Integration Libraries for the open Simulation Framework

OSFI

DEVELOPER'S MANUAL

Code : OPENSF-DMS-OSFI-DM
Issue : 1.18
Date : 17/07/2019

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

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

This page intentionally left blank

Document Information

Contract Data	
Contract Number:	4000120205/17/NL/CT/mg
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	ESA/ESTEC	1
Montserrat Pinõl Sole	ESA/ESTEC	1
Marcos Bento	ESA/ESTEC	1

Archiving	
Word Processor:	MS Word 2016
File Name:	OPENSF-DMS-OSFI-DM-118.doc

Document Status Log

Issue	Change description	Date
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 4.4 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
1.15	Introduction of new sections on building modules with OSFI and upgrading from previous OSFI versions. Updated section on the API and added coverage tables.	14/06/2018
1.16	Updated Sec. 3.4.1 to clarify the use of CMake for Python and Matlab implementations.	14/12/2018
1.17	Removed references to specific OSFI releases	11/06/2019

1.18	Updated Sec. 3.4.3.3 to include build instructions for generating a Python Wheel binary package. Updated Sec. 4.5.7 to include examples of use of OSFI Python through an installed package.	17/07/2019

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
PM	Progress Meeting Project Manager
PMP	Project Management Plan
QA	Quality Assurance
RD	Reference Document
RID	Review Item Discrepancy

Acronym	Description
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 Rev. 2.4

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 3.14
[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

Reference	Code	Title	Issue	Date
[C++11]	ISO/IEC 14882:2011	Information technology – Programming languages – C++	Ed.3	Sep 2011
[C99]	ISO/IEC 9899:1999	Information technology – Programming languages – C	Cor3, ed.1	Jan 2007
[F2003]	ISO/IEC 1539-1:2004	Information technology – Programming languages – Fortran	Cor4, ed.1	Sep 2009
[Java7]	https://docs.oracle.com/javase/specs/jls/se7/html	The Java® Language Specification	Java SE 7	Feb 2013
[Py2.7]	https://docs.python.org/2.7/reference/	The Python Language Reference	2.7	Jul 2010
[Py3.5]	https://docs.python.org/3.5/reference/	The Python Language Reference	3.5	Sep 2015
[F77]	ISO 1539:1980	Programming languages – FORTRAN	1	Mar 1980
[PEP427]	https://www.python.org/dev/peps/pep-0427/	The Wheel Binary Package Format 1.0	1	Sep 2012

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 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 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 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® Runtime Environment, Standard Edition 1.7 or superior	Oracle Binary Code License Agreement for the Java SE Platform Product	http://www.oracle.com/technetwork/java/index.html

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.

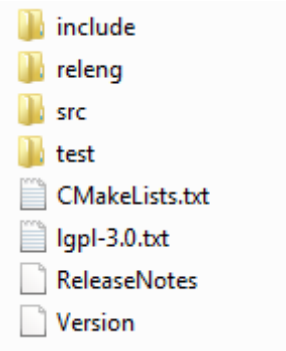

Figure 1: OSFI distribution

3.4.1. Build Instructions

Note: The build step described in this section is not necessary for the Matlab and Python versions of OSFI. These libraries can be used directly as source code, but notice that the build process provisions packaging and testing for all versions of OSFI, including those languages.

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 that are not deprecated (F77 and IDL). Nevertheless, the user can choose which languages to build by providing the **OSFI_LANGS** argument with either:

- A semicolon-separated list of the languages to be built. E.g.: CXX;Fortran;Python
- The special value "ALL" which enables all languages, even the deprecated ones

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-C 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 there for the Xerces library.
- **BUILD_XERCES**: if this boolean flag is set to true, CMake will download and build Xerces-C 3.2.0 in the *xerces/ExternalProject* folder under the build folder.

Other optional arguments accepted by the build configuration are:

- **BUILD_SHARED_LIBS**, default value OFF. If set to ON, the build process generates shared libraries. If not, static libraries are created.
- **BUILD_DOC**, default value ON. Enables building the Doxygen documentation, but requires Doxygen to be installed on the machine.
- **BUILD_TESTING**, default value ON. If set to ON, each language will generate extra targets to build tests for OSFI itself, and the "test" target will be available to run such tests. Note that some tests are language-specific and may require extra

libraries, e.g. some C++ tests require the GoogleTest library, while some of the Java tests use JUnit.

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 -D "OSFI_LANGS=CXX;Fortran;Python" ../
```

Once the build system is configured, the selected OSFI libraries with the appropriate build-system specific command; or in general with the following call to CMake executed from the build directory:

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

If enabled, the OSFI unit tests can be launched by using the "test" target of the build system, or with the following command executed from the build directory:

```
$ ctest
```

3.4.2. Packaging and/or Installation

An additional "package" target is available to generate an "installation package" 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 for C/C++, plus the Python and Matlab 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 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*.

3.4.3. Building and Distribution of Modules with OSFI

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.

This section provides instructions for building the modules using CMake, the suggested build system. It assumes that OSFI (and Xerces, where needed) are already built. It also provides advice for setting up such modules correctly to ease their distribution and execution from an E2E simulation orchestrator compatible with [E2E-ICD], like openSF.

3.4.3.1. C++, C and Fortran

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 module programming language. In the example below, both C++ and Fortran are selected.

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

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

```
enable_language(CXX)
add_executable(mymodule main.cpp otherfile.cpp)
target_link_libraries(mymodule OSFI::osfi-common)
```

In order to distribute such a module, the integrator must ensure that **all** required dynamic libraries are available when it is going to be executed. Note that OSFI and Xerces are only part of the larger set of libraries required by the program: the C/C++/Fortran runtime may also need to be distributed in a platform-dependent manner. For example:

- In Linux, the "rpath" attribute of the executable itself may list absolute or module-relative paths to try. Thus, build options could be provided to set rpath to "$ORIGIN/../lib", so the module could be distributed in a folder called "X/bin" and its dynamic libraries (OSFI/Xerces, runtimes…) could be placed in "X/lib".
- In Windows, the folder containing the executable is tried first, then those in the PATH variable. Thus, OSFI and Xerces, if built as DLLs, could simply be deployed alongside the executable file.

Another option is to try and build a fully statically-linked executable, which may bloat the binary size but makes it easily redistributable. However, this option may require custom build options, and is not always available in all platforms.

Thus, the module developer/integrator would have to use the proper build settings to create the desired scenario (building with/without dynamic libraries) and then take the proper actions on installation to deploy all needed files in the necessary structure.

3.4.3.2. Java

Should the user want to use CMake for building a Java module with OSFI (instead of Maven, Ant or other systems), they may use the *find_package* command to import the OSFI-Java target. Unlike in C++, there is no need to find other libraries first; only Java should be available beforehand. The OSFI libraries are added using the same command as before, and the *COMPONENTS* option allows selecting only some languages:

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

After these commands, OSFI is available for the building process, which shall be performed with the CMake commands. The OSFI-Java target can be linked into a JAR:

