default version)
#!/usr/bin/env python2/3             (Python, script choice of a version)
#!/opt/bin/PowerShellCore6/pwsh        (Other, custom script interpreters)
```

Since .sh/.bat files are executed directly, Windows users cannot run POSIX shell modules and Linux/Mac users cannot run CMD batch modules. However, the system is extensible and new "interpreted file" types can be added in future versions.

6.3.5. Python scripts execution in Windows

Due to the possibility of having more than one Python interpreter (different versions, venvs for different modules, etc.), a method to choose the correct interpreter for each script is necessary. Giving the version of the interpreter to be used via shebang lines in the beginning of the Python script solves the issue.

Since Windows does not allow shebang lines to choose between which Python version to use, the solution is to use a launcher[12] for aiding in the location and execution of different Python versions, allowing the scripts to indicate a preference for a specific Python version.

---

[12] PEP-397, "Python launcher for Windows"

Thus, in Windows, the Python interpreter "py.exe" should be preferred instead of a specific interpreter version i.e. "python.exe". This Python interpreter, as well as others interpreters from the compatible languages, can be chosen by accessing Systems → Preferences → Interpreters Definition, as thoroughly described in section 4.5.4.

6.4. Handling large reference data files

In general, circulating large files around a simulation as inputs and outputs should be avoided as much as possible, since openSF copies them into the simulation folder for reproducibility. For example, for reference files that seldom or never change, such as planetary ephemerides, gravity models or surface digital elevation models (DEMs), it makes little sense to have one copy for each simulation that is executed.

However, while these datasets themselves may be static, a simulation may switch between different versions of an "equivalent" dataset due to other reasons. For example, a few DEM files with varying resolutions can be present, offering a trade-off between simulation execution speed and precision of results.

The first thing to do would be to have the files hardcoded if they are *always* going to be the same. If it does not make sense to do so, before having it as an input file, it can be selected by a parameter in the configuration file. Here there are two possibilities: one would be directly pointing to the corresponding file/folder and the other would be to have a specific number of models.

Therefore, the solution is to point the module(s) to the reference files in question by using parameters in the global or local configuration files. There is a variety of ways to organize such access, as described in the following sections.

6.4.1. As a FILE or FOLDER parameter

If a module uses a single code path that reads data from multiple files but always in a specific file format, then it may make sense to have a file or folder parameter.

Normally, files or folders are referred to with their own specific type of parameter. The advantage over the general STRING type is that, if the path that is mentioned is relative, OSFI automatically resolves it against the value of the E2E_HOME environment variable.

```
<parameter name="earthDem" type="FILE">
    <!-- Relative path, E2E_HOME implied -->
    data/earth/topo/ASTER_GDEM2.zip
</parameter>
```

The example above is the recommended practice, since it allows the module configuration files to be easily distributed, as long as the reference data files are placed in folders inside E2E_HOME. It is applicable if the different data files are in some way interchangeable, such as with the compatible DEMs with different resolutions. If they are fundamentally different, consult the section 6.4.2 regarding enumerated choices.

If the data files cannot be inside E2E_HOME, there are two solutions. First, an absolute path is valid, although it makes the modules and/or simulator more difficult to distribute and setup: either the absolute paths would need to be kept on each machine, or the base configuration files modified to change the paths.

```
<parameter name="earthDem" type="FILE">
    <!-- Absolute path valid but not recommended -->
    /sat_sim/topographical/ASTER_GDEM2.zip
</parameter>
```

Note that the interpolation of the E2E_HOME environment variable is automatic for parameters of type FILE and FOLDER where the value is a relative path. The variable does not need to, and in fact it cannot, be explicitly mentioned. To clarify, **no other variables are used and OSFI does not recognize any kind of variable interpolation syntax such as $ or %**. It is not recommended, but the value of a parameter can be read instead of as a file as a string, so OSFI does not do that interpolation.

```
<parameter name="planetEphInvalid" type="FILE">

    <!-- NOT valid: only E2E_HOME is used -->

    $EPH_LOC/de431.bsp

</parameter>
```

6.4.2. As an enumerated STRING parameter

If a module uses different models for some purpose that vary widely in the set of data files to be used, it may make sense to have the selection parameter be an "enumeration", at least conceptually.

In that situation, each value is recognized by the module as a "known name", thus using a parameter of type STRING or, alternatively, an integer. The accepted values would need to be described in the documentation for the module. As an example, consider a module that may accept multiple models for the physical and chemical characteristics of the atmosphere of Mars:

```
<parameter name="marsAtmModel" type="STRING">MarsClimateDb_v5.2</parameter>

<parameter name="marsAtmModel" type="STRING">ExponentialNeutral</parameter>
```

In the above example, the module knows which files to access for each accepted value, normally by using paths that are hardcoded (relative to E2E_HOME). The exponential model could use a single file containing the initial conditions, or even module-internal data. The MCD code, on the other hand, will read a multitude of data files from a path inside E2E_HOME and interpolate from them.

7. ANNEX C: Packaging & delivering an E2E simulator

!

This section outlines the recommendations and steps required to package an E2E simulator and ship it to the user and those for the user to extract it.

To understand this section, the reader should be acquainted with the openSF elements (section 4.3) and its folder structure (see section 3.3.1).

7.1. Create the simulator

The integrator is recommended to generate the folder structure that will contain all the input files, configuration files, tools and modules of the simulations contained in the simulator. All the files used by the simulations should be placed in a common root folder. An example of folder structure is presented in section 8.2.1.

Note that modules are understood to be standalone tools, is the sense that their dependencies of the modules should also be made available in this folder in order to ease the extraction/deployment process.

Once the simulator root folder is assembled, set the E2E_HOME variable to the aforementioned root folder.

As a final step, the integrator must configure all the simulation elements (descriptors, modules, simulations, tools) in openSF. Notice that by placing all necessary files inside the simulator root folder all paths used to configure the simulation elements can be made relative to E2E_HOME. If absolute paths are used, one of the following should apply:

❑ The absolute path points to a location in a shared memory device and accessible by all the intended users of the simulator via the same path (e.g. a network server).

❑ The absolute path points to a location always available in the end user's machines (e.g. "C:\Program Files\" in Windows). In this case, the integrator shall list the file or folder as a requirement (or provide an installer that takes care of providing it).

Paths referring to a location outside the root folder must be avoided, so that the simulator is relocatable and thus can safely be installed in any location. For the same reason, symlinks pointing outside the root folder, or using an absolute path for the link, should be avoided. The same applies to contents of the configuration files for parameters of type FILE or FOLDER.

At this point, all the simulations and tools should run correctly.

7.2. Package the simulator

To package the simulator the integrator must export the database generating an XML file and create a compressed archive of the root folder (e.g. zip, tar.gz). Note that the openSF does not guarantee the portability of the database file itself, hence the creation of the XML file.

The shipping material then consists in:

❑ The archive with all the input files, configuration files and modules

❑ The database in XML format, possibly located in the same archive. If the database file is named `default_database.xml` and ends up in E2E_HOME (see section 3.3.1), it will be recognized by openSF and offered to the user when a new DB needs to be created.

❑ Platform-specific openSF package (see section 7.2.1)

Note that depending on the modules a platform-specific package may be needed e.g. a Fortran program that is compiled for a specific platform, or a Java/Python program requiring the platform-specific runtime/interpreter.

For a visual representation of the desired result, you can refer to the example simulator package located at `/resources/simulators/example_simulator.zip`. If the workspace-specific items (openSF.properties and DB files) mentioned in section 3.3.1 are removed, this may be done simply by archiving the workspace folder.

7.2.1. Package openSF

The openSF framework can be delivered by bundling the installer itself with the rest of the package. It may either be run silently by a custom installer/script provided by the integrator, or normally by the person performing the overall simulator installation. See the automation section below for more details.

In addition, it is possible to package an existing openSF installation, after removing any absolute paths from the configuration file. Note that this option is not officially supported and so it is not recommended.

7.3. Install the packaged simulator

In order to set-up the simulator, the following steps need to happen:
1. Extract the provided folder containing all the modules and files to a new folder, which will become the workspace (or E2E_HOME for shared installations, see section 7.3.1).
    - If the modules' dependencies were not packaged directly inside the provided zip, resolve those dependencies as needed (e.g. install Python, Java or other runtime).
    - If a `default_database.xml` file exists in the root of E2E_HOME, openSF will offer it to the user in the welcome and new DB dialogs. Refer to sections 3.3.1 and 3.6.2 for details.
2. Install openSF. For non-shared installations, select the folder created in step 1 as the workspace; otherwise select a new, empty folder. This step may be automated, see section 7.3.2 for details.
3. Set any required openSF configurations, such as:
    - Set E2E_HOME to the E2E simulator root folder
    - Set the required module runners (interpreters)
    - Set the CLI version to be used

At this point, the simulator shall be functioning and ready to be used.

7.3.1. Multi-user environment

In case of a multi-user environment, the integrator needs to balance flexibility and ease of installation with security. OpenSF uses the workspace to store data and to run simulations. This means that a user of the integrator should have writing permits at least to the "executions" folder in order to use it. Hence, one of the following options may be chosen:

- ❑ Have an independent openSF installation for each user. In this case, merely perform the steps of section 7.3 for each user that needs to use the simulator.
- ❑ Have a common installation in a non-writable folder. Any common customization can be performed in openSF_defaults.properties. In this case, either:
    - Have a separate E2E_HOME folder for every user, inside their own workspace. In this case, step 1 of section 7.3 is performed once per user while step 2 happens just once, leaving the "workspace folder" field empty during installation. When running openSF for the first time, each user would need to point it to their own workspace folder.
