Integration Libraries for the open Simulation Framework

OSFI

DEVELOPER'S MANUAL

Code : OPENSF-DMS-OSFI-DM-013
Issue : 1.14
Date : 15/12/2017

	Name	Function	Signature
Prepared by	Enrique del Pozo Jose Julio Ramos Alberto Monescillo Jose Luis Garcia Rui Mestre Javier Martín Gonzalo Vicario	Project Engineers	
Reviewed by	Federico Letterio	Project Manager	
Approved by	Antonio Gutierrez	Project Manager	
Signatures and approvals on original			

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

This page intentionally left blank

Document Information

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

Internal Distribution		
Name	Unit	Copies
Antonio Gutierrez	Head of the Ground Segment Business Unit	1
Internal Confidentiality Level (DMS-COV-POL05)		
Unclassified ☑	Restricted ☐	Confidential ☐

External Distribution		
Name	Organisation	Copies
Michele Zundo Montserrat Pinõl Sole Maurizio De Bartolomei	ESA/ESTEC	1

Archiving	
Word Processor:	MS Word 2010
File Name:	OPENSF-DMS-OSFI-DM-013-114.doc

Document Status Log

Issue	Change description	Date	Approved
1.0	First issue of this document	15/12/08	
1.1	ANSI C programming language support	27/04/09	
1.2	Quality indicators functionality added	26/05/09	
1.3	Mac Installation added	15/03/10	
1.4	OSFI library for F77, IDL and Matlab. Added sections describing the integration of models in F77, IDL and Matlab.	22/09/10	
1.5	Updated OSFI library for Matlab. ❑ Added configuration for IDL. ❑ Section 4.1 now points to openSF ADD, OSFI section. ❑ Added new section 0 with OSFI additional features.	12/11/2010	
1.6	Updated OSFI library for Python.	02/04/2014	
1.7	Updated after review comments from ESA: implemented RIDS OPENSF_v3.2_RID_03, OPENSF_v3.2_RID_04 and OPENSF_v3.2_RID_05 by updating section 4.2.7.	30/04/2014	
1.8	Updated OSFI library for Java Update of AD/RD Reworded introduction	15/08/2016	
1.9	Updated OSFI available API (added existParameter method)	20/10/2016	
1.10	Updated after review comments from ESA: added Python requirements, added reference documentation (doxygen)	18/11/2016	
1.11	Updated due to support to Python 3.X	16/06/2017	
1.12	Update version of compilers used in openSF	06/07/2017	
1.13	New section defining the OSFI implementation of the E2E ICD	13/09/2017	
1.14	New build system based on CMake New Fortran interface and Foreign Function Interface Fortran 77 and IDL deprecated	15/12/2017	

Table of Contents

List of Tables

List of Figures

1. INTRODUCTION

The open Simulation Framework (openSF) relies on a well-defined set of interfaces [E2E-ICD] that the participating modules have to adhere to. The OSFI activity addressed the definition and development of a set of software libraries to ease the integration of modules into openSF system by providing a ready-made implementation of these interfaces.

Usage of OSFI libraries are therefore a key component to easily develop modules using openSF as orchestrating framework.

Terminology Note: starting with openSF 3.3 the recommended term to identify the orchestrated software components within an E2E simulation is **Module** instead of **Model**.

The text in this document has been amended accordingly however the name of software functions and variables still reflects the old naming convention.

1.1. Purpose

The objective of this document is to provide a detailed description and a development manual for the set of software libraries (OSFI) that can be used during the development and deployment of the modules within an E2E Mission Performance simulator

The intended readerships for this document are model developers and scientists that are in charge of integrate those models into the openSF.

This document is also useful to software engineers responsible of the testing stage.

1.2. Scope

This document shows a detailed description of the integration libraries and an API that can be used as a reference manual by model developers. It also includes a brief architecture description and some examples of use.

This document contains the following sections:

- An introduction (current section 1) for giving a quick overview of the project;
- A list of related documents to provide a documentary background (section 2)
- An introduction to the integration libraries, installation and linking instructions (section 3)
- A description of the architecture, the process logic and some examples of use. It also includes the coding guidelines (section 4)

1.3. Acronyms and Abbreviations

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

Acronym	Description
AD	Architectural Design Applicable Document
ADD	Architectural Design Document
API	Application Programming Interface
AR	Acceptance Review Analysis of Requirements
CFI	Customer Furnished Item
CLP	Command Line Parser
CM	Configuration Management Configuration Manager
CMP	Configuration Management Plan
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
DD	Detailed Design
DMS	DEIMOS Space
DRR	Document Review Record
ECP	Engineering Change Proposal
E-R	Entity Relationship
FFI	Foreign Function Interface
GUI	Graphical User Interface
HW	Hardware
I/F	Interface
I/O	Input/Output
ICD	Interface Control Document
ITT	Invitation To Tender
KOM	Kick-Off Meeting
MD	Managing Director
MMI	Man-Machine Interface
MoM	Minutes of Meeting
MR	Management Review
NCR	Non-Conformance Report
O/S	Operating System
PA	Product Assurance
PDR	Preliminary Design Review

Acronym	Description
PM	Progress Meeting Project Manager
PMP	Project Management Plan
QA	Quality Assurance
RD	Reference Document
RID	Review Item Discrepancy
SOW	Statement Of Work
SPR	Software Problem Report
SR	Software Requirements
SRD	Software Requirements Document
SRN	Software Release Note
SRS	Software Requirements Specification
STR	Software Test Report
SUM	System User Manual
SVS	Software Validation Specification
SW	Software
TBC	To Be Confirmed
TBD	To Be Defined / Decided
TER	Test Execution Record
TN	Technical Note
TP	Test Plan
TR	Test Report
TS	Technical Specification
UML	Unified Modelling Language
URD	User Requirements Document
V&V	Verification & Validation

2. RELATED DOCUMENTS

2.1. Applicable Documents

The following table specifies the applicable documents compliant with OSFI development.

Table 1: Applicable documents

Reference	Code	Title
[OSF-ICD]	openSF-DMS-ICD-001	OpenSF Interface Control Document Issue 3.0
[E2E-ICD]	PE-ID-ESA-GS-464	ESA Generic E2E ICD Issue 1.2.2

2.2. Reference Documents

The following table specifies the reference documents to be taken into account during module development.

Table 2: Reference documents

Reference	Code	Title
[OSF-SUM]	OPENSF-DMS-SUM-001	OpenSF System User Manual Issue 3.8
[OSF-ADD]	openSF-DMS-ADD-001	OpenSF Architecture Design Document Issue 2.2
[OSF-SRD]	openSF-DMS-SRD-001	OpenSF System Requirements Document Issue 3.2
[OSF-DOC]	http://eop-cfi.esa.int/Repo/PUBLIC/DOCUMENTATION/OPENSF/DOXYGEN/	OSFI Doxygen documentation

2.3. Standards

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

Table 3: Standards

Reference	Code	Title	Issue	Date
[XML]	(www.w3.org/TR/xml11/)	Extensible Markup Language (XML) 1.1	Second Edition	Sep 29 2006
[UML]	www.uml.org/#UML2.0)	Unified Model Language (UML)	2.1	Oct 6 2006
[BNF]	(see also en.wikipedia.org /wiki /Backus-Naur_form)	Algol-60 Reference Manual	5	1979

3. GETTING STARTED

3.1. Introduction

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

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

The open Simulation Framework (openSF) is a generic simulation framework aimed to cope with these major goals. It provides end-to-end simulation capabilities that allow assessment of the science and engineering goals with respect to the mission requirements and it is available for Linux and OSX.

This openSF framework allows the users to integrate and execute pieces of code, «modules» that form the building blocks of a simulation process.

To integrate an external module into the framework, the module needs to fulfil a series of interface requirements detailed in [OSF-ADD] and [OSF-ICD].

The Open Simulation Framework Integration Libraries (OSFI) version 3.4.0 will be used to ease the integration of modules into the open Simulation Framework version 3.7.1.

The Integration Libraries activity provides the module developer with a set of routines with a well-defined public interface hiding the implementation details. This set of routines is currently available in C++, ANSI C, Fortran, Fortran 77, IDL, Matlab, Python and Java (Fortran 77 and IDL are deprecated).

3.2. Conventions used in this Manual

This chapter lists all the conventions used throughout this Developer's Manual

3.2.1. $OSFI_HOME

All through the contents of this Developer Manual, a "variable" called $OSFI_HOME is exhaustively used as a placeholder. The variable value points to the root folder that contains the OSFI installation. Typically, this folder could be similar to this:

```
/home/user_name/OSFI
```

3.2.2. Data types

Every requested or given piece of data in OSFI is formatted in one of the following data types, as defined in [E2E-ICD]:

❑ *STRING*. A string of alphanumeric characters. While the ICD limits strings to 255 characters, OSFI places no a priori restriction on their size.

❑ *INTEGER*. Integer number (no decimal part) between -2^{31} and $2^{31}-1$. This matches the ranges of the C and Java data types int32_t and int, respectively. *FLOAT*. Decimal number with a range defined by the Java type double (IEEE-754 binary64 format)..

❑ *BOOLEAN*. TRUE of FALSE.

❑ *FILE*. The absolute (or $OSFI_HOME-relative) path and name of a file into the file system.

❑ *FOLDER*. The absolute (or $OSFI_HOME-relative) path and name of a folder into the file system.

Parameter elements may have compound types, such as ARRAY or MATRIX, as defined in the referred document. The element type of the compound will be one of the above.

3.3. Initial Requirements

The OSFI v3.4.0 system is prepared to run in a hardware and software platform with the following requirements. These must be fulfilled before installing the distribution.

3.3.1. Hardware requirements

OSFI v3.4.0 is compatible with the following architectures and operating systems:

❑ *Operating systems*: Linux, OSX

❑ *Architectures*: x86-64 (also known as AMD64 or Intel 64)

3.3.2. Software requirements

This is the list of suggested compilers for the sources.

Table 4: Suggested compilers for sources

Language	Compiler	Licensing	Distribution Site
Fortran	Intel Fortran compiler v 9.1	Several options. There is a free edition for the community	http://support.intel.com/support/performancetools/fortran/linux
Fortran	GNU Fortran Compiler v 4.9 or superior	GNU General Public License, GNU Lesser General Public License	http://gcc.gnu.org/fortran/
C/C++	GNU C/C++ compiler v4.9 or superior	GNU General Public License, GNU Lesser General Public License	http://gcc.gnu.org
Java	Oracle Java(TM) Runtime Environment, Standard	Oracle Binary Code License Agreement for the Java SE Platform Product	http://www.oracle.com/technetwork/java/index.html

	Edition 1.7 or superior		

Nevertheless, developers can use their favorite compilers in each case, as long as they support the relevant standards (C++11, C90, Fortran 2003, etc.).