```
find_package(Java 1.7 COMPONENTS Development) # find javac
include(UseJava)
add_jar(mymodule com/dms/ModMain.java com/dms/ModComp.java
    ENTRY_POINT com/dms/ModMain
    INCLUDE_JARS OSFI::osfi-java
)

# Install phase: copy the JAR to the "bin" folder
install_jar(TestModel DESTINATION bin)
# Then copy the OSFI jar to some other path we can find, e.g. "lib"
get_target_property(osfijar OSFI::osfi-java JAR_FILE)
install(FILES "${osfijar}" DESTINATION "lib")
```

In order to distribute such a module, the OSFI-Java JAR needs to be available on execution too. Note, however, that the CMake Java-related functions will not install the OSFI JAR along with the module automatically, hence the extra step above.

For the same reason, and since it does not know where such dependencies will be found, CMake does not add them to the Class-Path attribute in the JAR manifest either. The main consequence is that the class path needs to be indicated on execution, as follows:

```
$ java -cp bin/mymodule.jar:lib/osfi.jar com.dms.ModMain
```

In order to ensure that the class path stored in the JAR can find the dependencies, the module developer needs to decide at build time where they will be installed (relative to the module JAR file). The following addition will generate the required Class-Path entry:

```
# Generate a custom manifest so we can put the OSFI jar in the
# Class-Path property. We will install the module at X/bin, and its
# dependencies at X/lib.
file(GENERATE
    OUTPUT "${CMAKE_CURRENT_BINARY_DIR}/MANIFEST.MF"
    CONTENT [[Class-Path: ../lib/$<TARGET_FILE_NAME:OSFI::osfi-java>
]]) # Note this extra newline in the content - it is important!
# Otherwise this will not be "merged" with the CMake-made manifest
add_jar(mymodule com/dms/ModMain.java com/dms/ModComp.java
```

```
    ENTRY_POINT com/dms/ModMain
    INCLUDE_JARS OSFI::osfi-java
    MANIFEST "${CMAKE_CURRENT_BINARY_DIR}/MANIFEST.MF"
)
```

If the module is built and installed with this configuration, it can be executed with the default class path in the JAR manifest, like:

```
$ java -jar bin/mymodule.jar
```

3.4.3.3. Python and Matlab

Python and Matlab are mainly interpreted languages, and thus they are not compiled by CMake in any way. The only condition to execute a module using OSFI in those languages is that the libraries in question are accessible to the modules at runtime, using any of the language-specific methods:

- In Python: two different approaches are feasible:
  - Using Python modules: if a single version of OSFI is used, the library may be installed under the site-packages directory of the Python interpreter. If it ought to be distributed alongside the module, the OSFI Python modules shall be available in the directory in which the interpreter is running, or the OSFI Python path can be added to the PYTHONPATH environment variable, or to the *sys.path* list.
  - Using a Python package: a Python binary package can be generated from the distributed sources and installed in the system or in a virtual environment, being automatically available for any module that desires to import it.
- In Matlab: the library needs to be available in the Matlab path. In general, the module function itself will be in the Matlab path, so the OSFI library files could be placed in the same folder, this may not be valid solution if the target folder may be cluttered.

If the Python/Matlab modules are going to be used as they are (first approach for Python and unique approach for Matlab), CMake can be used to aid in the install and distribution, since the OSFI build defines imported targets for the Python and Matlab OSFI libraries. Such targets are defined as INTERFACE libraries, which places them in the same category as C++ header-only libraries, for example.

This means that they are not *built* as such, but still are able to provide relevant properties, in particular INTERFACE_INCLUDE_DIRECTORIES, which is a list of paths that will contain the OSFI-Python/Matlab library directory as its first item. It can then be used in order to install the library alongside the module as follows:

```
find_package(OSFI REQUIRED CONFIG
                  COMPONENTS Python Matlab
                  HINTS "${OSFI_HOME}")

# Install our Python module to the bin folder. Using PROGRAMS will
# ensure that the executable bit is set
install(PROGRAMS "mymodule.py" DESTINATION bin)
```

```
# Also get the path where the OSFI-Python libs are, and copy them
# to the lib/OSFI folder on install
get_property(OSFI_PYTHON_DIR
    TARGET OSFI::osfi-python
    PROPERTY INTERFACE_INCLUDE_DIRECTORIES)
list(GET OSFI_PYTHON_DIR 0 OSFI_PYTHON_DIR) # Get 1st element
install(DIRECTORY "${OSFI_PYTHON_DIR}"
    DESTINATION "lib/OSFI"          # Will create a OSFI/Python/ dir
    FILES_MATCHING PATTERN "*.py") # Avoid copying *.pyc, etc.
```

The above setup ensures that the library files (in this case, OSFI-Python) are copied to the "lib" folder of the install directory, while the module itself is copied with execution permissions to the "bin" folder. The code for Matlab is exactly the same, performing the adequate substitutions of "osfi-python" for "osfi-matlab" and ".py" for ".m".

In order for a module to be executable, it needs to be able to find the library, so in the case of Python, the following code would need to be inserted at the start of the module. It first tries to load OSFI from the default Python path, and if that fails it tries to load it from the module-relative path "../lib/OSFI/Python", which is where the above CMake code will have installed it.

```
#!/usr/bin/env python

from os.path import dirname, realpath, join
import sys
try: # Try to load something from OSFI in case it is in sys.path
    import Logger
except ImportError:
    # If not, add the known path $moduledir/../lib/OSFI to sys.path
    cur_fdir = dirname(realpath(__file__))
    osfi_dir = realpath(join(cur_fdir, '..','lib','OSFI','Python'))
    sys.path.append(osfi_dir)
    try: # And try again
        import Logger
    except ImportError as e: # Give up
        raise ImportError("Cannot find the OSFI library in the"
                          "Python path, or at " + osfi_dir)
import other_things
# Rest of the module code
```

If the above steps are followed, the Python/Matlab module will always be able to access the OSFI library, and so it will be able to be executed from the command line without adding any special variables or settings to the environment:

```
$ ./bin/mymodule.py
```

The Matlab version of the module code adaptation is similar in both methodology and implementation, although the way of calling the module is E2E-ICD implementation dependent – check the corresponding manual for details.