    - Share the E2E_HOME folder, denying writing permissions to the users. Every user will use a different workspace, and configure their E2E_HOME setting to point to this shared folder. In this case, all steps of section 7.3 happen just once. The setting of E2E_HOME can be set in the openSF_defaults.properties file in the installation folder (see section 7.4).

Note that in any case the database shall not be shared, since there is no protection against concurrent usage.

7.3.2. Automation

The extraction process can be eased by automating some of the steps with a custom installer script. Such script can in fact:
- ❑ Extract the simulator package to the future workspace.
- ❑ Install openSF silently. Refer to the primary source at install4j[13] for the command line options to pass.
- ❑ Set any required openSF configurations by pre-defining values in the configuration files, or any other methods defined in section 7.4.

---

[13] https://www.ej-technologies.com/resources/install4j/help/doc/installers/options.html

❑ Resolve any required dependency. An example of dependency resolution would be to install the right version of Python and possibly some specific packages with "pip".

7.4. Framework configuration

This section describes the most common configuration options that an integrator might wish to customise on the installed framework, *without* requiring a rebuild of openSF from the sources. Most of these options are customised by updating the configuration file (see below), but others (e.g. the splash image) require modifying other files.

Regarding the configuration files, the framework reads two separate files: one in the workspace root and one in the program installation folder (see section 3.3.1 for more details). Normally, the second option ("openSF_defaults.properties" under <OPENSF_INSTDIR>) should be preferred by an integrator delivering a simulator. This is because options in the "defaults" file automatically apply to all workspaces that the user might create or switch to while using the same openSF installation.

The settings themselves may be created in the openSF GUI itself as part of the process of creating the simulator (section 7.1). The integrator must however take care to only extract the relevant values (as described in the list below) to place in the delivered configuration file.

❑ Paths and environment:
  ➢ E2E_HOME can be set either by defining it as an environment variable external to openSF, or by creating a property **"env.E2E_HOME"** in the configuration file. Note that this step can be skipped if the root folder is already a subfolder of the workspace (provided that the database elements correctly point to them)
  ➢ The same applies to any other environment variable that the modules might require in order to run: openSF will take and forward any pre-defined environment variables, and the configuration file may be used to override those for all modules and simulations by defining a variable **"env.VARNAME"** with the desired value.
  ➢ Simulations folder: the **"openSF.simulationsDir"** option stores the path (absolute or E2E_HOME-relative) that will be used to write the application log and simulation execution files and folders. It must be writable by the user.
❑ Interpreter setup:
  ➢ The **"exec.rt.java"** option stores the path to the Java runtime executable used to launch ".jar" files as modules. The default is to launch "java" from the system PATH.
  ➢ The **"exec.rt.matlab"** option does the equivalent for ".m" and ".p" modules, launching them with Matlab. The default is to launch "matlab" from the system PATH.
  ➢ In Windows, the **"exec.rt.winpython"** option stores the Python launcher to use for ".py" modules. This is required because in Linux and macOS the "hashbang" ("#!") line can be used to select the Python interpreter to use (different versions or installations), while in Windows this requires a helper program. See section 6.3.5 for details.
  ➢ All the above support an additional option named **"X_enabled"** where X is the name of the main interpreter option. Setting it to false will disable the built-in interpreter, allowing a user-defined interpreted to handle the same extension.
  ➢ The user-defined interpreters use options whose names start with **"exec.rt.user.X"**, where X is the extension that the interpreter is supposed to handle. Several sub-properties are required to define each interpreter, so it is recommended that the openSF GUI is used to create the configuration.
❑ General openSF settings:
  ➢ E2E-ICD module CLI version: the **"openSF.cliVersion"** option stores the E2E-ICD version that is used to call modules in simulations. Note that this applies to all modules and simulations run by the framework.
  ➢ User roles: the **"openSF.appModesUnlocked"** option allows or forbids the user role selection, while the **"openSF.appMode"** option stores the active mode/role. See section 3.2.2 for an explanation of user roles.
  ➢ Maximum simultaneous modules: the **"openSF.maxThreads"** option is the number of modules that are allowed to run simultaneously. The default value of 0 means that the value should be autodetected. For example, if the modules in the simulation are themselves written to use all CPU cores at once, it makes sense to prevent openSF from launching them in parallel by setting this option to a value of 1.

- ➢ Auto-update settings: the "**openSF.autoUpdateCheck**" option can be set to false to prevent the automatic update check on start. Furthermore, if the "**openSF.updateURL**" option is present but empty, even manual update checks will be disabled.

❑ Branding elements:
- ➢ Application name (executable):
  - ▪ Windows and Linux: the launcher binary can be renamed, as long as the ".ini" file that lives alongside it is also renamed.
  - ▪ macOS: the application bundle can be renamed, but it contains some symlinks (created by the installer) that will get broken. While they can be created again, it is difficult to ensure that the updated setup is correct so renaming the application bundle is not recommended.
- ➢ Application name (text): the "**openSF.appLongName**" and "**openSF.appShortName**" options can be configured to replace the names the program uses to refer to itself. These strings are shown in the main window title and in some menus and dialogs.
- ➢ Splash image: the image to be used as a splash is stored inside the JAR file of the openSF "platform" plugin. This file lives under <OPENSF_INSTDIR>, as "**plugins/platform_X.jar**" where X is a version number. The splash image is a bitmap stored at the root of the file.
  - ▪ To ensure the correct display of the image (i.e. avoid distortions and void areas) the replacement image must have the same **dimensions and DPI** of the original.
  - ▪ Note: in macOS, the JAR file is under "openSF.app/Contents/Eclipse/plugins".