Table 5 shows the system pre-requisites in order to build the OSFI libraries.

Table 5: System pre-requisites

Component	Purpose	License	Distribution Site
De-compressor	Extract files from release packaged in a compressed tarball	N/A	N/A
CMake 3.9 or higher	Build, test and pack the OSFI libraries	BSD 3-clause	Linux repository or https://cmake.org/

Table 6 shows a set of utilities that are recommended to build the OSFI libraries. If Xercesc is not installed in the system, the OSFI build system can be configured to download and build it.

Table 6: Recommended utilities

Component	Purpose	License	Distribution Site
Doxygen 1.8.13 or higher	Generate OSFI libraries documentation	GNU General Public License	Linux repository or http://www.stack.nl/~dimitri/doxygen/index.html
Google Test	Generate and execute C++, C and Fortran tests	BSD 3-clause	Linux repository or https://github.com/google/googletest
Xercesc 3.2.0 or higher	Parse XML files	Apache License 2.0	http://xerces.apache.org/

3.3.2.1. IDL

To execute modules in IDL with openSF it is necessary to have installed IDL software 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 eventually 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 ConFM module, which uses some internal classes of IDL that must be in the default path, because otherwise the application does not find them. This is

caused because IDL looks 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.

3.3.2.2. 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.

3.3.2.3. Python

There are two prerequisites to execute a Python module from openSF:

1. Python correctly installed
2. Environment variable PYTHONPATH configured to point to the necessary libraries (e.g. with PYTHONPATH=$OSFI_HOME/include/Python).

Python interpreter could be found in the public repositories for the most popular Linux distributions, in Yum system for SUSE/RedHat or Synaptic in Debian/Ubuntu. For further details about installation please visit the Python Project webpage (https://www.python.org)

The OSFI Python libraries are developed to be compliant with both Python 2.X and Python 3.X interpreter. The recommended version is Python 2.7.

3.4. Installation

OSFI is distributed as source package, with the necessary sources in every language supported, for including and compiling with other sources. Figure 1 shows a high-level view of the contents of the OSFI distribution:

- The folder include contains the header files of the library
- The folder releng (release engineering) contains CMake configuration files
- The folder src contains the source files of the library
- The folder test contains a set of unit and integration tests that ensure the proper performance of the library

In addition, the distribution includes the main CMake make file, the license, the release notes and the version information file.

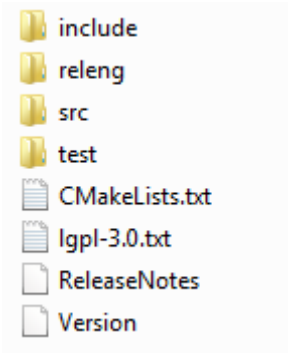

Figure 1: OSFI distribution

3.4.1. Build Instructions

First, extract the integration libraries into the desired location and enter it:

```
$ tar -xvzf OSFI-<version>-src.tar.gz
$ cd OSFI
```

Next, create a folder where the products of the building process will be generated (e.g. build) and enter it:

```
$ mkdir build
$ cd build
```

The command to configure the make files have a set of optional arguments that must be reviewed. First of all, the default behavior of the build system is to configure the make files for all the languages supported by the OSFI libraries. Nevertheless, the user can deactivate any language by setting the corresponding boolean flag to OFF:

- OSFI_CXX
- OSFI_C
- OSFI_F77
- OSFI_Fortran
- OSFI_Java
- OSFI_IDL
- OSFI_Matlab
- OSFI_Python

It shall be remarked that the OSFI implementations of C and Fortran (the two of them) depend on the C++ implementation. Thus, the C++ language cannot be deactivated if one of the aforementioned implementations is active.

In addition, the OSFI libraries depend on Xerces v3.2.0. The default behavior of the build system is to look for the library in the user's system, but two optional arguments can be used to change the behavior:

- XercesC_DIR: it forces CMake to look for the Xerces library in the directory provided.
- BUILD_XERCES: if this boolean flag is set to true, CMake will download and build Xerces-c 3.2.0 in the directory *xerces/ExternalProject* created in the build folder.

If the boolean optional argument BUILD_SHARED_LIBS (which default value is OFF) is set to ON, the build process generates shared libraries. If not, static libraries are created.

If the boolean optional argument BUILD_DOC (which default value is ON) is set to OFF, the doxygen documentation of the OSFI libraries will not be created. It shall be remarked that doxygen must be installed in order to generate it.

Finally, the boolean argument BUILD_TESTING (which default value is ON) can be set to OFF in order to skip the test building process. Nevertheless, Google Test must be installed in order to build the tests.

The following example shows how to configure the OSFI make files from the build folder created inside the OSFI directory to generate the static libraries. It can be seen that the IDL, Matlab, Java, Fortran 77 and Python languages are disabled and that the Xerces library is downloaded and built. It shall be remarked that the optional arguments are provided starting with "-D".

```
$ cmake -DBUILD_XERCES=ON -DOSFI_IDL=OFF -DOSFI_Matlab=OFF -
DOSFI_Java=OFF -DOSFI_F77=OFF -DOSFI_Python=OFF ../
```

Once the make files are configured, the selected OSFI libraries are built with the following command, executed from the build directory:

```
$ cmake --build .
```

The OSFI unit tests can be launched with the following command executed from the build directory:

```
$ ctest
```

If the test execution has been successful, the last step is to package the OSFI build products with the following command:

```
$ cpack
```

If the installation has been successful, the package folder structure should be as follows:

- include: header files
- lib: dynamic or static libraries of OSFI. In addition, the folder cmake/OSFI/ contains the CMake configuration files.
- share: documentation of the libraries API in html format. This folder is not be available if the documentation is not created.

The module developer has the responsibility to include in the package the Xerces library used during the build process. If the library was built with OSFI, the generated products are located in the build directory in the folder xerces/ExternalProject/Install.

3.4.2. Modules Distribution

It is under the module developer responsibility to distribute it with the OSFI libraries and other dependencies of the module, ensuring that it will execute properly in the environment of the E2E Mission Performance simulator.

For simulators with few modules, it is recommended to compile them statically with the static version of the OSFI libraries, in order to guarantee its execution in any environment. However, for simulators with a large number of models, it is more efficient in terms of simulator size to build the modules with the dynamic version of OSFI libraries, although the compilation process becomes more critical in order to avoid potential conflicts during modules execution.

3.4.3. Modules Building Instructions

This section provides instructions for building the modules using CMake, the suggested build system. It assumes that OSFI and Xerces are already built.

In order to provide the Xerces and OSFI libraries to the building system, the user should use the CMake command *find_package*. Firstly, the developer shall add the Xercesc package with the commands shown below. It can be seen that function *find_package* allows the user to input the location of the library to be added. The package Threads refers to the threading library of the system and it is usually needed by Xercesc.

```
find_package(Threads REQUIRED)
find_package(XercesC REQUIRED CONFIG HINTS "${XercesC_DIR}")
```

The OSFI libraries are added using the same command. It can be seen that with the option *COMPONENTS*, the developer can select the libraries needed in terms of the programming language. In the example below, the CXX and Java libraries are selected.

```
find_package(OSFI REQUIRED CONFIG COMPONENTS CXX Java HINTS
"${OSFI_HOME}")
```

After these commands, Xercesc and OSFI are available for the building process, which shall be performed with the proper CMake commands.

4. OPENSF INTEGRATION LIBRARIES

In this section, the following is given:

- An architectural overview, giving structural descriptions of the elements offered in the APIs (such as inheritance diagrams for C++ classes, etc).
- A complete set of examples of how to use the APIs and how to compile and execute them.

4.1. Architectural Overview

The Integration Libraries will serve as interface between the openSF component and the external module, as shown in Figure 2.

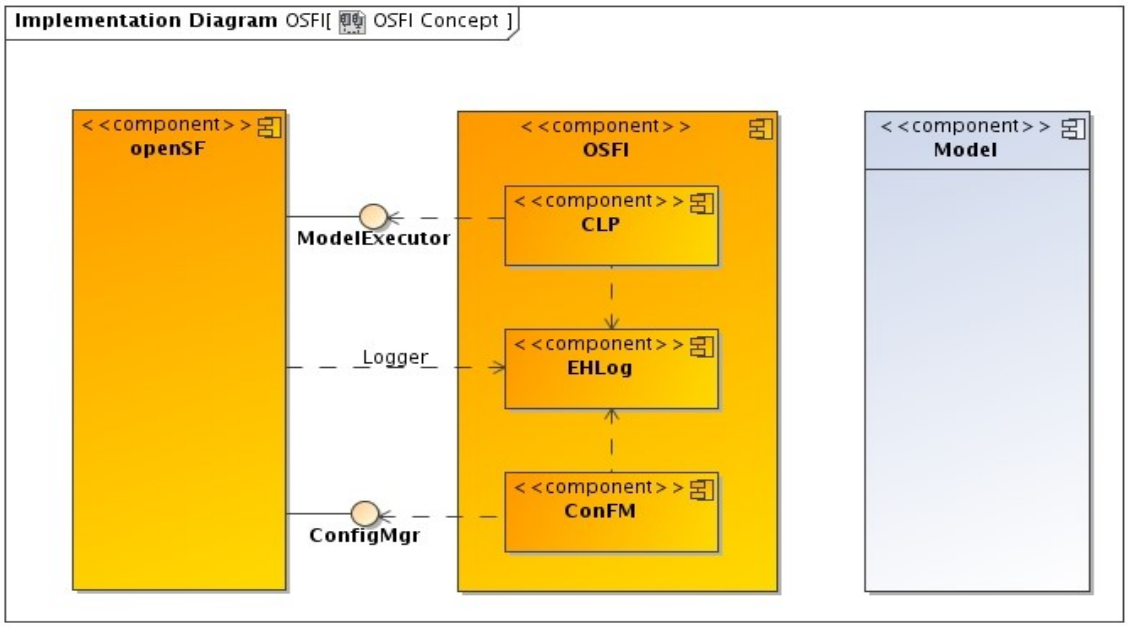

Figure 2: Relationship with openSF and modules

The package "Model" depends on the functionalities implemented in the "OSFI" package.

This package, in turn, depends on the functionalities provided by the "ConfigMgr" and "Model Executor" interfaces of the "openSF" package. This "Model Executor" is the responsible to provide the proper command line arguments for the module execution. The "ConfigMgr" is the module generating the XML configuration files.