```
function testModel (configurationParameters, inputs, outputs)
% Check that OSFI is in the Matlab path. If not, add the known path
% $moduledir/../lib/OSFI/Matlab to it and check again.
if ~exist('ConFM', 'class')
    cur_fdir = fileparts(mfilename('fullpath'));
    osfi_dir = fullfile(cur_fdir, '..', 'lib', 'OSFI', 'Matlab');
```

```
    addpath(osfi_dir);
    if ~exist('ConFM', 'class')
        error('mymod:depmissing', ['Cannot find the OSFI library' ...
                  'in the Matlab path, or at %s'], osfi_dir)
    end
end
% Rest of the module code
```

In order to use the "installed package" approach for OSFI Python, first the built-package shall be generated from the sources. Wheels [PEP427] are the new standard of Python distribution and are intended to replace eggs. A wheel package can be generated as follows:

```
$ cd osfi/include/Python
$ python setup.py bdist_wheel clean --all
```

The wheel package will be generated into a new "*dist"* subdirectory and will have the following naming convention *"OSFI-{version}-{python}-{abi}-{platform}.whl*", where possible options for the various tags can be checked in [PEP427].

Once generated, the wheel package can be installed into the system, or into any desired virtual environment, by simply using the *pip* Python package manager:

```
$ pip install dist\OSFI-{version}-{python}-{abi}-{platform}.whl
```

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.

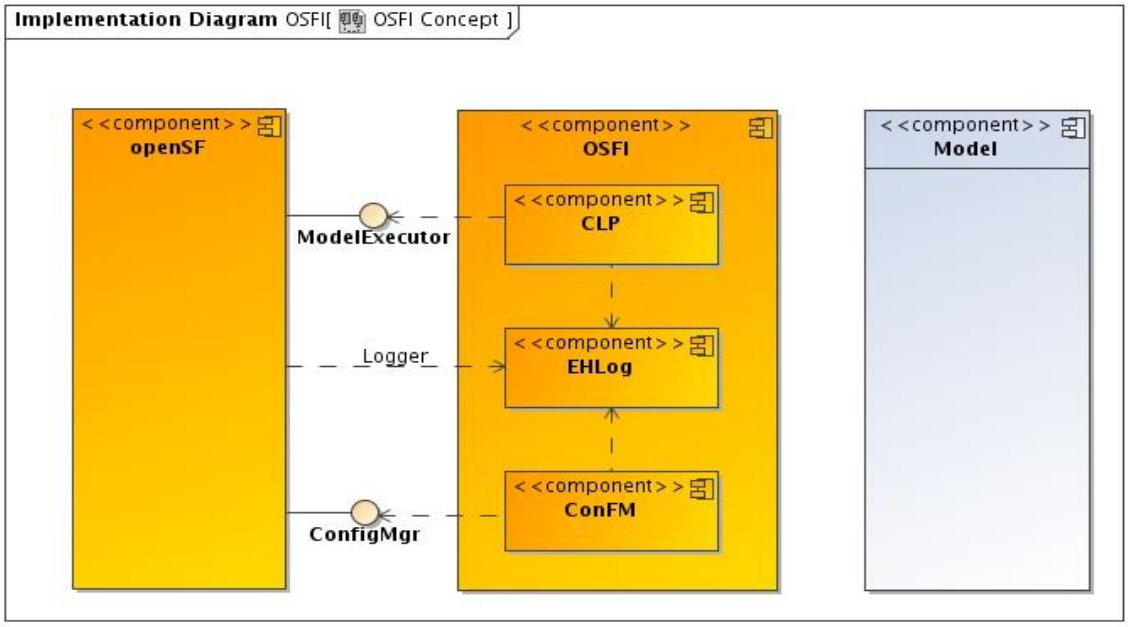

Figure 2: Relationship with openSF and modules

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.

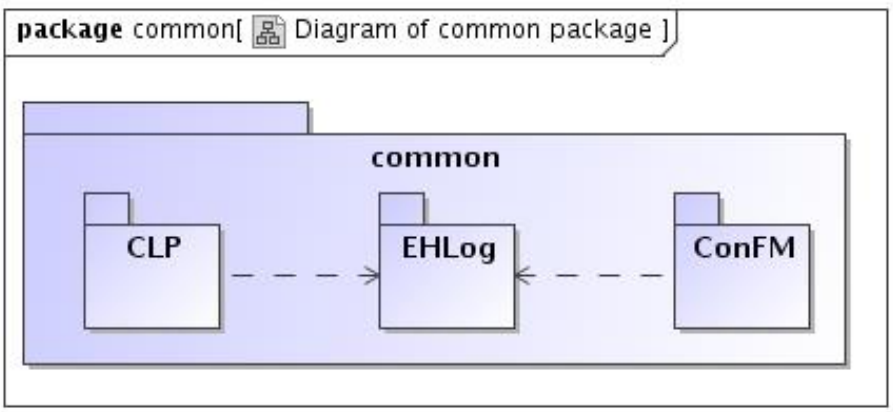

Figure 3 : OSFI common packages

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 natively.

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. In it, the general API of OSFI is described using concepts such as "a list of X", which are later determined in each language-specific section to the concrete implementation provided, e.g. List<X> in Java, a cell array in Matlab, etc.

The handling of error conditions is also language specific: when this section mentions that certain condition "results in an error", that will in general log an error to the OSFI logger, but other extra effects could be:

- In C++, Java or Python, raise an exception or simply return a certain flag value.
- In C, return an error code or a null pointer.
- In Fortran, both of the above depending on the presence or not of the STAT argument

In general, functionalities provided by each module will be described as a series of "cards" in the following format:

Operation name: (inputs…) → (outputs…)	
Component	Name of the (sub-)component e.g. CLP or ConFM.Parameter
Generic name	Name that in most cases will the part of the name of the actual function implementing the operation.
Precondition	Conditions that need to hold before calling this operation. Does not include conditions only on the inputs, which are described in their own slot.
Inputs	List of inputs for the operation, and conditions that they need to hold. For object-oriented languages, it does not include the "this" or "self" object handle, which is listed in preconditions.
Outputs	List of outputs from the operation, and conditions that are verified by them. For object-oriented languages, it does not include the "this" or "self" object handle, changes to which are listed in postconditions.
Postconditions	Conditions that are verified after returning from the call.
Errors	List of error conditions. Unless otherwise specified here, what OSFI does in an error is language-specific. Furthermore, "none" here does not preclude language-specific problems such as out-of-memory errors in list-related operations, etc.
Notes	Any extra comments.

4.2.1. Command Line Parser (CLP)

According to section 2.1.2 of E2E-ICD, a module shall be invoked with three-command arguments: two configuration files separated by commas., multiple (at least one) input files separated by commas and multiple (at least one) output files separated by commas.

This package provides the following functionalities:

Build CLP: args → ()	
Component	CLP
Generic name	Build CLP, usually class constructor, or "create"
Precondition	None
Inputs	Command-line arguments. Depending on the language, it may be a list of strings, three strings, or even optional (causing the program command line arguments to be read by OSFI).
Outputs	None per se (usually, the constructed CLP object)
Postconditions	The CLP is properly initialized; functions to query the lists of C/I/O files can be called and will return without errors.
Errors	In general, if the format does not match the description of 2.1.2 of E2E-ICD. Possible problems are: the number of command line arguments is not 3, the number of configuration files is not 2, the name of a file contains an invalid character, etc.

Get the list of C/I/O files: () → files	
Component	CLP
Generic names	getConfigFiles, getInputFiles, getOutputFiles
Precondition	CLP has been built
Inputs	None (or, if a single function, the type of files)
Outputs	List of files of the required type
Postconditions	None
Errors	None directly.
Notes	In some languages, may be implemented as a pair of functions to get the number of elements and the individual list item by index.

Table 7 summarizes the functionalities that this package shall provide (Function column) and the current state in terms of implementation depending on the programming language (a red cell means that the function is missing). The column labeled as E2E-ICD contains the functions that are strictly needed taken into account the description given by the document.

In addition, Table 8 shows some functions (blue cells) that are currently implemented by some languages and accessible to the user, but are not needed to comply with the requirements derived from E2E-ICD. Thus, they may be deprecated and later eliminated from the public interface.

Table 7: Functions of the CLP package

Function	E2E-ICD	C++	C	Fortran	Java	Matlab	Python	F77	IDL
Parse command line arguments ("constructor")									
Get (full) list of C/I/O files									

Table 8: Additional functions of the CLP package

Function	E2E-ICD	C++	C	Fortran	Java	Matlab	Python	F77	IDL
Legacy: get single configuration file									
Query (only) number of C/I/O files									
Tokenize string as list of C/I/O files									
List of C/I/O files to string format									
Check that a file name is valid									
Legacy: get single input file									
Legacy: get single output file									

4.2.2. Logger (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. 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.

Additionally, the OSFI implementation introduces extra functions. For example, a function to terminate the execution and exit with a certain exit code is present in all implementations even though it's not a strict requirement. Also, other types of messages with similar formats are supported, like "quality messages".