❑ Other customizations:
- ➢ Embedded documentation: both openSF and ParameterEditor show a submenu under "Help" with useful documentation like this document. Each program examines the subfolders of "**<OPENSF_INSTDIR>/resources/documentation**". If a folder contains a file named ".openSF_doc" or ".PE_doc" (respectively for openSF and PE), any file inside it with the extension ".pdf" will appear in the menu.
- ➢ Removal of ParameterEditor: if the folder for the companion application cannot be found, the corresponding icon in openSF will simply be inactive (greyed out).
- ➢ Removal of the example simulator: if the file "example_simulator.zip" cannot be found under the "<OPENSF_INSTDIR>/resources/simulators", the option to initialize a database with the example simulator will be inactive (greyed out)

8. ANNEX D: TUTORIAL - CREATING AN E2E SIMULATION

I	A	E

This chapter will show user how to create and end-to-end simulation within openSF software. The simulation chain used is the one installed as validation scenario during openSF deployment.

This chapter is divided in the following sections:

- **Scenario Description**, showing the outline of the E2E simulation, logic entities, input and output identification, etc...

- **Module Development Guidelines** detailing the module development process aimed to be integrated in openSF.

- **Framework Structure Definition**, which details the steps that shall be taken in order to create a whole simulation scenario within openSF HMI. This section also gives some guidelines about the recommended folder structure for a simulation project that will be integrated within openSF.

- **Product Tools Specification**, including the definition of data exploitation tools.

This tutorial should be complemented by the information in the openSF Training course material [RD-3][RD-3] available on the openSF web page (https://eop-cfi.esa.int/index.php/openSF).

8.1. Scenario Description

The outline of a test simulation scenario is shown in Figure 8-1. The drawing of this diagram is the first step to integrate a simulation scenario within openSF.
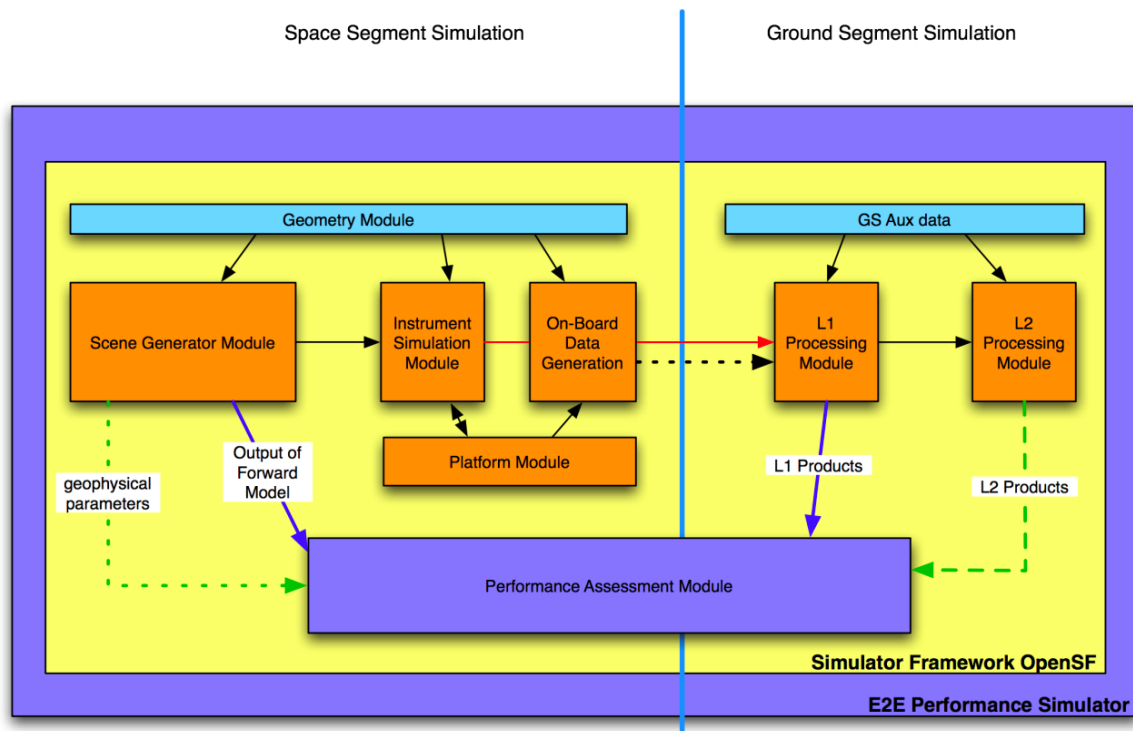
Figure 8-1: Outline of a test simulation scenario

After this point users shall identify and define the openSF entities that will be part of the simulation scenario. The entities that take part in this tutorial E2E simulation are listed in the following sections.

8.1.1. Descriptors – Input and Output Files

The definition of the descriptors (file sets) shall be done together with the module definition as input and files generated are the interfaces for simulation modules. This is described in [AD 1][AD 1].

The files showed in this section have been extracted from the validation test data set, available in the "/resources/simulators" folder within the openSF installation. Note that even though some are XML files, they are input/output files and *not* configuration files as defined in [AD-E2E], and therefore they do not use the same format.

- **Input_Geometry,** input used for the Geometry computation module.
  - *ref_orbit.oem*, the reference orbit in CCSDS Orbit Ephemeris Message (OEM) text format
  - *ref_orbit.aem*, the corresponding attitude, in CCSDS Attitude Ephemeris Message (AEM) format
- **Product_Geometry,** files generated by the Geometry computation module.
  - *real_orbit.oem* and *.aem*, the simulated "real world" equivalent of the reference orbit/attitude
- **Input_Scene,** input used for the Scene Generator module.
  - *real_orbit.*oem, and *.aem*, from the Geometry module output
  - *Geophysical.xml*, atmospheric parameters
- **Product_Scene,** output from the Scene Generator module.
  - *TOA_Stimuli.csv*
- **Input_Instrument,** compound of the two outputs from Geometry and Scene Generator
- **Product_Instrument,** file generated by the Instrument Simulation module.
  - *Instrument_Meas.nc*, instrument measurement data in a netCDF container
  - *Instrument_AD.xml*, ancillary data from the instrument
- **Product_Platform,** file generated by the Platform module.
  - *meas_orbit.oem* and *.aem*, orbit and attitude as measured by the simulated onboard NAV/AOCS
  - *Platform_AD.xml*, ancillary data from the platform
- **Input_OBDGM,** compound of the two outputs from the Instrument and Platform modules
- **Product_OBDGM,** file generated by the On-Board Data Generation module.
  - *NAVATT.bin*, the navigation&attitude data as a stream of CCSDS space packets
  - *Instrument_Space_Packets.bin*, the instrument measurement and ancillary data in the same format
  - *Platform_Space_Packets.bin*, the platform ancillary data in the same format
- **Product_L1Proc,** descriptor that represents the level 1 simulated product.
  - *L1.xml*
  - *rest_orbit.oem* and *.aem*, the restituted orbit and attitude
- **Input_L2,** descriptor that takes only part of the L1 output.
  - *L1.xml*
- **Product_L2,** descriptor that represents the level 2 simulated product.
  - *L2.xml*

8.1.2. Modules

This is the list of modules identified within the test simulation chain. Each module has associated an input and output description to allow a proper orchestration of the simulation scenario. Other module configuration items such as executable file and XML configuration file are described in [AD 1].

- Geometry
    - o Input descriptor: **Input_Geometry**
    - o Output Descriptor: **Product_Geometry**
- SceneGenerator
    - o Input descriptor: **Input_Scene**
    - o Output Descriptor: **Product_Scene**
- InstrumentSimulator
    - o Input descriptor: **Input_Instrument**
    - o Output Descriptor: **Product_Instrument**
- Platform
    - o Input descriptor: **Product_Geometry**
    - o Output Descriptor: **Product_Platform**
- OnBoardDataGen
    - o Input descriptor: **Product_OBDGM**
    - o Output Descriptor: **Product_OBDGM**
- L1Processor
    - o Input descriptor: **Product_OBDGM**
    - o Output Descriptor: **Product_L1Proc**
- L2Retrieval
    - o Input descriptor: **Input_L2**
    - o Output Descriptor: **Product_L2**

8.2. Framework Structure Definition

This task consists in defining all the entities specified in section 8.1 into the openSF HMI. This can be performed following the instructions specified in the openSF reference manual, chapter 4 of this document. Furthermore, a step by step example of the creation of this simulation in the openSF HMI can be found in the openSF Training course material [RD-3] available on the openSF web page (https://eop-cfi.esa.int/index.php/openSF).

8.2.1. Folder Structure Guidelines

This section gives some tips and recommendations about the folder structure within a simulation project that is integrated into openSF. This section is aimed at easing the integration process. As mentioned, the following instructions are not mandatory as users can choose whatever structure they prefer.