There exists a tight integration between the "openSF" package and the "Integration Libraries" package because the former also needs the latter for reading the events raised and logged out from the module execution.

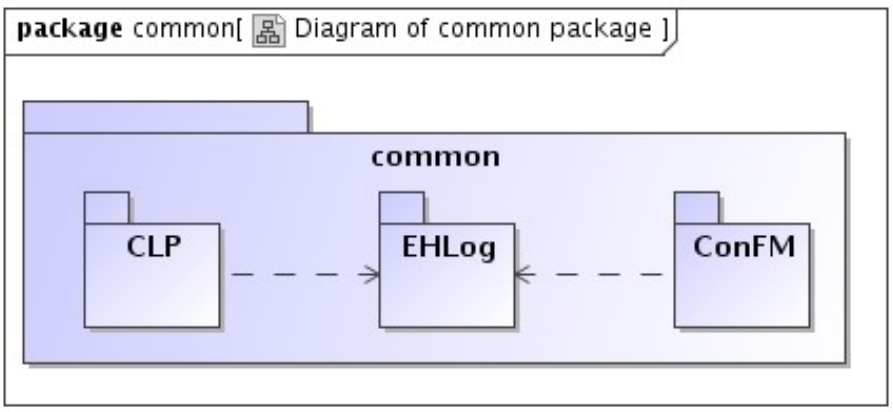

Figure 3 : OSFI common packages

This diagram shows how a system is split up into logical groupings by showing the dependencies among these groupings. As a package is typically thought of as a directory, package diagrams provide a logical hierarchical decomposition of a system.

It can be seen that both "CLP" and "ConFM" packages depend on the "EHLog" package because they are also able to raise certain events to be logged out.

Regarding the interaction between languages, the C++ implementation serves as a reference for function naming convection and availability to the module developer. The Fortran and C OSFI libraries use the C++ libraries by means of an intermediate bridging C++ library called FFI (Foreign Function Interface). This library implements an interface to the OSFI functions that can be called without C++ name mangling or other C++-specific features such as exception handling. It is intended as an intermediate layer to OSFI-C++ from other languages and should be used to extend OSFI capabilities to other languages that are not currently supported.

For a deeper analysis of OSFI libraries there is a section in [OSF-ADD] describing the architecture and the interaction with the different programming languages.

4.2. OSFI Common Packages

This section describes the functions provided by the OSFI common packages in terms of input, output and behavior.

4.2.1. CLP

According to section 2.1.2 of E2E-ICD, the module shall be invoked with the following command arguments separated by white spaces:

- Two configuration files separated by commas.
- Multiple (at least one) input files separated by commas.
- Multiple (at least one) output files separated by commas.

This package provides the following functionalities:

- Build command line parser: this function reads and parses the command line arguments.
  - Input: the command line arguments of the module. If the format does not match the description of 2.1.2 of E2E-ICD, an error will be raised. Possible format mismatches are:

    Number of configuration files is not 2.

    No input or output files provided.

    The name of a file contains a blank space.
  - Output: although it depends on the programming language, it returns a handle to access the command arguments.
- Get configuration files: it returns a list with the configuration files.
- Get input files: it returns a list with the input files.
- Get output files: it returns a list with the output files.

4.2.2. EHLog

According to section 2.2.4 and 2.2.5 of E2E-ICD this package shall provide functions to send information, warning, error, debug and progress messages. In addition, the current implementation of the OSFI libraries reveals that an additional function to indicate the end of the execution is implemented in all the languages. Thus, the EHLog package shall provide the following functions:

- Information: it sends an informative message raised by the module describing a harmless event.
- Warning: it sends a message of a non-fatal error or anomalous condition in data or during the processing that may cause a fatal error or affect the outputs in format or content. The execution should continue with no interruption.
- Error: it raises an error. For those programming languages able to throw exceptions, this strategy will be used.
- Debug: it sends an information message only if the debug mode is active.
- Progress: it sends numerical information on the amount of module execution performed.
- Finish execution: it shows a message and finishes the execution with a given error code.

In addition, for some languages (like C++ with streams or C with printf) the OSFI libraries provide some functionalities that allow the module developer to format more comfortably the messages.

4.2.3. ConFM

According to section 2.2.6.2 of E2E-ICD this package shall provide the functions that are needed to read the parameters that the users can define.

4.2.3.1. Parameter Reader

This function accepts as input the absolute (or relative to the shell environment variable E2E_HOME according to section 2.1.2 of E2E-ICD) pathname to the file where the parameters are defined and returns a handle to a reader (although the output depends on the programming language). It returns an error if the file does not exist or cannot be accessed.

Optionally, it accepts also the pathname of a schema file to validate the parameter file, returning an error if it does not comply with the format. The validation feature is not supported by all the programming languages.

4.2.3.2. Access Parameters

The parameters are identified by its complete name, which is obtained concatenating the name of the groups that contain the parameter (from the root to the most nested one) and the parameter name, all separated by dots.

The following functions are related with parameter access. If the file that contains them has not been validated previously, these functions may raise errors if the format of the parameters declared is not correct.

- Parameter exists: it checks if a parameter exists or not given its complete name.
  - Input: complete name of the parameter.
  - Output: returns true if the parameter exists, false otherwise.
- Get parameter: this functionality returns a parameter.
  - Input: complete name of the parameter.
  - Output: the parameter if it exists. Otherwise, the function raises an error specifying the complete name given as input.
- Get parameter reference: this functionality returns a reference to a parameter that will be valid until the instance of the parameter reader from which it was obtained is destroyed.
  - Input: complete name of the parameter.
  - Output: the handle to the parameter if it exists. Otherwise, the function raises an error specifying the complete name given as input.
- Get parameters of group: it returns a list with the parameters that belong to the input group.
  - Input: complete name of the group.
  - Output: list with the handles to the parameters that belong to the group. If the group does not contain any parameters (e.g. it contains another group), the list will

be empty. If the group does not exist, the function will raise an error specifying the complete name given as input.

- Get all the parameters: it returns a list with all the parameters declared in the file.
- Print parameters: it prints the information of the parameters declared in the file. It prints the attributes defined by the user (see section 2.2.6.2 of E2E-ICD) and its value.

4.2.3.3. Get Parameter Value

This set of functions is in charge of reading the value of the parameter. They shall support 6 different element types according to section 2.2.6.2 of E2E-ICD (integer, float, boolean, string, file and folders) and 4 levels of complexity (scalar, vector, matrix and array).

For those programming languages where the type of the output returned by a function must be known at compilation time (C++, C, Java, Fortran 77 and Fortran) 24 different functions will be available, covering each data combination of type and element types. However, for those languages where the type of the data is known at run time (Matlab, Python and IDL), these capabilities are encapsulated in a single generic function.

Depending on the element type and complexity level of the value to read, there are certain details that must be considered that are described in the following sections.

4.2.3.3.1. Access Scalar Values

This set of functions read the scalar value of a parameter. There is a dedicated function for each value type. If the function used does not match the type of the parameter, an error will be thrown reporting the name of the parameter, the type and the type expected by the function.

- Get integer value: it reads an integer number (without decimal part). If the value has a decimal part or is out of the acceptable range defined by the user (default minimum and maximum values are -2^{31} and $2^{31}-1$ according to section 2.2.6.2 of E2E-ICD) the function returns an error. The minimum and the maximum are acceptable values of the parameter.
- Get double value: it reads a positive or negative decimal number. The accepted notations are fixed point (3.141593) or exponential (3.141593e+00 or 3.141593E+00). For positive values the + sign at the beginning is optional. Numbers without decimal part are also accepted.

  If the value does not correspond to any of the given formats or is out of the range defined by the user (default range is between $\pm 1.7 \cdot 10^{308}$ according to section 2.2.6.2 of E2E-ICD), the function will return an error. The minimum and the maximum are acceptable values of the parameter. However, the decimal numbers within the interval $\pm\ 5.0 \cdot 10^{-324}$ (such as 0.0) should not be accepted according to the definition of float type given in section 2.2.6.2 of E2E-ICD.
- Get boolean value: the accepted values are TRUE or FALSE, raising an error otherwise.
- Get string value: according to section 2.2.6.2 of E2E-ICD, it reads a string of alphanumeric characters (from a to z including capital characters and numbers from 0 to 9) with a length not greater than 255. It will return an error if the size is greater that

the limit or an invalid character is detected. Nevertheless, a less restrictive definition is proposed without length limit and a wider set of valid characters, which comprises the ASCII ones except the single quote (') due to the array and matrix of strings declaration (see section 2.2.6.2 of E2E-ICD).

- Get file value: it reads an absolute (or session folder relative) pathname of a file in the file system. It has the same restrictions that the string value has.
- Get folder value: it reads an absolute (or session folder relative) pathname of a folder in the file system. It has the same restrictions that the string value has.

4.2.3.3.2. Access Arrays

This set of functions are used to read parameters of type array, i.e. generic arrays of elements (up to 3 dimensions), where each dimension may have distinct sizes. The elements of the array are blank-separated, throwing an error if other separators are used. The data types of the elements supported are string, file, folder (not included in section 2.2.6.2 of E2E-ICD), float, int and boolean.

The restrictions described in section 4.2.3.3.1 for the different types apply to the elements of the arrays. It shall be remarked that the elements of type string, file or folder must be enclosed in single quotes, raising an error if they are not, according to section 2.2.6.2 of E2E-ICD.

If the number of elements of the array does not match the dimension declared, an error will be raised. The functions available are:

- Get integer array.
- Get double array.
- Get boolean array.
- Get string array.
- Get file array.
- Get folder array.

4.2.3.3.3. Functions to Access Matrices

This set of functions are used to read parameters of type matrix, an array of arrays with predefined dimensions where the contained arrays must all use the same dimensions, throwing an error in case of mismatch between the dimensions declared and the number of elements.

The matrices are considered a special case of arrays, so the restrictions described in section 4.2.3.3.2 are applicable.

- Get integer matrix.
- Get double matrix.
- Get boolean matrix.
- Get string matrix.
- Get file matrix.

- Get folder matrix.

4.2.3.3.4. Functions to Access Vectors

Although vector is not defined in E2E-ICD, most of the OSFI implementations provide functions to read vectors. Thus, the following set is proposed to read them, defined as parameters of complex type arrays with a single dimension. Hence, the restrictions described in section 4.2.3.3.2 are applicable:

- Get integer vector.
- Get double vector.
- Get boolean vector.
- Get string vector.
- Get file vector.
- Get folder vector.

4.2.3.4. Get Parameter Attributes

This set of functions is used to read the attributes of a parameter.