Show info/warning/error/debug message: text → ()	
Component	Logger
Generic names	info, error, warning, debug
Precondition	Logger has been initialized (if necessary in the language)
Inputs	Message text
Outputs	None
Postconditions	The given message is written to the log output, formatted according to the corresponding E2E-ICD message type.
Errors	None directly.
Notes	If the message type is *debug*, the message is only written if the debug mode is activated. Implementations **may** allow formatting of extra data in a language-specific way, e.g. the C implementations have API similar to "printf".

Show progress notice: (currentStep, totalSteps) → ()	
Component	Logger
Generic name	progress
Precondition	Logger has been initialized (if necessary in the language)
Inputs	Integers representing the number of the current work step and the total amount of steps to be done, respectively. It must hold that 0 ≤ currentStep ≤ totalSteps.
Outputs	None
Postconditions	The given message is written to the log output, formatted according to the E2E-ICD "progress" message type.

Errors	None directly.
Notes	-

Finish execution: (exitCode) → ()	
Component	Logger
Generic names	finishExecution
Precondition	Logger has been initialized (if necessary in the language)
Inputs	Integer exit code to be passed to the language runtime / OS.
Outputs	N/A
Postconditions	This function does not return normally, instead attempting to terminate execution of the module through language-specific means.
Errors	None directly.
Notes	**OSFI extension**

Show quality message: (name, value) → ()	
Component	Logger
Generic names	qualityReport
Precondition	Logger has been initialized (if necessary in the language)
Inputs	Quality indicator name: string Quality indicator value: number or string
Outputs	None
Postconditions	The given message is written to the log output, formatted according to a custom message type in the vein of E2E-ICD formats.
Errors	None directly.
Notes	**OSFI extension**

Table 9 summarizes the functionalities that this package provides, along with the current state in terms of implementation depending on the programming language: a red cell means that the function is missing, a blue one that it is available.

The column labeled as E2E-ICD contains the functions that are strictly needed taken into account the description given by the document. It can be seen that the function finish execution is not defined in the E2E-ICD, but since it has been implemented in all the languages and is useful, it has been kept as a function to be provided by the package.

In addition, Table 10 shows some functions (blue cells) that are currently implemented by some languages and accessible to the user, but are not needed to comply with the

requirements derived from E2E-ICD. Thus, they may be deprecated and later removed from the public interface.

Table 9: Functions of the EHLog package

Function	E2E-ICD	C++	C	Fortran	Java	Matlab	Python	F77	IDL
Show I/W/E/D msg									
Show progress									
Finish execution									

Table 10: Additional functions of the EHLog package

Function	E2E-ICD	C++	C	Fortran	Java	Matlab	Python	F77	IDL
Quality reports									
Format I/W/E/D msg with extra data									
Query debug mode									
Set debugging mode									
Query color output									
Redirect output to a file (instead of out)									
Get OSFI version									

4.2.3. Configuration File Manager (ConFM)

This group of functions deals with the configuration files. First, user code must read a configuration file, and then the parameters inside can be accessed.

4.2.3.1. Parsing and validating configuration files

Read configuration file: xmlPath → ()	
Component	ConFM.ParamReader
Generic names	Constructor, read/create
Precondition	None
Inputs	Path to an existing file formatted according to E2E-ICD.
Outputs	None, possibly a handle to the constructed object

Postconditions	The configuration file is parsed and functions to access parameters may be called.
Errors	If the file cannot be found or read as XML. If the file is valid XML but is malformed according to E2E-ICD.
Notes	-

Validate configuration file against XSD: xsdPath → valOk	
Component	ConFM.ParamReader
Generic names	validateAgainst
Precondition	ParamReader has been initialized successfully.
Inputs	Path to an existing file describing a XSD 1.0 schema.
Outputs	One flag value (e.g. Boolean "true") if the file passes validation, a different one if the schema can be read but the file does not validate.
Postconditions	-
Errors	If the file cannot be found or parsed as an XSD schema.
Notes	Some languages also support an alternative form that takes no arguments: "validateAgainstInternalSchema". It works as described but the path to the schema is taken from the XML file itself as noted in an *xsi:[noNamespace]SchemaLocation* attribute.

4.2.3.2. Finding and accessing parameters

The parameters are identified by its complete *path*, which is obtained concatenating the name of the groups that contain the parameter (from least to most nested) and the parameter name, all separated by dots. Note that the root element of the XML file does *not* count as a group, so its parameter children have paths without any dots.

The following functions are related with parameter access. If the file that contains them has not been validated previously and/or if the format of the parameters declared is not correct, these functions may raise errors themselves or defer them to a later time (e.g. when trying to retrieve a parameter value).

Check parameter existence by path: path → paramExists	
Component	ConFM.ParamReader
Generic names	existParameter
Precondition	ParamReader has been initialized successfully.
Inputs	String describing a parameter path
Outputs	True if a parameter under that name has been parsed, false otherwise.

Postconditions	-
Errors	None directly.
Notes	-

Get parameter by name: path → handle	
Component	ConFM.ParamReader
Generic names	getParameter
Precondition	ParamReader has been initialized successfully.
Inputs	String describing a *full* parameter path
Outputs	Some kind of handle that may later be used to query information about the parameter.
Postconditions	-
Errors	If a parameter by that name does not exist.
Notes	Depending on the language, the "handle" may be an actual copy of the parameter data, or a reference to ParamReader-held data.

Get parameters by group: path → list of handles	
Component	ConFM.ParamReader
Generic names	getParameters
Precondition	ParamReader has been initialized successfully.
Inputs	String describing a *partial* parameter path.
Outputs	List of handles that may later be used to query information about the parameters.
Postconditions	The list is always valid, although it may be empty. For every parameter p, if its path begins by the given argument, it is contained in the list. For every handle in the list, the path of the parameter it refers to begins by the given argument.