Simulation Project Structure:

- *modules* folder where all files regarding to the simulation algorithms are stored including executable, configuration and input files
    - o *src* for modules source code
        - ▪ *docker* for an example Docker-based module and wrapper script.
    - o *bin* for modules binaries
    - o *lib* for the libraries (example libProducts.dll for input output management that can be common to all modules within the simulation chain)
    - o *cots* folder for storing third party applications and libraries used within the modules
    - o *data*
        - ▪ *conf* for global and module configuration files

- ▪ *input* for files used as input
- ▪ *ref* for large data files used by modules and that should not be copied to each simulation folder. See section 6.4.
- *other_examples* folder
  - o *multiexec* for the example of exported multi-execution settings.
  - o *visibility_* files for the example of exported visibility settings.
- *tools* folder where store source code and related files for project specific tools (ex: end-to-end comparator)
  - o *bin* for tools binary files
  - o *lib* for tools library files
  - o *src* for tools source files
- *doc* folder where useful documentation of the project can be located

openSF team recommends storing all the data regarding to the project in folders using E2E_HOME as root directory or a subfolder of it. Example: */home/tester/openSF/E2Etutorial/* being */home/tester/openSF/* the E2E_HOME folder. This will help in the framework integration process and in possible future distribution of the simulator, as relative paths can be used.

8.3. Product Tools Specification

The definition of product tool is detailed in section 4.3.5; it is recommended that users take a look to this section before going on reading.

8.3.1. Simulation Products Exploitation

Following the mechanism described in section 4.3.5, openSF users are able to plug-in any product tool in order to visualize, post-process or archive the simulation products.

The selection of this product tools depends on the type of simulation products (definitively files) users want to analyse. A list of popular product tools used in openSF related simulation projects is shown in section 4.3.5.5.

In case of the test simulation scenario where all product files are XML the tool associated can be the user-preferred text editor (Notepad, Gedit, Emacs etc...).

8.3.2. Closing the Loop in an E2E Simulation

Usually the target of performing and E2E simulation is to validate the output of a set of algorithms comparing the input and the output of the simulation. Other objectives can be the sensitivity and stability analysis of a full processing chain over a set of simulation parameters.

In any of the mentioned cases it is necessary to perform a comparison between two points of the simulation chain in order to analyse the results. This connection **closes the loop** within a simulation scenario.

In order to close the loop in openSF, users can follow the following strategies depending on the simulation scenario.

- Development of a product specific processing tool.
- Development of a new module and insert it into the simulation chain as a new processing stage.
- Use of a cots comparison tool. This mechanism is recommended when the product format can be compared directly without any pre-processing step.

For the tutorial scenario possible points to close the loop are the Scene input (*Scene.xml*) and the L1b or L2 product (*L1b.xml* or *L2.xml*). This action would require the development of a product specific tool that performs a simple processing of the Scene input in order to compare with the L1b or L2 products. This situation can be also solved including a new module in the simulator where this comparison is performed.

![INDRA logo]

openSF

System User Manual

OPENSF-DMS-TEC-SUM01

4.9

18/06/2025

9. ANNEX E: INSTRUCTIONS TO BUILD THE FRAMEWORK

| I | A | E |

This annex explains how the openSF framework is built. This section is oriented only for developers that need to build openSF from sources due to project specific customizations.

The openSF development team recommends the use of Eclipse IDE as it is the platform used for developing the framework. It is just a recommendation, as the platform uses Maven as a build system and thus can be built directly from the command line, or with another Java IDEs.

9.1. Pre-requisites to Build the Framework

- openSF source files
- Java Development Kit, version 17 or later
- Apache Maven tool, version 3.8.6 or later
- If installer packages are to be generated, install4j version 11.x with a valid license.

9.2. How to Build the openSF Platform

Apache Maven is a Java-based build system that uses Project Object Model (POM) files to orchestrate compilation and packaging of applications. Maven can automatically pull dependencies from the Internet, a functionality that is used to download the Eclipse RCP Java files and native launchers.

Due to the above, building openSF requires a connection to the Internet, at least the first time that the build is attempted. This is necessary for Maven to download its own plugins, including the Tycho system that builds Eclipse RCP applications, and the Eclipse runtime files.

Once unpacked, the openSF tree contains the following relevant files and folders:

❑ build.sh/.cmd: support script performing most of the Maven build steps for *nix/Windows.

❑ openSF/pom.xml: Maven project to run for the openSF build.

❑ openSF/openSF.build/generate-installers.py: support script that gathers the files for each platform into the right structure and calls install4j to generate the installer packages.

❑ openSF/openSF.build/installers: folder containing install4j project files in order to generate installation packages for openSF (see §9.3). Also contains a folder to place the external packages to be bundled in, like documentation files, example modules, ParameterEditor, etc.

The output of this step is a series of "Eclipse packaged product" ZIP files, one for each platform, available at the openSF.product/target/products/ folder. Note that a single build in one machine generates the files for all platforms[14], since the platform-dependent components are downloaded from Eclipse and do not need to be built.

These files are not full installations of openSF: they are only one of the multiple components that are needed to build a fully-functional installer package, as described in section 9.3. However, they *can* be used as-is during development and testing, if they are unpacked on top of an *existing* openSF installation, they will in effect "upgrade" that installation.

9.2.1. Simplified procedure

Using the support script provided, it is easy to build the openSF platform files