- Get number of dimensions: it returns the number of dimensions of the parameter (i.e. 1 for scalars, 2 for matrices etc).
- Get dimensions: it returns the dimensions of the parameter. Depending on the complex type, the returned value is:
  - Scalar: returns always 1.
  - Matrix: returns two numbers, the first one refers to the columns and the second one to the rows (see section 2.2.6.2 of E2E-ICD). Nevertheless, the order is not intuitive (usually the first value refers to the rows and the second to the columns) and should be changed.
  - Array: it returns the maximum size of each dimension starting from the most nested one to be consistent with the matrix dimensions.
- Get name: it returns the name of the parameter.
- Get description: if the parameter has a description, it returns it. Otherwise, it returns an error.
- Get units: if the parameter has units, it returns it. Otherwise, it returns an empty string.
- Get maximum: this attribute is only applicable to float and integer types, so the function returns an error when the parameter has a different type. This function returns the maximum value of the parameter as string, or an empty string if it has not been declared.
- Get minimum: this attribute is only applicable to float and integer types, so the function returns an error when the parameter has a different type. This function

returns the minimum value of the parameter as string, or an empty string if it has not been declared.

- Get complex type: it returns the value of the attribute type defined in section 2.2.6.2 of E2E-ICD. Thus, the returned value is one of: string, integer, float, boolean, file, folder, array or matrix.
- Get element type: it returns the data type of the elements of a complex type. Thus, according to section 2.2.6.2 of E2E-ICD, the returned value is one of: string, integer, float, boolean, file (folder type is missing). If the parameter is not a complex type (i.e. array or matrix) the value returned is the output of the function get complex type.

4.2.3.5. Files and Folders

Two functions are provided to check if a file or folder provided by a parameter exists.

- File exists: returns true if the file exists and false otherwise. The function will raise an error if the parameter that provides the file is not of type file (or the elements if it is a complex type). For complex types like array or matrix, the operation will be performed element by element, returning an array or matrix of booleans.
- Folder exists: returns true if the folder exists and false otherwise. The function will raise an error if the parameter that provides the folder is not of type file (or the elements if it is a complex type). For complex types like array or matrix, the operation will be performed element by element, returning an array or matrix of booleans.

4.3. Process logic

In this section, the process logic of using the libraries in modules source code is shown. It is described for C++, Fortran, IDL, Matlab, Python and Java module developers. Note that additional documentation on the APIs available for several languages is available through [OSF-DOC].

4.3.1. C ++ Programming Language

4.3.1.1. CLP

Steps for using the Command Line Parser module:

3. Include the OSFI.h header file in your code
4. Create an instance of the CLP class passing the command line arguments of the main function. The constructor throws an exception in case of error, so remind to catch it.
5. Access the fields with one of the following recommended methods:
   - `getConfFiles()`
   - `getInputfiles()`
   - `getOutputFiles()`
6. Destroy the instance once not needed.

4.3.1.2. EHLog

Steps for using the Error Handler and Logging module:

1. Include the OSFI.h header file in your code
2. Use the static functions as a library of functionalities. No instance creation is needed.

4.3.1.3. ConFM

Steps for using the Configuration File Manager module:

1. Include the OSFI.h header file in your code
2. Create an instance of the ParamReader class passing the name of the XML configuration file and, optionally, the name of a XSD schema file to validate it. The constructor throws an exception in case of error, so remind to catch it. Alternatively, you can use the copy constructor to clone an existing instance. Two examples:

```
ParamReader * reader = new ParamReader("config.xml", "");
ParamReader * cloned = new ParamReader(reader);
```

3. Optionally, check for presence of a parameter with a given name. For example:

```
reader->existParameter ("group1.parameter1");
```

4. Access the parameter values by their complete name, using several methods:

- Directly, with no intermediate instances. For example:

```
reader->getParameter("group1.parameter1").getBooleanValue()
```

- Accessing the parameters map. For example:

```
t_params_map params = reader->getParameters();
params["group1.parameter1"].print();
```

- Using an intermediate instance of the Parameter class. For example:

```
Parameter param = reader->getParameter("group1.parameter1");
param.getBooleanValue()
```

- Accessing the list of parameters under a certain group. For example:

```
vector<Parameter> paramList = reader->getParameters("group1");
for (Parameter& p : paramList)
{
  p.print();
}
```

5. Destroy the instance once not needed.

4.3.2. ANSI C Programming Language

The OpenSF integration libraries for C language are designed as a bridge to provide integration between C++ code and C code. Module developers should take account on it because the use of the libraries, most of the times, is as simple as calling a function in module source code.

4.3.2.1. CLP

Steps for using the Command Line Parser module:

1. Include OSFIC.h header file in your code

```
#include "OSFIC.h"
```

2. Initialize the CLP module passing the command line arguments of the main function. The function returns an integer, 1 if the initialisation has been correct 0 otherwise, so remind to catch it.

```
int result = osfiCLP(argc, argv);
```

3. Access the fields with one of the following methods:

```
void osfiCLPGetConfFile(char * fileName, int *length);
void osfiCLPGetConfFiles(char *confFiles[], int *noFiles);
void osfiCLPGetInputFiles(char *inputFiles[], int *noFiles);
void osfiCLPGetOutputFiles(char *outputFiles[], int *noFiles);
```

4. Developers must previously allocate memory for storing file names and file name length.

```
char * fileName = malloc(sizeof(char)*MAX_FILE_ LENGTH NAME);
int length;
osfiCLPGetConfFile(fileName, &length);
```

4.3.2.2. EHLog

Steps for using the Error Handler and Logging module:

1. Include the OSFIC.h header file in your code
2. Use the static functions as a library of functionalities. No instance creation is needed.
3. Methods available in EHLog module are:

```
void osfiLoggerError (char * message,...);
void osfiLoggerWarning(char *message,...);
void osfiLoggerInfo(char *message,...);
void osfiLoggerDebug(char *message,...);
void osfiLoggerProgress(int n, int m);
void osfiLoggerFinishExecution(int errorCode);
```

4.3.2.3. ConFM

Steps for using the Configuration File Manager module:

1. Include the OSFIC.h header file in your code
2. Create an instance of the ParamReader class passing the name of the XML configuration file and, optionally, the name of a XSD schema file to validate it. The constructor throws an exception in case of error, so remind to catch it. The wrapper for ANSI C returns an integer so developers shall handle this exception through it.

```
int osfiConFMParamReader(char *fileName, char *schemaName);
```

3. Optionally, check for presence of a parameter with a given name.

```
void osfiConFMExistParameter (int *value, char *paramName);
```

4. Access the parameter values by their complete name, using two different methods:
   - Directly with the complete name of the parameter

```
void osfiConFMGetIntegerValue(int *value, char *paramName);
void osfiConFMGetDoubleValue(double *value, char *paramName);
bool osfiConFMGetBoolValue(char *paramName);
void osfiConFMGetFileValue(char *param, int *length, char
*paramName);
void osfiConFMGetStringValue(char *param, int *length, char
*paramName);
void osfiConFMGetVectorDoubleValues(double *doubleList, int *size,
char *paramName);
void osfiConFMGetVectorIntegerValues(int *intList, int *size, char
*paramName);
void osfiConFMGetMatrixDoubleValues(double *doubleMatrix, int *rows,
int *columns, char *paramName);
void osfiConFMGetMatrixIntegerValues(int *intMatrix, int *rows, int
*columns,
char *paramName);
```

   - Through a pointer to a osfiParameter structure:

```
typedef struct {
char *name;
char *description;
char *value;
char *units;
int dims[2];
char *min;
char *max;
} osfiParameter;
void osfiConFMGetParameter(osfiParameter *param, char *paramName);
```

5. Additionally developers can:
   - Prints a textual representation of a parameter or the list of parameters

```
void osfiConFMPrintParameters();
void osfiConFMPrintParameter(char *paramName);
```

- Access to the dimensions of a parameter in order to allocate the accurate memory for parameter values container

```
void osfiConFMGetDimension(char *paramName, int index, int *size);
```

4.3.3. Fortran Programming Language

The OpenSF integration libraries for Fortran language are designed using Fortran 2003 standard and mimic the design pattern (object oriented) and functions of the C++ implementation. It is to be highlighted that the Fortran libraries rely on the FFI library, which in turn calls the C++ implementation. However, due to the particularities of the Fortran language, there are certain differences that shall be noted.

First of all, Fortran does not support a run time error managing system like C++ or Java, where errors are raised and then contained within try-catch statements. In order to provide the module developer with a mechanism to manage the run time errors, all the Fortran functions which C++ equivalent can raise them have an optional output parameter called *stat* which value is zero if the function has been properly executed. Indeed, the value of the parameter is different from zero only if the OSFI C++ underlying functions raise an exception during the execution. In addition, objects returned by functions (e.g. when a parameter is requested) have a method called isValid that checks if the object returned is correct or not.

Another feature that shall be remarked is that the OSFI Fortran libraries use Fortran 2003 feature realloc_lhs or automatic allocation on intrinsic assignment. Before Fortran 2003 the array on the left-hand side of such an assignment statement must be allocated, and of the same shape as the array on the right-hand side. From Fortran 2003 only the rank needs match. If there is a shape mismatch, the array would be first deallocated and then reallocated to the correct shape. If it was not allocated initially, it would be allocated.

Therefore, the module developers shall make sure that the compiler selected supports this syntax.

KNOWN ISSUE: some functions of the C++ OSFI libraries do not raise exceptions when a problem is detected. Thus, in this case the Fortran function using it will not be able to report the problem and make it visible in to the module developer. Nevertheless, the C++ functions always write warnings when such thing happens, so the user is able to know if something has gone wrong. Future releases of the OSFI libraries will fix these bugs in the C++ implementation.

4.3.3.1. CLP

Steps for using the Command Line Parser module

1. Include the OSFI_CLP module in your code

```
Use OSFI_CLP
```

2. Create an instance of the OSFI_CommandLineParser type. Unlike in C++, this function does not take the command line arguments; instead it parses them by itself.

```
Type(OSFI_CommandLineParser) clp
clp = OSFI_CommandLineParser()
if (clp%isValid()) then
   ! Desired actions …
else
```

```
   ! Finish the execution with error code
   call osfi_finishExecution(1)
end if
```

3. Access the fields with the following subroutines
   - `getConfFiles(confFiles)`
   - `getInputFiles(inputFiles)`
   - `getOutputFiles(outputFiles)`

```
Type(OSFI_STR), Allocatable, Dimension(:) :: confFiles
Call clp%getConfFiles(confFiles)
```

4.3.3.2. EHLog

Steps for using the logging module.