Errors	-
Notes	Depending on the language, the "handles" may be actual copies of the parameter data, or references to ParamReader-held data. An extra version may be provided that does not take any arguments and simply returns handles to all parameters read from the file.

Furthermore, some versions include a function to print information about either a single parameter, or all read from the file. It usually 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 (for non-ARRAY parameters)

This set of functions is in charge of reading the value of the parameter. They shall support the different element types in section 2.2.6.2 of E2E-ICD. Currently, type TIME is generally unsupported because it was introduced in a review of the standard document very close to the release of this version of OSFI.

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) many different functions will be available, covering each data combination of type and element types. However, for languages where the type of the data is known at run time (Matlab, Python and IDL), these capabilities could be encapsulated in a single generic function.

Depending on the element type and structure of the value to read, there are certain details that must be considered that are described in the following sections. Note that these functions may also be applied to an ARRAY parameter; depending on the specific language they may return a nested structure as described in the following section, or if the getVector(T) functions are used the array is first flattened in a depth-first fashion before parsing the elements and returning a 1D list.

Get non-ARRAY parameter value: paramHandle → value	
Component	ConFM.Parameter
Generic names	All languages: get(T)Value, getVector(T), getMatrix(T) Dynamic languages may also have a getValue / getParsedValue
Precondition	-
Inputs	Valid parameter handle
Outputs	Parsed parameter value according to the declared or chosen element type and data structure. For vectors, a list of values. For matrices, a list of lists or other language-specific concept representing a matrix.
Postconditions	-
Errors	If the value is not formatted according to E2E-ICD. If the value does not have the declared dimensionality. If any of the individual values cannot be parsed as the detected or chosen element type.
Notes	An extra version may be provided where the user is allowed to override the element type declared in the XML. If getVector(T) is applied to an ARRAY parameter, or a node thereof, the elements of the chosen subtree will be enumerated depth-first, and the result will be returned as a list of values. OSFI does not apply the limit of 255 characters for strings in E2E-ICD or check the string contents for non-alphanumeric characters.

4.2.3.4. Access parameter values (for ARRAY parameters)

Two different approaches are possible when accessing ARRAY parameters:

- The "getArrayValue" convention, where the implementation provides functions with an interface similar to the getVector(T) functions used for non-ARRAY parameters, but which take an additional argument representing indices into the parsed value of the parameter. Depending on the implementation, the return value may be restricted to a single vector (so only leaf nodes produce values) or an object representing the parsed value of (part of) the full structure.
- The "getRootNode" convention, where the library provides access to the structure of the unparsed ARRAY parameter. User code may navigate this structure, indexing into it to examine the number and value of its children. Upon reaching a leaf node, use code can ask for the parsed value of this node. Optionally, an implementation may also provide for a way to parse (part of) the full structure, returning a different type with a similar API that allows navigating the parsed structure and extracting the values.

Each implementation of OSFI provides at least one of these approaches. As mentioned in both descriptions, they may provide an extra optional feature which is a way to represent the parsed value of (part of) the full ARRAY structure. This is easier to do in dynamic languages, but e.g. OSFI-Java does provide this feature too.

Get ARRAY leaf node value: (paramHandle, nodeIndex) → value	
Component	ConFM.Parameter
Generic names	getArrayValue, getLeafVector(T)
Precondition	Parameter is of type ARRAY
Inputs	Valid parameter handle of type ARRAY. List of indices into the structure of the ARRAY
Outputs	Parsed parameter value according to the declared or chosen element type and data structure. For vectors, a list of values. For matrices, a list of lists or other language-specific concept representing a matrix.
Postconditions	-
Errors	If the value is not formatted according to E2E-ICD. If the indices do not reach a leaf node (containing data) or the node contents do not match its declared dimensionality. If any of the individual values cannot be parsed as the detected or chosen element type.
Notes	An extra version may be provided where the user is allowed to override the element type declared in the XML. OSFI does not apply the limit of 255 characters for strings in E2E-ICD or check the string contents for non-alphanumeric characters.

Get ARRAY root node: paramHandle → arrayNodeHandle	
Component	ConFM.Parameter
Generic names	getArrayValue, getLeafVector(T)
Precondition	Parameter is of type ARRAY
Inputs	Valid parameter handle of type ARRAY.
Outputs	Handle to an ArrayNode object representing the unparsed structure as defined in the XML.
Postconditions	-
Errors	-
Notes	-

Get type of array node: arrayNodeHandle → nodeType	
Component	ConFM.ArrayNode
Generic names	isLeaf, isDataNode, etc.
Precondition	-
Inputs	Valid array node handle.
Outputs	Flag value representing either a data/leaf node (directly containing values) or internal node (containing other array nodes reproducing the XML structure).
Postconditions	-
Errors	-
Notes	-

Get array node children: arrayNodeHandle → arrayNodeHandles	
Component	ConFM.ArrayNode
Generic names	getSubNodes
Precondition	-
Inputs	Valid array node handle representing an internal node.
Outputs	List of handles to ArrayNode objects representing the children of this node in the structure defined in the XML.
Postconditions	-
Errors	If the node is a data/leaf node that cannot contain child nodes.
Notes	-

Get array node data: arrayNodeHandle → values	
Component	ConFM.ArrayNode
Generic names	getData
Precondition	-
Inputs	Valid array node handle representing a data/leaf node of either an unparsed or parsed node structure.
Outputs	If the node represents unparsed values, returns an object that contains both the number of declared values (in the XML) and the unsplit, unparsed string. If the node represents parsed values, returns a list of parsed values of the right size.
Postconditions	-
Errors	If the node is an internal node (and does not contain data)
Notes	-

Parse value of array node: arrayNodeHandle → value	