```
(Linux/macOS) $ ./build.sh
```

---

[14] However, if building on Windows, the fact that the Windows file system does not save the "executable" bit will mean that the launchers inside the generated Linux and macOS product files will require a later chmod +x.

```
(Windows) > .\build.cmd
```

The support script runs the two-step build process outlined in the following simulation, accepting two environment variables that change its behaviour:

- ❑ MVN: path to the Maven executable, defaults to "mvn", so Maven is expected to be available in the system PATH.

- ❑ SKIP_TESTS: if defined to "1", skips the phase in which the openSF unit tests are run (the Maven target used is "package" instead of the next step "verify").

9.2.2. Detailed procedure

Any build configuration changes before the process can be performed on the various files that define the project settings. For example, the file "common.libs/releng/pom.xml" defines the system platforms to build for; the file "openSF/platform/plugin.properties" configures some strings in the program, etc.

In order to build the platform, OSFI-Java must be available in the local Maven repository. The latest pre-built version may be downloaded from ESA[15], or it can be built manually from sources (see [RD 1] for instructions). After this dependency is met, the main project can be built:

```
$ mvn -f openSF clean verify
```

The "verify" target builds the product and runs unit tests. If such tests fail, the build stops immediately and the product files are not generated. Some options that may be passed are:

- ❑ The -T ("threads") option to build components in parallel. A value such as "-T2C" (twice the number of cores) is reasonable, but the build may become brittle. If the build fails, consider removing this option first.

- ❑ Passing the -fae ("fail at end") flag to Maven. In this case, the test failure will be noted and the build will be reported as "failed" but the product files will still be generated if possible.

- ❑ Replacing the "verify" target with "package", which skips the execution of the unit tests.

- ❑ Adding the "site" target, which produces code quality reports using the PMD and Checkstyle tools.

9.3. How to Build the Installer Packages

As mentioned before, the output of the last step is a series of ZIP files in the Maven-generated target folder under openSF.product. They contain the archived installations of the RCP products for each platform, which can be used during development as mentioned in §9.2. However, a working openSF installation is made up of several components, which if missing will make the system work partially, or not at all. Those components are:

- ❑ Documentation files, the set of PDFs shown in the help menu. Without these files, no help files will be shown at all in the HMI.

- ❑ ParameterEditor, which is built separately as another RCP archived product, but with the same procedure described for openSF (but passing "ParameterEditor" instead of "openSF" in the -f argument to Maven). If this platform-specific component missing, openSF will not be able to launch PE.

- ❑ Example modules and database, which is a set of dummy modules, configuration and input files, along with a database with simulations including them. Since they include C++ modules along with Java and Python examples, they are platform specific, built using CMake and require linking against the OSFI library. If missing, only the "empty" database template file is available in the HMI, but no openSF functionality is lost.

- ❑ Template openSF.properties file, placed by the installer in the installation folder after some variables have been replaced. Without this file, openSF might not start.

In order to generate the installer, the openSF.build/generate-installers.py script can be used. It requires the other components to be present at openSF.build/installers/external. The user can place them there manually, or, if these outputs are uploaded to some internal Maven repository (e.g. as part of some continuous integration build

---

[15] See https://eop-cfi.esa.int/index.php/opensf/download-installation-packages

system), the dependencies.pom and gather-components.py files in that same folder are designed to download them from such a repository automatically. In either case, the layout should look like Figure 9-1.

Finally, if install4j is available, running the installer generation script will create files like those displayed in Figure 9-2. The script accepts settings through several environment variables including the product version and installer signing capabilities; see the script itself for details.
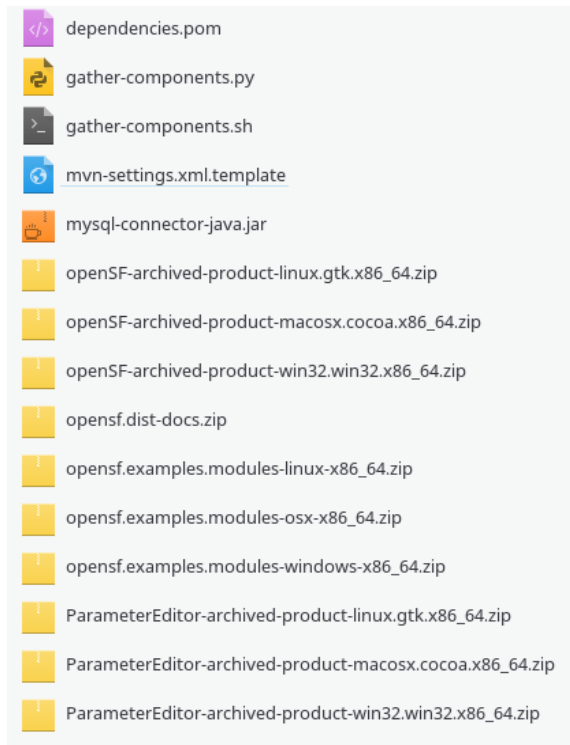
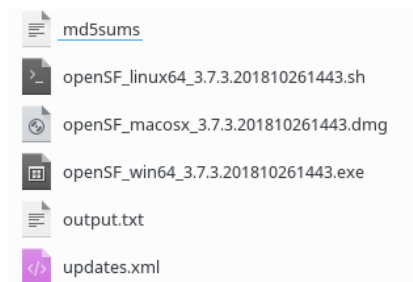

Figure 9-1: External components

Figure 9-2: Generated installers for a development build

10. ANNEX F: USING DOCKER IN OPENSF SIMULATIONS

| I | A | E |

The ability to run applications in virtual containers enables developers to easily integrate heterogenous environments in which E2E simulators are nowadays expected to work. The use of Docker promotes also the automated deployment of the execution environment and favours simpler and continuous testing procedures. This section describes how a simulation module can be setup inside a container and how to integrate it in an openSF E2E simulation.

It is highly recommended that each simulation module is deployed inside a separate Docker container, with OpenSF used to orchestrate the multiple modules that make part of the simulation. Notice that the goal of using containers, in the scope of E2E development, is to allow the integration of modules developed in heterogenous OS. This facility should not be used to favour the creation of deployment schemes that create unnecessary coupling between components.

The deployment of OpenSF inside a container is considered highly inappropriate.

10.1. Concepts and Requirements

When running a module inside a container, it is by default isolated from the host environment in multiple ways which depend on the system and its configuration. For example, Docker isolates the host filesystem and, depending on the daemon configuration, it may also provide a separate user namespace; that is, have different users than the host.

In order for openSF to run a module distributed as a container image, the following points must be taken into consideration:

1. **Dependencies**: all dependencies of the module must be included in the container image. For example, this may include dynamic libraries, OSFI and other packages.
2. **Paths mapping**: from the point of view of openSF, the container-based module must respect the E2E-ICD. This means that the module will be passed some arguments and environment variables which represent paths into the *host* filesystem, while the container internally sees a separate filesystem. Thus, some paths from the host need to be mapped into the container.
3. **File access**: the module needs read and write access to the simulation folder (the path represented by the E2E_EXECUTION_HOME variable), in order to read the configuration/input files and to generate the output files as required. Furthermore, read-only access to other paths may also be needed.
4. **Permissions of output files**: the output files created by the module need to be themselves accessible by the (host environment) user running openSF, so they can be opened by any further modules in the simulation, or by the user when the simulation completes.

In general, point 1 is taken care of at the time of image definition (Dockerfile) and build. For point 2, the module needs some external support program or script that presents an E2E-ICD compatible interface to openSF, while launching the container with an appropriate configuration to map the necessary host paths into the container environment.

In case the simulation module needs to rely on data in other paths (either directly or via filesystem symlinks), beware that extra caution is required. The path mappings defined when running the Docker container need to ensure that all necessary data is correctly available for the module.

Points 3 and 4 present a more complex problem that depends on the setup of the container system. For example, the default settings for Docker on Linux share the user IDs between the host and the container, but security-oriented deployments may use that separate user namespaces that appear as different numbers inside and outside the container.

If the container process can be made to run using the same (external) user as openSF, then the issue is likely moot. Otherwise, permissions and/or ACLs need to be set appropriately on the simulations folder to allow the proper interoperation of files.

10.2. Example

Considering that an openSF simulation module is ready for deployment, in order to run it inside a Docker container the following steps will be considered:

1- Create Docker image
2- Invoke Module in Docker container
3- Setup Module in openSF

The example provided as part of openSF assumes a Linux host and Docker containers, and that the Docker daemon is *not* configured to remap the user namespace. Refer to the contents of folder `<OPENSF_INSTDIR>/modules/src/docker` for the files described in the following sections, which provide details of how these steps allow to set up the *Simple* simulation module.

Please note that other container environments are not covered by this example. In particular, macOS/Windows Docker is not considered, even when running Linux containers. Neither are Windows containers on Windows Docker, other Linux container runtimes such as Podman, etc. While it is possible that such systems may be able to run a module in openSF by carefully applying the four points in section 10.1, they are not the main focus of this section.

10.2.1. Create Docker Image

A Docker image is created by following the recipe described in a Dockerfile. Typically, images take an existing base image (e.g. Ubuntu 20.04) and build on top of it by installing only what is essential to run the application, which in our case the simulation module.