1. Include the OSFI_EHLog module in your code

```
Use OSFI_EHLog
```

2. Use the static functions for logging a message.

```
call osfi_debug('Debugging')
call osfi_error('Error message')
call osfi_info('Info message')
call osfi_warning('Warning message')
call osfi_progress(2, 45)
```

4.3.3.3. ConFM

Steps for using the Configuration File Manager module.

1. Include the OSFI_ConFM module in your code

```
Use OSFI_ConFM
```

2. Parse the data of the XML configuration file with the function `OSFI_ParamReader(fileName, schemaName)`. Optionally a XSD schema file can be passed as argument if a validation of the XML file is required. This function returns a Logical to check if an error has happened. Remember to handle those errors.

```
pr = OSFI_ParamReader(confFiles(1)%str, '')
if (pr%isValid()) then
          ! Desired actions …
else
     ! Finish the execution with error code
     call osfi_finish_execution(1)
end if
```

3. Optionally, check for presence of a parameter with a given name.

```
Logical :: exist
exist = pr%existParameter('group1.group2.param1')
```

4. Access the parameter values by their complete name, using several subroutines:

```
Integer err
call pr%print()
p = pr%getParamRef(paramName, stat=err)
if(err == 0) then
call p%print()
end if
```

5. Accessing the list of parameters with double matrix structure. For example:

```
Double precision, Allocatable :: doubleMatrix(:,:)
Character(*), Parameter :: matrixDoubleParam = "group1.group2.param1
p = pr%getParamRef(matrixDoubleParam)
doubleMatrix = p%getMatrixDouble()
```

4.3.4. Fortran 77 Programming Language

The OpenSF integration libraries for Fortran 77 language are deprecated.

It is recommended to use the OpenSF integration libraries for Fortran (see 4.3.3).

4.3.4.1. CLP

Steps for using the Command Line Parser module.

1. Init the Command Line Parser using the subroutine OCLP()
2. Access the fields with one of the following methods:
   - `OCLPNC(nconf)`: get the number of configuration files
   - `OCLPNI(nin)`: get the number of input files
   - `OCLPNO(nout)`: get the number of output files
   - `OCLPGC(i, fname)`: get configuration file "i"
   - `OCLPGI(i, fname)`: get input file "i"
   - `OCLPGO(i, fname)`: get output file "i"

4.3.4.2. EHLog

Steps for using the Error Handler and Logging module.

1. Use the provided subroutines to generate logs:
   - `OLERR(mess)`: error message
   - `OLINFO(mess)`: information message
   - `OLWAR(mess)`: warning message
   - `OLDEB(mess)`: debug message
   - `OLPROG(n,m)`: progress message (step n of m)
   - `OLFE(errcod)`: finish execution with error code "errcod"
   - `OLQC(vname,value)`: quality with message
   - `OLQD(vname,value)`: quality with double value

4.3.4.3. ConFM

Steps for using the Configuration File Manager module.

1. Initialise the param-reader using the following subroutine:
   - `OPREAD(cnfile,scfile,stat)`
2. Use one of the following subroutines to access the parameter values or properties:

- `OPEX(pname, pexist)`: check if a parameter exists
- `OPGPR(pname, rows)`: get number of rows
- `OPGPC(pname, cols)`: get number of columns
- `OPDOUB(dvalue, pname)`: get double parameter
- `OPINT(ivalue, pname)`: get integer parameter
- `OPBOOL(bvalue, pname)`: get boolean parameter
- `OPFILE(fvalue, length, pname)`: get file parameter
- `OPSTR(svalue, length, pname)`: get string parameter
- `OPINV(vector, vsize, pname)`: get integer vector
- `OPDBV(vector, vsize, pname)`: get double vector
- `OPBLV(vector, vsize, pname)`: get boolean vector
- `OPSTRV(vector, vsize, pname)`: get string vector
- `OPFLV(vector, vsize, pname)`: get file vector
- `OPINM(vector, rows, cols, pname)`: get integer matrix
- `OPDBM(vector, rows, cols, pname)`: get double matrix
- `OPBLM(vector, rows, cols, pname)`: get boolean matrix

3. Close param reader:
   - `OPCLS()`

4.3.5. IDL Programming Language

The OpenSF integration libraries for IDL language are deprecated.

Before using the IDL library for OSFI, it is necessary to compile the corresponding modules: 'CLP.pro', 'Logger.pro', 'Parameter.pro' and 'ConFM.pro' so that all functions are available for IDL.

These files are located in: $OSFI_HOME/include/IDL/

A possible example is:

```
.COMPILE '/home/abma/OSFI/include/IDL/CLP.pro'
.COMPILE '/home/abma/OSFI/include/IDL/Logger.pro'
.COMPILE '/home/abma/OSFI/include/IDL/Parameter.pro'
.COMPILE '/home/abma/OSFI/include/IDL/ConFM.pro'
```

Once these files have been compiled, the developer can define objects of these classes in his own module, and run it.

4.3.5.1. CLP

Steps for using the Command Line Parser module:

1. Create an object of the CLP class passing it as arguments the configuration files, the input files and the output files. It is important to pass these arguments in the correct order.
2. Access the fields with one of the following methods:
   - `getConfFiles()`: Return all the configuration files inside a matrix
   - `getInputFiles()`: Return all the input files inside a matrix
   - `getOutputFiles()`: Return all the output files inside a matrix
   - `getConfFile(index)`: Return the configuration file at the position 'index'.
   - `getInputFile(index)`: Return the input file at the position 'index'.
   - `getOutputFile(index)`: Return the output file at the position 'index'.
3. Destroy the object once not needed.

An example of this procedure is shown below:

```
CLP = OBJ_NEW('CLP', ConfFiles, InputFiles, OutputFiles)
InputFiles = CLP->getInputFiles()
OutputFiles = CLP->getOutputFiles()
ConfFiles = CLP->getConfFiles()
Input = CLP->getInputFile(2)
Output = CLP->getOutputFile(3)
Conf = CLP->getConfFile(0)
OBJ_DESTROY, CLP
```

4.3.5.2. Logger

Steps for using the Logging module:

1. Create an object of the Logger class passing it as argument the debug mode (On=1 or Off=0).
2. Use one of its methods to show different types of messages in stdout:
   - error, message: Shows an error message in openSF format
   - warning, message: Shows a warning message in openSF format
   - info, message: Shows an information message in openSF format
   - debug, message: Shows a debug message in openSF format if debug mode is activated
   - progress, step, nsteps: Shows the progress of the module in openSF format
   - finishExecution: Shows that the module has finished with an information message
   - qualityReport, name, value: Shows a variable and its value
   - setDebugMode, debugMode: Set the debug mode property (On=1, Off=0).
3. Destroy the object once not needed.

An example of this procedure is shown below:

```
LOG = OBJ_NEW('Logger', DebugMode)
LOG->Info, "This is an info message"
LOG->warning, "This is a warning message"
LOG->error, "This is an error message"
LOG->debug, "This is a debug message"
LOG->progress, 2, 21
LOG->qualityReport, 'a', 23
OBJ_DESTROY, LOG
```

4.3.5.3. ConFM

Steps for using the Configuration File Manager module:

1. Create an object of the ConFM class passing the name of the XML configuration file.
2. Optionally, check for a parameter existence with a given name:

   xmlObj->ExistParameter('los.LOS.name')
3. Obtain a parameter of the configuration file by their complete name, using the associated method of ConFM class:

   parameter = GetParameter, path

   This method returns an instance of an object of the Parameter Class.
4. Access the parameter values using several methods:
   - `getPath()`: Returns the path of the parameter

- `getName()`:Returns the name of the parameter
- `getDescription()`:Returns the description of the parameter
- `getType()`:Returns the type of the parameter
- `getUnits()`:Returns the units of the parameter
- `getNDims()`:Returns the number of dimensions of the parameter
- `getDims()`:Returns the dimensions of the parameter
- `getValue()`:Returns the value of the parameter
- `getMin()`:Returns the minimum value of the parameter
- `getMax()`:Returns the maximum value of the parameter
- `print`: Shows all the attributes of the parameter in stdout

5. Destroy the objects of classes ConFM and Parameter once not needed.

An example of this procedure is shown below:

```
xmlObj = OBJ_NEW('ConFM', Conf)
xmlPar = xmlObj->GetParameter('los.LOS.name')
print, xmlPar->GetPath()
print, xmlPar->GetValue()
xml->print
OBJ_DESTROY, xmlPar
OBJ_DESTROY, xmlObj
```

4.3.6. Matlab Programming Language

In order to use OSFI in Matlab it is necessary to add the directory containing the OSFI-Matlab files to the search path. It can be easily done by reading the environment variable $OSFI_HOME:

```
% Add OSFI path
OSFI_HOME = getenv('OSFI_HOME');
OSFI_MATLAB = [OSFI_HOME '/include/Matlab/'];
addpath (OSFI_MATLAB);
```

To run the Matlab code from OpenSF, the designed module must be a function with the following structure:

```
function matlabModel (configurationParameters, inputs, outputs)
…
end
```

The following template of a Matlab module is provided to ease the development:

```
function matlabModel (configurationParameters, inputs, outputs)

% Check input arguments
if (nargin<3)
    error ('number of argumets not valid');
end

%--------------------------------------------------------------------
% OSFI Initialization and parameter reading
%--------------------------------------------------------------------
% Add OSFI path
OSFI_HOME = getenv('OSFI_HOME');
OSFI_MATLAB = [OSFI_HOME '/include/Matlab/'];
addpath (OSFI_MATLAB);

% Init CLP and Logger
clp = CLP (configurationParameters, inputs, outputs);
log = Logger ();

log.setDebugMode (true);

% TO-DO: get inputs, outputs and configuration files using
%        CLP getters:
%           + getConfFile(i)
%           + getInputFile(i)
%           + getOutputFile(i)
conf1 = clp.getConfFile (1);

% TO-DO: parse configuration files and read all the parameters
cfm = ConFM (conf1);
%--------------------------------------------------------------------
% Module Processing Core
%--------------------------------------------------------------------
```

```
% TO-DO: add code here to perform the processing
end
```

4.3.6.1. CLP

Steps for using the Command Line Parser module:

1. Create an instance of the CLP class passing the configuration parameters, inputs and outputs to the constructor:

```
clp = CLP (configurationParameters, inputs, outputs);
```

2. Access the fields with one of the following methods:
   - `getConfFiles()`
   - `getConfFile(n)`
   - `getInputfiles()`
   - `getInputfile(n)`
   - `getOutputFiles()`
   - `getOutputFile(n)`
   - `nConfFiles()`
   - `nInputFiles()`
   - `nOutputFiles()`

4.3.6.2. EHLog

Steps for using the Error Handler and Logging module:

1. Create an instance of the Logger class

```
log = Logger ();
```

2. Set the debug mode (true or false):

```
log.setDebugMode (true);
```

If debugMode is false debug messages are not shown.