Component	ConFM.Parameter
Generic names	getArrayValue, getLeafVector(T)
Precondition	Parameter is of type ARRAY
Inputs	Valid array node handle representing an unparsed data/leaf node.
Outputs	Parsed parameter value according to the declared or chosen element type and data structure. For vectors, a list of values. For matrices, a list of lists or other language-specific concept representing a matrix.
Postconditions	-
Errors	If the value is not formatted according to E2E-ICD. If the indices do not reach a leaf node (containing data) or the node contents do not match its declared dimensionality. If any of the individual values cannot be parsed as the detected or chosen element type.
Notes	An extra version may be provided where the user is allowed to override the element type declared in the XML. OSFI does not apply the limit of 255 characters for strings in E2E-ICD or check the string contents for non-alphanumeric characters. An extra version may be provided that parses the full structure from this node downwards, instead

4.2.3.5. Query parameter attributes

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

Get number of dimensions: paramHandle → ndims	
Component	ConFM.Parameter
Generic names	getNdims
Precondition	-
Inputs	Valid parameter handle
Outputs	Number of dimensions of a parameter. Will be zero for scalars, 1 for vectors and 2 for matrices. For ARRAY-typed parameters, the number of dimensions of the rectangular envelope of the actual shape.
Postconditions	-
Errors	None directly.
Notes	-

Get dimensions: paramHandle → dims	
Component	ConFM.Parameter
Generic names	getDims
Precondition	-
Inputs	Valid parameter handle
Outputs	List of dimensions for a parameter. For scalars it is empty, while it has one element for vectors and two for matrices: (cols, rows). For arrays it represents the rectangular envelope of the actual shape.
Postconditions	-
Errors	None directly.
Notes	-

Get description/units/maximum/minimum: paramHandle → attrVal	
Component	ConFM.Parameter
Generic names	get(Description, Units, Max, Min)
Precondition	-
Inputs	Valid parameter handle
Outputs	String, possibly empty
Postconditions	-

Errors	None directly.
Notes	-

Get data element type: paramHandle → elType	
Component	ConFM.Parameter
Generic names	getElementType
Precondition	-
Inputs	Valid parameter handle
Outputs	Token representing an element type, may be a value from an enumerated type if the language supports such a concept.
Postconditions	-
Errors	None directly.
Notes	See section 2.2.6.2 of E2E-ICD

Query parameter structure: paramHandle → isArray	
Component	ConFM.Parameter
Generic names	isArray
Precondition	-
Inputs	Valid parameter handle
Outputs	Flag representing whether or not a parameter has the ARRAY structure type.
Postconditions	-
Errors	None directly.
Notes	This returns a false flag value for MATRIX parameters, since they cannot sport a tree-like structure with different sizes.

Get unparsed value: paramHandle → rawValue	
Component	ConFM.Parameter
Generic names	getRawValue
Precondition	-
Inputs	Valid parameter handle
Outputs	String representing the unparsed value of the parameter, as read, without any XML syntax. For MATRIX parameters, the values in each row are joined, with

	matrix elements appearing in row-major ordering. For ARRAY parameters, the returned string is language-specific, but it should represent the structure of the parameter.
Postconditions	-
Errors	None directly.
Notes	-

4.2.3.6. Files and Folders

One function is 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.

4.2.3.7. Summary tables

The analysis of section 2.2.6.2 of E2E-ICD and the current implementation of OSFI libraries reveals a set of functions that are needed to read the parameters that the users can define, which are described in Table 11. The first two columns of the table list the functions and group them according to their functionality.

The blue cells of the third column of Table 11 shows the functions that are strictly needed to comply with E2E-ICD according to the types of parameters defined in section 2.2.6.2. For instance, the functions related with accessing vectors appear in red because the complex type vector is not defined in E2E-ICD. Other functions that are red in Table 11 are the ones related with reading arrays and matrices of element type folder, since its definition is also omitted in E2E-ICD.

The rest of the columns of Table 11 refer to an implementation of the ConFM package on a certain programming language. It shall be remarked that the C++, C, Fortran 77 and Fortran implementations do not have function specifically dedicated to access arrays, although this functionality is supported by the functions to read matrices. Other remarkable gap in Table 11 is related to the lack of functions in Fortran 77 and Fortran to get the attributes of the parameters. In addition, although it is very useful, some implementations and the E2E-ICD do not offer the possibility to access multiple parameters at the same time (either under the same group or all the parameters in the file).

In addition, Table 12 shows some functions (blue cells) that are currently implemented by some languages and accessible to the user but are not needed to comply with the requirements derived from E2E-ICD. Thus, they should be eliminated from the public interface. It is especially remarkable the case of the C++ language, which has a wide set of functions that should not be available to the user (such as functions to edit parameters, get vectors of nodes or split strings).

Table 11: Functions of the ConFM package

Group		Function	E2E-ICD	C++	C	Fortran	Java	Matlab	Python	F77	IDL
Parse file		Load file			[1]					[1]	
		Validate against XSD							[2]		
Access params		By name									
		By group									
		Get full list									
		Existence									
Get Parameter Value	Scalar	INTEGER									
		FLOAT									
		BOOLEAN									
		STRING									
		FILE									
		FOLDER		[3]	[3]	[3]	[3]	[3]	[3]	[3]	[3]
		TIME									
	Vector	INTEGER									
		FLOAT									
		BOOLEAN									
		STRING									
		FILE									
		FOLDER		[3]	[3]	[3]	[3]	[3]	[3]	[3]	[3]
		TIME									
	Matrix	INTEGER									
		FLOAT									
		BOOLEAN									
		STRING									
		FILE									
		FOLDER		[3]	[3]	[3]	[3]	[3]	[3]		[3]
		TIME									
	General array	INTEGER									
		FLOAT									
		BOOLEAN									
		STRING									
		FILE									
		FOLDER		[3]	[3]	[3]	[3]	[3]	[3]	[3]	
		TIME									

[1] Only a single configuration file may be loaded at once; calling the loading function a second time unloads the previously loaded file

[2] Only if the lxml library is available; otherwise the xml.etree package provided by the standard CPython library is used instead and calling validation functions raises an exception.

[3] The parameter type FOLDER is recognized; functions used to access it are the same as for FILE.