```
1   FROM ubuntu:20.04
2
3   ENV DEBIAN_FRONTEND noninteractive
4
5   # Make basic tooling available
6
7   RUN apt-get update \
8        && apt-get install -y --no-install-recommends \
9             software-properties-common \
10            sudo \
11            gosu \
12            python-is-python3 \
13       && rm -rf /var/lib/apt/lists/*
14
15  # Install the openSF (Simple) Module. Other dependencies (e.g. OSFI) would go here too
16
17  COPY simple.py /opt/simple/simple.py
18  RUN chmod +rx /opt/simple/simple.py
19
20  ENV PATH="/opt/simple:${PATH}"
21
22  # Setup a generic entry point
23
24  COPY entrypoint.sh /usr/local/bin/entrypoint.sh
25  RUN chmod +x /usr/local/bin/entrypoint.sh
26  ENTRYPOINT ["/usr/local/bin/entrypoint.sh"]
```

Figure 10-1: Simple Dockerfile

The *Simple* Dockerfile, presented in Figure 10-1, is based on Ubuntu 20.04 (line 1).

The Dockerfile then installs the basic tools necessary to deploy and run the *Simple* simulation module (lines 7-14). Notice that in the example, auxiliary tools such as sudo/gosu are installed to facilitate the execution of the simulation module, and specifically to allow running the simulation module with root and non-root users. Depending on the use case, the simulation module developer might want to install the build toolchain and allow building/deploying the module from sources.

The deployment of the simulation module is handled in lines 17-18. Considering that *Simple* is a single Python script file, the deployment just copies the file to the expected installation location and ensures that it is executable. Since this example comes with openSF itself, it will re-use the OSFI-Python library that is distributed with the openSF examples. However, normal modules must include their own dependencies, including OSFI, since they should run on multiple versions of openSF.

![INDRA logo] ✕ INDRA

openSF

System User Manual

OPENSF-DMS-TEC-SUM01

4.9

18/06/2025

In case it is necessary, the Dockerfile also allows to customize the execution environment (see line 20, where the PATH environment variable is adjusted to allow finding `simple.py`).

The deployment step in the Dockerfile should be considered as a working specification of how to deploy the simulation module, either base on a distribution package or on a sources package. In this sense, it should be a dynamic reflection of what is described in the module's user manual.

The setup finishes by provisioning and configuring an entry point to be executed when the docker container is run. This file is a small auxiliary script that runs inside the container, and customizes the environment based on parameters passed when running the container (e.g. adjusts the user running the simulator inside the container).

Assuming that the environment variable OPENSF_HOME is defined to <OPENSF_INSTDIR>, extract the example simulator and then to build the Docker image `opensf:simple` execute the following commands:

```
$ cd $OPENSF_HOME/modules/src/docker/simple
$ docker build -t opensf:simple .
```

10.2.2. Invoke Module in Docker container

Even though running inside a Docker container, the simulator module must be able to access configuration and input files and to generate output files at locations designated by openSF. The simulator module must also be invokable using the standard CLI parameters (as described in [AD-E2E]). This can be achieved via a *Simple* adapter script that maps all necessary folders inside the Docker container, and forwards all the options to the module.

```
 1   #!/usr/bin/env bash
 2
 3   IMAGE=opensf:simple
 4
 5   docker run --rm -t \
 6       -e RUN_AS_USER="$(id -u)" \
 7       -e RUN_WITH_USERNAME="${USER}" \
 8       -e E2E_HOME="${E2E_HOME}" \
 9       -e E2E_EXECUTION_HOME="${E2E_EXECUTION_HOME}" \
10       -v "${E2E_HOME}:${E2E_HOME}" \
11       -v "${E2E_EXECUTION_HOME}:${E2E_EXECUTION_HOME}" \
12       "${IMAGE}" \
13       simple.py "$@"
14
```

Figure 10-2: Simple adapter

The *Simple* adapter script (shown in Figure 10-2) creates a temporary container to run the simulation module `simple.py` inside. The script calls the `docker run` command on line 5, with option `--rm` to dispose of the container one the simulation module is finished.

The options on lines 6-7 provide environment variables for the entry point script, customizing the user that runs the simulator inside the container – this is important to correctly match file ownership/permissions inside and outside of the container as stated in points 3 and 4 of 10.1.

The relevant environment variables (E2E_HOME and E2E_EXECUTION_HOME) are made available to the simulator based on the options in lines 8-10. Typically OPENSF_HOME is not used by the simulation module and thus is not necessary, but in the current example it is used to locate and provision the OSFI Python library.

In case the simulation depends on other environment variables, made available via openSF preferences, these must be added as further options in the adapter script.

To allow reading configuration and inputs and writing outputs, the options in lines 11-13 mount the relevant folders inside the container at the locations they are expected. This means that even though running inside the container, the simulation module will find the files/folders and no path translation is actually needed.

The Docker image to be used is determined by line 14 and then line 15 specifies the command to be executed – i.e. the simulator module called with the parameters exactly as defined by openSF.

Assuming that the OPENSF_HOME variable is defined and points to <OPENSF_INSTDIR>, to run *Simple* inside the container run the adapter script as follows:

```
$ cd $OPENSF_HOME/modules/src/docker
```

```
$ ./simple.sh -g gcf.xml -l lcf.xml
Info    | OpenSF (Simple) Module v1.0 is inside docker
Info    | Inside docker, using configuration file: gcf.xml
Info    | Inside docker, using configuration file: lcf.xml
```

10.2.3. Setup Module in OpenSF

In practice, with the Docker image created and an invokable script to run the Simple simulation module ready, there are no differences to set up any other simulation module. From an openSF perspective, the invokable script will act as a regular/native module.

For the *Simple* simulation module, do as follows:

1) Create the appropriate input and output descriptors
2) Create the module, defining test/data/docker/simple.sh as the executable
3) Add the new module to an existing simulation or create a new simulation and use it

After these three steps are concluded, simply run the simulation. Figure 10-3 shows the execution log of a simulation where the provided *Simple* example was configured with input descriptor Product_L1b, output description OutputGeneric, and included in E2E_test simulation.
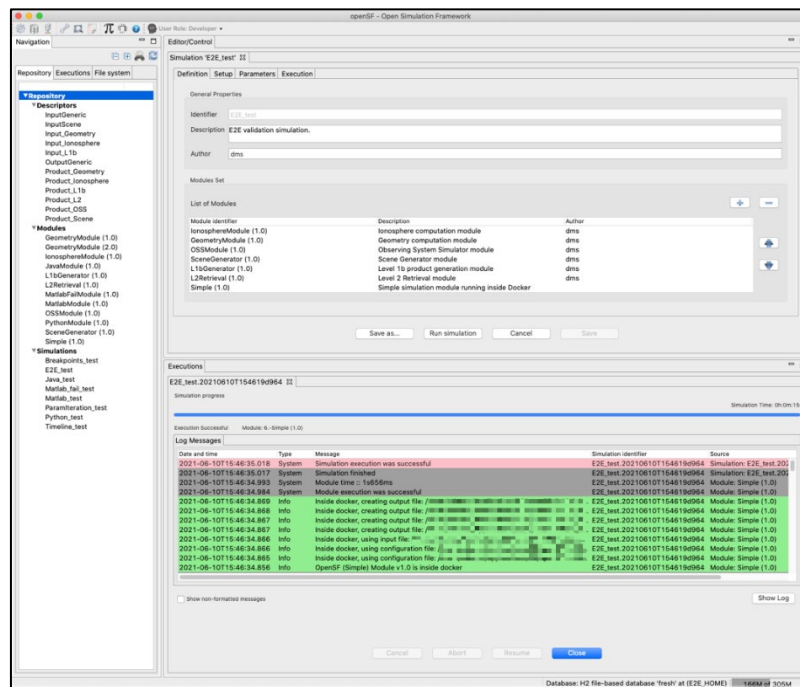

Figure 10-3: Example container module configured and running in openSF

END OF DOCUMENT