3. Use the following methods to report logs:
   - `error(message)`
   - `warning(message)`
   - `info(message)`
   - `debug(message)`
   - `progress(step, totalSteps)`

- finishExecution()
- qualityReport (name, value)

```
log.info ('Writing output data');
```

4.3.6.3. ConFM

Steps for using the Configuration File Manager module:

1. Create an instance of the ConFM class passing the name of the configuration file:

```
cfm = ConFM (confFile);
```

2. Optionally, check for presence of a parameter with a given name:

```
exist  = cfm.existParameter ('thresholds.brightness');
```

3. Access the parameter values by their complete name:

```
paramBT  = cfm.getParameter ('thresholds.brightness');
```

4. Each parameter returned by the previous function is an object with several methods that allow accessing to the value and other properties. Use the following methods to read value and properties:
   - getValue()
   - getPath()
   - getName()
   - getDescription()
   - getType()
   - getNdims()
   - getDims()
   - getValue()
   - getMin()
   - getMax()

```
BT = paramBT.getValue();
minBT = paramBT.getMin();
maxBT = paramBT.getMax();
…
```

4.3.7. Python Programming Language

In order to use OSFI in Python the Environment variable PYTHONPATH needs to be configured to point to the necessary libraries (default configuration points to OSFI Python libraries PYTHONPATH=$OSFI_HOME/include/Python). This can be done thru the shell or inside a Python module by reading the environment variable $OSFI_HOME:

```
# Set PYTHONPATH
OSFI_HOME = os.environ['OSFI_HOME']
PYTHON_PATH = OSFI_HOME+"/include/Python"
os.environ['PYTHONPATH'] = PYTHON_PATH
```

To run the Python code from OpenSF, the designed module must have the following structure:

```
#!/usr/bin/python

from ParamReader import ParamReader
from CLP import CLP
import Logger

def main(argv=None):
    '''Command line options.'''
…

if __name__ == '__main__':
    main()
```

The following template of a Python module is provided to ease the development:

```
#!/usr/bin/python

import sys
import os
# Set PYTHONPATH
OSFI_HOME = os.environ["OSFI_HOME"]
sys.path.append(OSFI_HOME+"/include/Matlab")
from ParamReader import ParamReader
from CLP import CLP
import Logger

def main(argv=None):

    try:

        #-------------------------------------------------------------
--------
        # OSFI Initialization and parameter reading
        #-------------------------------------------------------------
--------

        # Init CLP
        clp = CLP(argv)
```

```
        # TO-DO: get inputs, outputs and configuration files using
        #        CLP getters:
        #           + getConfFiles()[i]
        #           + getInputFiles()[i]
        #           + getOutputFiles()[i]
        cf = clp.getConfFiles()

        # TO-DO: parse configuration files and read all the
parameters
        reader = ParamReader(cf[1], "")

        #--------------------------------------------------------------
--------
        # Module Processing Core
        #--------------------------------------------------------------
--------

        # TO-DO: add code here to perform the processing


        #  Finish and process errors
        Logger.finishExecution(0)
        return 0
    except Exception e:
        Logger.error("TestModule failed")
        Logger.finishExecution(1)
        raise(e)

if __name__ == '__main__':
    main()
```

4.3.7.1. CLP

Steps for using the Command Line Parser module:

1. Import the CLP module in your code
2. Create an instance of the CLP class passing the command line arguments of the main function. The constructor throws an exception in case of error, so remind to catch it.
3. Access the fields with one of the following methods:
   - `getConfFile()`
   - `getConfFiles()`
   - `getInputfiles()`
   - `getOutputFiles()`

4.3.7.2. EHLog

Steps for using the Error Handler and Logging module:

1. Import the Logger module in your code
2. Use the static functions as a library of functionalities. No instance creation is needed.

4.3.7.3. ConFM

Steps for using the Configuration File Manager module:

1. Import the ParamReader Python module in your code
2. Create an instance of the ParamReader class passing the name of the XML configuration file and, optionally, the name of a XSD schema file to validate it. The constructor throws an exception in case of error, so remind to catch it:

```
reader = ParamReader("config.xml", "")
```

3. Optionally, check for presence of a parameter with a given name:

```
reader.existParameter("group1.parameter1")
```

4. Access the parameter values by their complete name, using several methods:
   - Directly, with no intermediate instances. For example:

```
reader.getParameter("group1.parameter1").getBooleanValue()
```

   - Accessing the parameters map. For example:

```
params = reader.getAllParameters()
params["group1.parameter1"].write()
```

   - Using an intermediate instance of the Parameter class. For example:

```
param = reader.getParameter("group1.parameter1")
param.getBooleanValue()
```

   - Accessing the list of parameters under a certain group. For example:

```
paramList = reader.getParameters("group1")
for param in paramList:
  param.write()
```

4.3.8. Java Programming Language

4.3.8.1. CLP

Steps for using the Command Line Parser module:

1. Import the osfi.CLP class in your code;
2. Create an instance of the CLP class passing it as parameter the command line arguments of the main function. The constructor throws an exception in case of error, so it should be handled;
3. Access the data fields with one of the following methods:
   - `getConfFile()`
   - `getConfFiles()`
   - `getInputfiles()`
   - `getOutputFiles()`

4.3.8.2. EHLog

Steps for using the Error Handler and Logging module:

1. Import the osfi.Logger class in your code
2. Use the static functions as a library of functionalities. No instance creation is needed.

4.3.8.3. ConFM

Steps for using the Configuration File Manager module:

1. Import the osfi.ParamReader class in your code
2. Create an instance of the ParamReader class passing as argument the name of the XML configuration file and, optionally, the name of a XSD schema file to validate it. The constructor throws an exception in case of error, so it should be handled:

```
ParamReader reader = new ParamReader("config.xml", "");
```

3. Optionally, check for availability of a parameter with a given name:

```
reader.existParameter("group1.parameter1")
```

4. Access the parameter values by their complete name, using several methods:
   - Directly, with no intermediate instances. For example:

```
reader.getParameter("group1.parameter1").getBooleanValue()
```

   - Accessing the parameters map. For example:

```
Map<String, Parameter> params = reader.getAllParameters();
params.get("group1.parameter1").write()
```

- Using an intermediate instance of the Parameter class. For example:

```
Parameter param = reader.getParameter("group1.parameter1")
param.getBooleanValue();
```

- Accessing the list of parameters under a certain group. For example:

```
List<Parameter> paramList = reader.getParameters("group1");
for (Parameter param : paramList)
  param.write();
```

4.4. Additional Features

4.4.1. Debug Mode

Debug mode logs are activated creating the environment variable "DEBUG_MODE" and setting it to "On". By default if this variable is not present, no debug logs are shown during the execution.

```
export DEBUG_MODE=On
```

4.4.2. Coloured Logs

OSFI provides a mechanism to colour logs when the module is run from command line (only for Unix terminals).

Coloured logs are activated creating the environment variable "OSFI_LOG_COLOR" and setting it to "On".

```
export OSFI_LOG_COLOR=On
```

4.5. Examples of use

4.5.1. C++ Programming Language

Here is an example of C++ code that uses the different modules of the integration libraries.

```
#include "OSFI.h"
#include <iostream>
#include <cstdlib>
#include <list>
#include <string>

using namespace std;

int main(int argc, char * argv[]){

   string matrixIntParam = "los.LOS.polyParY";

   try
   {
      CLP CLP(argc, argv);

      cout << "input files = ";
      for (i = inputFiles.begin(); i != inputFiles.end(); ++i)
         cout << *i << " ";
      cout << endl;

      cout << "output files = ";
      for (i = outputFiles.begin(); i != outputFiles.end(); ++i)
         cout << *i << " ";
      cout << endl;
      string config = CLP.getConfFile();

      // Create a ParamReader
      ParamReader reader (config, "");
      Logger::info("Printing whole parameters file");
      reader.print();
      DynamicArray<int> mi;
      mi = reader.getParameter(matrixIntParam).getMatrixInt();
      Logger::info(matrixIntParam);
      for (int i = 0, n = mi.getRows(); i < n; i++)
      {
         for (int j = 0, m = mi.getColumns(); j < m; j++)
         {
            printf("%4d\t", mi[i][j]);
         }
         cout << endl;
      }

      vector<double> vd =
reader.getParameter(vectorDoubleParam).getVectorDouble();
```

```
    Logger::info(vectorDoubleParam);
    for (int i = 0, n = vd.size(); i < n; i++)
      printf("%4.1f\t", vd[i]);
    cout << endl;
} catch (...)
  {
    Logger::error("Example failed");
    Logger::finishExecution(1);
  }
} // End of main
```

4.5.1.1. C++ Compilation and Execution process.

The recommended build system is CMake. See section 3.4.3 for detailed instructions regarding the compilation process and section 3.4.2 for recommendations on building the modules.

Integration libraries come in two distribution types, shared or static libraries.

If you have linked the shared libraries you can execute the binary files after specifying the location of those shared libraries like this:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$OSFI_HOME/lib
```

Linking with static libraries does not imply to specify the location of the linked libraries since the executable already includes the object files.

The sentences for executing the test binaries are:

```
./cppExample <arguments>
```

4.5.2. ANSI C Programming Language

Here is an example of ANSI C code that uses the different modules of the integration libraries.