Group	Function	E2E-ICD	C++	C	Fortran	Java	Matlab	Python	F77	IDL
Query parameter attributes	# dimensions									
	Dimensions									
	Full path									
	Description									
	Units									
	Max									
	Min									
	Element type									
	Is array									
	Raw value									
	Print									
Files	File exists									

Table 12: Additional functions of the ConFM package

Function	E2E-ICD	C++	C	Fortran	Java	Matlab	Python	Fortran 77	IDL
Set value									
Tokenize string according to type									
Get complex type (equivalent to "is MATRIX")									
Get extended attribute									
Set extended attribute									
Add extended attribute									
Get path from a Parameter instance									

4.3. Language-specific interfaces

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

OSFI-C++ code is written to comply with the [C++11] standard, although compilation settings may be set for C++14 if Xerces-C requires so. A main header file OSFI.h is provided which exposes all public API, but module-specific headers are also provided.

The implementation of OSFI-C++ is object-oriented, with both the CLP and ConFM modules implemented using classes: CLP, ParamReader and Parameter are the main ones. The Logger module is mainly function-based and stores any state globally. In general, errors are communicated through exceptions, although some methods just log errors to the OSFI log stream and return token values instead.

General description:

- All classes are directly in the global namespace. In future versions, it is possible that a namespace will be introduced.
- Types in E2E-ICD are mapped to their reasonable equivalents: INTEGER to int, FLOAT to double, BOOLEAN to bool and the STRING, FILE and FOLDER types to std::string. In future versions, it is possible that the arithmetic types will be mapped to a different type (e.g. std::int32_t).
- 1-D "list" types are generally mapped to std::vector<T> except in the CLP class. Matrices (described in the general API as "lists of lists") are represented by a custom DynamicArray<T> class wrapping a vector of vectors.
- Access to ARRAY-typed parameters is twofold, implementing both the "getRootNode" and the "getArrayValue" approaches described in §4.2.3.4. The latter is implemented as getVectorT functions that take the desired slice indices.

Known issues:

- Currently some OSFI headers import the std namespace ("using namespace std;") but this is considered bad form and will be removed in the future. Thus, for future compatibility, do not depend on OSFI importing any namespaces.
- Large objects are frequently returned directly (lists, vectors). While the penalty in performance may be reduced by compiler optimizations (RVO, NRVO) and has also recently been ameliorated by move semantics in C++11, user code should consider this fact when making use of such methods.
- Const correctness of the code is an issue, and it is difficult to effectively utilize const references to OSFI objects. This is likely to be improved in the future.

4.3.1.1. CLP

The CLP module API is provided by header `CLP/CLP.h`. The implementation takes the form of a single class "CLP" which parses command line arguments as passed. No global state is stored and thus multiple instances can coexist. Furthermore, all parsing is done by the constructor, and accessor functions only return copies of stored data.

Table 13 details the interface of the CLP module in OSFI-C++.

4.3.1.2. EHLog

The EHLog module API is provided by header `EHLog/Logger.h`. The implementation provides the functions as static member functions of a fake class named Logger. Extra data-formatting is implemented by returning C++ streams that the user can append to.

Relevant status variables (debug/color) are initialized at first execution of an output, and stored as global data from them on. Furthermore, functions operate on the global streams cout and cerr. Thus, thread safety is not guaranteed in these functions.

Table 14 details the interface of the EHLog module in OSFI-C++.

4.3.1.3. ConFM

The ConFM module API is provided by four headers under the `ConFM` folder: `ArrayNode.h`, `DynamicArray.h`, `Parameter.h` and `ParamReader.h`. Each defines the class of the same name, and the latter header transitively includes all four.

The ParamReader class is the main access point to the module interface. Each instance is independent and holds no global state, so several instances can be kept (e.g. for the global and local configuration files). The instance holds ownership of and provides access to a set of Parameter instances, which are likewise independent of any other instance. However, thread safety is not guaranteed in any of the functions because they may call the Logger functions to report errors.

Table 15 details the interface of the ConFM module in OSFI-C++.

Table 13: Functions of the CLP module in C++

General operation	C++ Prototype	Notes
Parse command line arguments ("constructor")	`CLP:: CLP(int argc, char* argv[])`	The inputs are those provided to the C++ main() function.
Get (full) list of C/I/O files	`list<string> CLP:: getConfFiles()` Same signatures: **getInputFiles** and **getOutputFiles**.	Should be const. If no file of a type is provided, they return an empty list.
Legacy: get single configuration file	`string CLP:: getConfFile()`	Should be const. If called in a case with two configuration files, returns the unparsed string ("file1,file2").

Table 14: Functions of the EHLog module in C++

General operation	C++ Prototype	Notes
Show I/W/E/D message	`static void Logger:: info(string msg)` Same signatures: **warning**, **error** and **debug**.	The user should not introduce newlines in the string, because it will break the output format.
Extension: format I/W/E/D message with extra data	`static ostream& Logger:: getInfoStream()` Same signatures: **getWarningStream**, **getErrorStream**, **getDebugStream**.	The user should not introduce newlines in the output given to those streams, because it will break the output format.
Show progress indication	`static void Logger:: progress(int, int)`	No validation is performed.
Finish execution	`[[noreturn]] static void Logger:: finishExecution(int exitCode)`	Calls std::exit with the given value as exit code.

General operation	C++ Prototype	Notes
Extension: show quality report	`static void Logger::quality(string name, double v)` `static void Logger::quality(string name, string v)`	The user should not introduce newlines in the strings, because it will break the output format.
Extension: format quality report	`static ostream& Logger:: getQualityStream()`	The user should not introduce newlines in the output given to those streams, because it will break the output format.
Query debug and colored output.	`static bool Logger:: isColored()` Same signature: **isDebugging**	Status set from environment variables DEBUG_MODE and OSFI_LOG_COLOR on first query.

Table 15: Functions of the ConFM module in C++