```
#include "OSFIC.h" // Include OSFI header for ANSI C Programming
Language
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

   char* doubleParam = "los.LOS.initialTime.second";
   char* matrixDoubleParam = "sensor.NumericModel.polyParX";

   osfiCLP(argc, argv); // Initialize CLP component

   if (osfiConFMParamReader(configurationFile, "") == 1) {

      osfiLoggerInfo("ConFM::Configuration File Parsed");

      double doubleParameter;
      osfiConFMGetDoubleValue(&doubleParameter, doubleParam);

      // Matrix Parameters
      int x, y;
      osfiConFMGetDimension(matrixDoubleParam, 1, &x);
      osfiConFMGetDimension(matrixDoubleParam, 2, &y);
      double *doubleMatrix = malloc(x * y * sizeof(double));
      int rows;
      int columns;
      osfiConFMGetMatrixDoubleValues(doubleMatrix, &rows, &columns,
           matrixDoubleParam);
      for (i = 0; i < rows; i++) {
         for (j = 0; j < columns; j++) {
           printf("Matrix Double %s [%d[%d]= %f\n", matrixDoubleParam,
i,
                j, doubleMatrix[(i * columns) + j]);
         }
      }
      free(doubleMatrix);

      //Parameter Structure Retrieval
      osfiParameter * param = malloc(sizeof(osfiParameter));
      param->name = malloc(MAX_PARAMETER_NAME_SIZE * sizeof(char));
      param->description = malloc(MAX_PARAMETER_DESC_SIZE *
sizeof(char));
      param->value = malloc(MAX_PARAMETER_FIELD_SIZE * sizeof(char));
      param->units = malloc(MAX_PARAMETER_FIELD_SIZE * sizeof(char));
      param->min = malloc(MAX_PARAMETER_FIELD_SIZE * sizeof(char));
      param->max = malloc(MAX_PARAMETER_FIELD_SIZE * sizeof(char));
      osfiConFMGetParameter(param, matrixDoubleParam);
      free(param->name);
      free(param->description);
```

```
    free(param->value);
    free(param->units);
    free(param->min);
    free(param->max);
    free(param);
  } else {
    osfiLoggerError("ConFM::Error Loading Configuration File\n");
  }
  osfiLoggerFinishExecution(0);
  return 0;
}
```

4.5.2.1. ANSI C Compilation and Execution process

The recommended build system is CMake. See section 3.4.3 for detailed instructions regarding the compilation process and section 3.4.2 for recommendations on building the modules.

Integration libraries come in two distribution types, shared or static libraries.

If you have linked the shared libraries you can execute the binary files after specifying the location of those shared libraries like this:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$OSFI_HOME/lib
```

Linking with static libraries does not imply to specify the location of the linked libraries since the executable already includes the object files.

The sentences for executing the test binaries are:

```
./cExample <arguments>
```

4.5.3. Fortran Programming Language

Here is an example of Fortran code that uses the different modules of the integration libraries.

```
Program f90Example

   ! Include the OSFI modules
Use OSFI_ConFM
Use OSFI_CLP
Use OSFI, Only: osfi_error, osfi_info, osfi_finishExecution

   Type(OSFI_CommandLineParser) clp
   Type(OSFI_STR), Allocatable, Dimension(:) :: confFiles,
inputFiles, outputFiles
   Character(*), Parameter :: matrixDoubleParam =
"sensor.NumericModel.polyParX"
Double precision, Allocatable :: doubleMatrix(:,:)
   Logical :: good
   Integer index
   Character(1) tmp
   Integer :: i, err

   clp = OSFI_CommandLineParser()
   if (clp%isValid()) then

      ! Parse configuration files
      Call clp%getConfFiles(confFiles)
      if (.not. allocated(confFiles)) Then
         call osfi_error('Configuration files were not parsed')
         call osfi_finishExecution(1)
      End If
      do index=1, size(confFiles)
         write (tmp, '(I1)') index
         call osfi_info("Configuration File [" // tmp // "] = " //
confFiles(index)%str)
      end do

      ! ConFM Module Example
      pr = OSFI_ParamReader(confFiles(1)%str, '')
      if (.Not. pr%isValid()) then ! Triggered if the file is not
found or cannot be parsed
         call osfi_error(`F90Example Failed: could not read file ' //
confFiles(1)%str)
         call osfi_finishExecution(2)
      End If

      call osfi_info("Printing whole parameters file")
      call pr%print()
      p = pr%getParamRef(matrixDoubleParam, stat=err)
      If (err /= 0) Then
         call osfi_error('Parameter ' // matrixDoubleParam // ' does
not exist')
         call osfi_finishExecution(2)
```

```
     End If
     doubleMatrix = p%getMatrixDouble(stat=err)

     if (err == 0) then
        call osfi_info(matrixDoubleParam)
        do i=1,size(doubleMatrix,1)
          write(*,*) doubleMatrix(i,:)
        end do
     Else
        call osfi_error('F90Example Failed: could not read matrix
double param')
        call osfi_finishExecution(2)
     end if

     call osfi_finish_execution(0)
End program
```

4.5.3.1. Fortran Compilation and Execution instructions.

The recommended build system is CMake. See section 3.4.3 for detailed instructions regarding the compilation process and section 3.4.2 for recommendations on building the modules.

Integration libraries come in two distribution types, shared or static libraries.

If you have linked the shared libraries you can execute the binary files after specifying the location of those shared libraries like this:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$OSFI_HOME/lib
```

Linking with static libraries does not imply to specify the location of the linked libraries since the executable already includes the object files.

The sentences for executing the test binaries are:

```
./f90Example <arguments>
```

4.5.4. Fortran 77 Programming Language

Here is an example of Fortran 77 code that uses the different modules of the integration libraries.

```
      program test
      implicit none

      INTEGER*4 n,m,errcod,stat
      CHARACTER*255 cc, fname
      CHARACTER*32 vname, cnfile, scfile, pname
      REAL*8 value,dvalue
      Character*66 tmp
      INTEGER*1 nconf, nin, nout, i

c---- ConFM param variables
      INTEGER nrows, ncols, length, vsize, k, j, p
      LOGICAL*1 pexist
      INTEGER ivalue
      REAL*8 dvector(25)
      LOGICAL*1 bmatrix(255)

c -----------------------------------------------------
c       TEST OSFI COMMAND LINE PARSER
c -----------------------------------------------------
      WRITE (*,*) ''
      call OLINFO ('----------------------------')
      call OLINFO ('      TESTING CLP')
      call OLINFO ('----------------------------')
c     call OCLP(stat)

c     Get number of configuration files
      call OCLPNC (nconf)
      WRITE(tmp, '(I2)') nconf
      call OLINFO ('Number of configuration files:  '//tmp)

c     Print configuration files
      i = 1
      DO WHILE (i.NE.nconf+1)
         call OCLPGC (i, fname)
         WRITE (tmp,'(I2)') i
         call OLINFO ('Configuration file '//tmp(1:4)//': '//fname)
         i = i + 1
      END DO

c -----------------------------------------------------
c       TEST OSFI LOGGER
c -----------------------------------------------------
      WRITE (*,*) ''
      call OLINFO ('----------------------------')
      call OLINFO ('      TESTING LOGGER')
      call OLINFO ('----------------------------')

      n = 3
```

```
      m = 4
      call OLPROG(n,m)

      cc = 'Test OSFI Error Message'
      call OLERR(cc)

      cc = 'Test OSFI Warning Message'
      call OLWAR(cc)

      vname = 'QualityVariable'
      value = 10.5
      call OLQD(vname,value)

      cnfile = 'exampleFile.xml'
      scfile = ''

      cc = 'Test OSFI Debug Message'
      call OLDEB(cc)

c -----------------------------------------------------
c       TEST OSFI PARAM READING
c -----------------------------------------------------
      WRITE (*,*) ''
      call OLINFO ('----------------------------')
      call OLINFO ('      TESTING PARAM READING')
      call OLINFO ('----------------------------')

c---- Param reader initialization
      call OPREAD(cnfile,scfile,stat)
      if (stat.NE.1) then
         cc = 'Error Parsing '//trim(cnfile)
         call OLERR(cc)
      end if

c---- Check if parameter exist
      call OLINFO('')
      call OLINFO('Testing if a parameter exists or not:')

      pname = 'los.LOS.polyParY'//CHAR(0)
      call OPEX (pname, pexist)
      WRITE (tmp,*) pexist
      call OLINFO('   los.LOS.polyParY:    '//tmp)

c---- READING INTEGER SCALAR PARAMETER
      pname = 'earth.Earth.demType'
      call OPINT(ivalue, pname)
      WRITE(tmp, *) ivalue
      cc = 'Integer parameter '//TRIM(pname)//' = '//tmp
      call OLINFO(cc)

c---- READING DOUBLE VECTOR PARAMETERS
      pname = 'los.LOS.doubleVector'
      call OPDBV (dvector, vsize, pname)
      cc = 'Double vector '//TRIM(pname)//':'
      call OLINFO(cc)
```

```
      WRITE (*, '(F12.1)') dvector (1:vsize)

c---- READING BOOLEAN MATRIX PARAMETERS
      pname = 'los.LOS.flagsMatrix'
      call OPBLM (bmatrix, nrows, ncols, pname)
      cc = 'Boolean matrix '//TRIM(pname)//':'
      call OLINFO(cc)
      k = 0
      DO WHILE (k.NE.nrows)
         j = 0
         DO WHILE (j.NE.ncols)
            p = k*ncols + j
            WRITE (*,20) 'row=', k, ' col=', j, ' ==> ', bmatrix(p+1)
 20         FORMAT(A4,I1,A5,I1,A5,L)
            j = j + 1
         END DO
      k = k + 1;
      END DO

c---- Close OSFI param-reader
      call OPCLS()

c---- Exit with error code
      errcod = 0
      call OLFE(errcod)
      end
```

4.5.4.1. Fortran 77 Compilation and Execution Process

The compilation process needs to specify the base location of the packages with these environment variables:

- OSFI_HOME, typically < openSF_install_dir>/OSFI

To compile your sources you must specify the location of the header files and the library binaries. This sentence is a valid example for compiling one of the distributed test sources.

```
gfortran -o test test.f -losfi-f77 -losfi-common-c -losfi-common -
lxerces-c -L$OSFI_HOME/lib
```

Integration libraries come in two distribution types, shared or static libraries.

If you have linked the shared libraries you can execute the binary files after specifying the location of those shared libraries like this:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$OSFI_HOME/lib
```

Linking with static libraries does not imply to specify the location of the linked libraries since the executable already includes the object files.

The sentences for executing the test binaries are:

```
./test <arguments>
```

4.5.5. IDL Programming Language

Here is an example of IDL code that uses the different modules of the integration libraries.

```
;  openSF Integration Libraries (OSFI)

PRO test_IDL, ConfFiles, InputFiles, OutputFiles, DebugMode

IF N_PARAMS() LT 3 THEN BEGIN
  EXECUTION_MODE = GETENV('IDL_EXECUTION_MODE')
  IF (STRCMP(EXECUTION_MODE, 'SAV') NE 1) THEN $
    print, 'Number of arguments not valid'
ENDIF

IF N_PARAMS() EQ 3 THEN $
  DebugMode = 0

;Show some logs
print, ''
print, 'Show some logs examples using Logger class...'
LOG = OBJ_NEW('Logger', DebugMode)
LOG->Info, "This is an info message"
LOG->warning, "This is a warning message"
LOG->debug, "This is a debug message"
LOG->progress, 2, 21
LOG->qualityReport, 'a', 23

;Show configuration files, inputs and outputs using CLP
print, ''
print, 'Parsing configuration, input and output files using CLP
class...'
CLP = OBJ_NEW('CLP', ConfFiles, InputFiles, OutputFiles)
InputFiles = CLP->GetInputFiles()
OutputFiles = CLP->GetOutputFiles()
ConfFiles = CLP->GetConfFiles()
Input = CLP->GetInputFile(2)
IF (N_ELEMENTS(ConfFiles) EQ 1) THEN BEGIN
  Conf = CLP->getConfFile(0)
ENDIF ELSE BEGIN
  Conf = CLP->getConfFile(1)
ENDELSE

LOG->Info, "Configuration files: " + ConfFiles
LOG->Info, "Input files: " + InputFiles
LOG->Info, "Output files: " + OutputFiles
LOG->Info, "Configuration file: " + Conf
LOG->Info, "Input file: " + Input
  success = 1

;Parse XML file and check read values
print, ''
print, 'Parsing XML file and checking that read values are
correct...'
```

```
xmlObj = OBJ_NEW('ConFM', Conf)

xmlPar = xmlObj->GetParameter('los.LOS.name')
IF (STRCMP(xmlPar->GetValue(), 'my LOS') EQ 1) THEN BEGIN
  print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
  print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
  success = 0
ENDELSE

xmlPar = xmlObj->GetParameter('los.LOS.polyParY')
value = xmlPar->GetValue()
result = [1,2,3,4,5,6,7,8,9,10,11,12]
IF max(value-result) EQ min(value-result) THEN BEGIN
  print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
  print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
  success = 0
ENDELSE

xmlPar = xmlObj->GetParameter('los.LOS.initialTime.year')
value = xmlPar->GetValue()
IF value EQ 2009 THEN BEGIN
  print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
  print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
  success = 0
ENDELSE

xmlPar = xmlObj->GetParameter('los.LOS.missionNames')
value = xmlPar->GetValue()
result = ['BioMass','Premier','CoreH2O']
IF where(strcmp(value, result) NE 1) EQ -1 THEN BEGIN
  print, 'Parameter: ' + xmlPar->GetPath() + ' --> OK'
ENDIF ELSE BEGIN
  print, 'Parameter: ' + xmlPar->GetPath() + ' --> No OK'
  success = 0
ENDELSE

OBJ_DESTROY, xmlPar
OBJ_DESTROY, xmlObj
OBJ_DESTROY, CLP
OBJ_DESTROY, LOG

print, ''

IF success EQ 1 THEN $
  print, 'Successful test' $
ELSE $
  print, 'Failed test'

END
```

4.5.5.1. IDL licenses

IDL provides three types of licenses in function of the needs of the user:

- 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.
- IDL runtime: Allows executing IDL programs precompiled and saved as .SAV 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. This kind of license has a few restrictions, like displaying a splash screen on startup, callable IDL applications are not available...

4.5.6. Matlab programming language

Here is an example of Matlab code that uses the different modules of the integration libraries.

```
function CloudsDetection (configurationParameters, inputs, outputs)

% Check input arguments
if (nargin<3)
    error ('number of argumets not valid');
end

%--------------------------------------------------------------------
% OSFI Initialization and parameter reading
%--------------------------------------------------------------------
% Add OSFI path
OSFI_HOME = getenv('OSFI_HOME');
OSFI_MATLAB = [OSFI_HOME '/include/Matlab/'];
addpath (OSFI_MATLAB);

% Init CLP and Logger
clp = CLP (configurationParameters, inputs, outputs);
log = Logger ();
log.setDebugMode (true);

% Get inputs, outputs and configuration files using
inputFolder = clp.getInputFile (1);
outFile     = clp.getOutputFile (1);
confFile    = clp.getConfFile (1);

% Parse configuration files and read all the parameters
log.info (['Reading configuration parameters from ' confFile]);
cfm = ConFM (confFile);

brightness_threshold  = cfm.getParameter
('thresholds.brightness').getValue;
NDSI_threshold        = cfm.getParameter
('thresholds.NDSI').getValue;
temperature_threshold = cfm.getParameter
('thresholds.temperature').getValue;
composite_threshold   = cfm.getParameter
('thresholds.composite').getValue;
filter5_threshold     = cfm.getParameter
('thresholds.filter5').getValue;
filter6_threshold     = cfm.getParameter
('thresholds.filter6').getValue;
filter7_threshold     = cfm.getParameter
('thresholds.filter7').getValue;
filter8_threshold     = cfm.getParameter
('thresholds.filter8').getValue;

%--------------------------------------------------------------------
% Module Processing Core
%--------------------------------------------------------------------
```

```
% Read input images
log.info ('Reading input files');
BLUE  = imread ([inputFolder '/B10.TIF']); % blue-green band
GREEN = imread ([inputFolder '/B20.TIF']); % green
RED   = imread ([inputFolder '/B30.TIF']); % red
NIR   = imread ([inputFolder '/B40.TIF']); % near infrared
MIR1  = imread ([inputFolder '/B50.TIF']); % mid-infrared
TIR   = imread ([inputFolder '/B60.TIF']); % thermal infrared
MIR2  = imread ([inputFolder '/B70.TIF']); % mid-infrared
[rows cols] = size (BLUE);

% Process images
log.info ('Processing images');
OUT = [];
NDSI      = (GREEN - MIR1)./(GREEN + MIR1);
composite = (1 - MIR1).*TIR;
filter5   = NIR./RED;
filter6   = NIR./GREEN;
filter7   = NIR./MIR1;
filter8   = MIR1./TIR;

NO_CLOUD   =
(RED<brightness_threshold)|(NDSI>NDSI_threshold)|(TIR>temperature_thr
eshold);
AMBIGUOUS  =
((composite>composite_threshold)|(filter5>filter5_threshold)|(filter6
>filter6_threshold)|(filter7<filter7_threshold));
WARM_CLOUD = (filter8>filter8_threshold);
COLD_CLOUD = (filter8<=filter8_threshold);

OUT = AMBIGUOUS*50;
pos = find (OUT==0);
OUT(pos) = WARM_CLOUD(pos)*150 + COLD_CLOUD(pos)*255;
OUT = OUT.*not(NO_CLOUD);

% Write data
log.info ('Writing output data');
imwrite (uint8(OUT), outFile);

end
```

4.5.6.1. Matlab Compilation and Execution Process

Since Matlab is an interpreted language it can't be compiled. There are only two prerequisite to execute a model from OpenSF:

1. Matlab
2. Each model must be a function with the following format (see section 4.2.5):

```
function matlabModel (configurationParameters, inputs, outputs)
…
end
```

4.5.7. Python Programming Language

Here is an example of Python code that uses the different modules of the integration libraries.

```
#!/usr/bin/python

from ParamReader import ParamReader
from CLP import CLP
import Logger

def main(argv=None):
    matrixIntParam = 'los.LOS.polyParY'
    vectorDoubleParam = 'los.LOS.iDomain'

    try:

        clp = CLP(argv)

        # Show conf files, inputs and outputs using CLP
        cf = clp.getConfFiles()
        Logger.info ('Configuration files: ' +  ', '.join(cf))
        inf = clp.getInputFiles ()
        Logger.info ('Input files:         ' + ', '.join(inf))
        outf = clp.getOutputFiles ()
        Logger.info ('Output files:        ' + ', '.join(outf))

        # Local parameters reading
        reader = ParamReader(cf[1], "")
        Logger.info("Printing whole parameters file")
        reader.write()

        mi = reader.getParameter(matrixIntParam).getMatrixInt()
        Logger.info(matrixIntParam)
        for i in range(len(mi)):
            for j in range(len(mi[0])):
                print "[%s][%s] = %s" % (i, j, mi[i][j])

        vd = reader.getParameter(vectorDoubleParam).getVectorDouble()
        Logger.getInfoStream().write(vectorDoubleParam)
        print("\n")
        for d in vd:
            print(d)

        Logger.finishExecution(0)
        return 0
    except Exception:
        Logger.error("TestModule failed")
        Logger.finishExecution(1)
```

4.5.7.1. Python Compilation and Execution process.

Since Python is an interpreted language it can't be compiled. There are only two prerequisite to execute a model from OpenSF:

1. Python correctly installed
2. Environment variable PYTHONPATH configured to point to the necessary libraries (default configuration points to OSFI Python libraries PYTHONPATH=$OSFI_HOME/include/Python)

4.5.8. Java Programming Language

Below is an example of Java code that uses the different modules of the integration libraries.

```
import java.util.ArrayList;

import osfi.CLP;
import osfi.Logger;
import osfi.ParamReader;
import osfi.Parameter;

public class TestModel {

  public static void main(String[] args) {

    try {
      CLP clp = new CLP(args);
      ArrayList<String> cf = clp.getConfFiles();

      Logger.info ("This is an info message");
      Logger.warning ("This is a warning message");
      Logger.debug ("This is a debug message");

      ParamReader cfm = new ParamReader(cf.get(1),"");
      Logger.info ("Configuration files: " +
CLP.arrayToString(cf));
      ArrayList<String> inf = clp.getInputFiles();
      Logger.info ("Input files:         " +
CLP.arrayToString(inf));
      ArrayList<String> outf = clp.getOutputFiles();
      Logger.info ("Output files:        "  +
CLP.arrayToString(outf));

      Parameter param = cfm.getParameter ("los.LOS.iDomain");
      Double[] valueVectorDouble = param.getVectorDouble();
      for (int i = 0; i < valueVectorDouble.length; i++) {
        System.out.println(valueVectorDouble[i]);
      }

      param = cfm.getParameter ("matrix5x4");
      Integer[][] matrix = param.getMatrixInt();
      for (int i = 0; i < matrix.length; i++) {
        for(int j = 0; j< matrix[i].length; j++) {
          System.out.println(matrix[i][j]);
        }
      }
    } catch (Exception e) {
      Logger.error("TestModule failed");
      Logger.finishExecution(1);
    }
  }
} // End of main
```

4.5.8.1. Java Compilation and Execution process.

The recommended build system is CMake. See section 3.4.3 for detailed instructions regarding the compilation process and section 3.4.2 for recommendations on building the modules.

The command line for executing the example is:

```
java -cp .:$OSFI_HOME/lib/osfi.jar TestModel <arguments>
```

For execution from openSF the user module needs to packaged as a jar file:

```
jar -cvmf MANIFEST.MF TestModel.jar TestModel.class
```

where the manifest file can be filled as shown below

```
Manifest-Version: 1.1
Created-By: Deimos Space, SL
Main-Class: TestModel
Class-Path: lib/osfi.jar
```

End of Document