General operation	C++ Prototype	Notes
Load config file	`ParamReader:: ParamReader (const string& xmlFile)`	Throws std::exception if the file cannot be parsed.
Validate against XSD	`bool ParamReader:: validateAgainst (const string& xsdFile) const`	Throws std::invalid_argument if the schema cannot be parsed.
Extension: validate against internal schema	`bool ParamReader:: validateAgainstInternalSchema() const`	Returns failure if the schema cannot be found or parsed.
Get parameter by full path	`Parameter ParamReader:: getParameter(string)` `Parameter& ParamReader:: getParameterRef (const string&)`	The first form returns a "dummy" parameter if the name is not found; the second throws std::invalid_argument. The lifetime of its return value matches that of the ParamReader instance.

General operation	C++ Prototype	Notes
Get parameters by partial path	`vector<Parameter> ParamReader:: getParameters (string)`	Returns an empty vector if no parameter matches. Copies Parameter objects
Get all parameters	`t_params_map ParamReader:: getParameters()`	The map key is the full path, which is not accessible from within each Parameter item. t_params_map is an alias to map<Parameter,string,X> with X a custom comparer. Copies Parameter objects
Query existence	`bool ParamReader:: existParameter (string)`	-

Get parameter parsed value (scalar, vector, matrix)	`#V# Parameter:: get#T#Value()`
	`vector<#V#> Parameter:: getVector#T#()`
	`DynamicArray<#V#> Parameter::getMatrix#T#()`
		#T# is one of Int, Double, Boolean, String or File, and #V# is the corresponding type (int, double, bool, string, string).
		On parsing error, a log message is emitted and a default value is returned (see docs). If the getVector#T# functions are used on an ARRAY parameter, it is flattened to 1D.

Extension: query file existence	`DynamicArray<bool> Parameter::fileExists()`	For FILE parameters. If the parameter is a scalar or vector, a 1x1 or single-row matrix is returned, respectively.

Query parameter attributes	`int Parameter:: getNdims()`
	`const vector<int>& Parameter:: getDims() const`
	`Parameter::ElementType Parameter:: getElementType() const`
	`string Parameter::getName()`
	Same sig.: **getDescription**, **getUnits**, **getMax**, **getMin**, **getType**
		Parameter::ElementType is a C++11 scoped enum ("enum class") with the E2E-ICD simple types, that is, it does not represent types ARRAY or MATRIX.

General operation	C++ Prototype	Notes

ARRAY access – getArrayValue API	`int Parameter:: getDims(vector<int>)`
	`bool Parameter:: isLeaf(vector<int>)`
	`vector<#V#> Parameter:: getVector#T# (vector<int>)`
		The vector represents the index of the desired slice. Slices of a parameter with dimensionality d must have at most d−1 indices, because a vector is returned for the last dimension.
		The getVector#T# functions do **not** flatten sub-elements: they only return non-empty values for leaf nodes.

ARRAY access – ArrayNode API	`const ArrayNode& Parameter:: getRootNode() const`
	`ArrayNode Parameter:: getNode(vector<int>)`
	`vector<#V#> ArrayNode:: getVector#T#() const`
	`int ArrayNode:: getDegree() const`
	`const vector<string>& ArrayNode:: getElements() const`
	`const vector<ArrayNode>& ArrayNode:: getChildren() const`
	`bool ArrayNode:: isLeaf()cons`
		For getNode, indices must have at most one dimension less than the parameter, since the last dimension (leaf node) contains a vector itself.
		getElements returns an empty vector for non-leaf nodes.
		getChildren returns an empty vector for leaf nodes.
		getDegree returns the size of the vector that is not empty.
		The getVector#T# functions do **not** flatten sub-elements: they only return non-empty values for leaf nodes.

4.3.2. ANSI C Programming Language

OSFI-C code is written to comply with the [C99] standard formally, although the design of the interface is not modern. A single header file `OSFIC.h` is provided which exposes all public API.

The implementation of OSFI-C is not object-oriented: it is implemented as a wrapper around OSFI-C++ which keeps **a single instance of the CLP and ConFM classes** live at any point. In general, errors are communicated through return codes, although some methods just log errors to the OSFI log stream and return token values instead.

General description:

- Types in E2E-ICD are mapped to their reasonable equivalents: INTEGER to int, FLOAT to double, BOOLEAN to _Bool (bool with stdbool.h) and the STRING, FILE and FOLDER types to character arrays. In future versions, it is possible that the arithmetic types will be mapped to a different type (e.g. int32_t).
- 1-D "list" types are generally mapped to arrays except in the CLP class. Matrices (described in the general API as "lists of lists") are represented by linear (1D) arrays containing the matrix elements in row-major order.
- Access to ARRAY parameters is provided using the the "getArrayValue" approach described in §4.2.3.4. It is implemented as getLeafVectorT functions that take the desired slice indices.

Known issues:

- Memory allocation must be performed by the user, leading to possible buffer overflows if the sizes indicated by the API are not respected. A series of constants are available for the user to perform these allocations. In general, user code should preallocate string buffers to a size of MAX_x+1, where x is the constant in question, since the API expects that the buffer may hold MAX_x chars plus the C string terminator character.

4.3.2.1. CLP

All CLP functions are prefixed by "osfiCLP" except for the "general destructor" osfiCommonClose that destroys the held CLP and ConFM instances. Table 16 details the interface of the CLP module in OSFI-C.

4.3.2.2. EHLog

All EHLog functions are prefixed by "osfiLogger", and they relay calls to the related C++ versions. Thus, status variables (debug/color) are initialized at first execution of an output and stored as global data from them on. Furthermore, functions operate on the global C++ output and error streams, which may or may not be synchronized with the C conterparts. Thread safety is not guaranteed in these functions.

Table 17 details the interface of the EHLog module in OSFI-C.

4.3.2.3. ConFM

All ConFM functions are prefixed by "osfiConFM" except for the "general destructor" osfiCommonClose that destroys the held CLP and ConFM instances.

The interface does not provide individual functions to query all the parameter attributes, but it does provide a structure to read them all at once (struct osfiParameter and function osfiConFMGetParameter). However, as mentioned in the general considerations and known issues, it is the responsibility of the user to allocate and free memory for such a structure.

Table 18 details the interface of the ConFM module in OSFI-C.

Table 16: Functions of the CLP module in C

General operation	C Prototype	Notes
Parse command line arguments ("constructor")	`int osfiCLP(int argc, char* argv[])`	Returns FALSE (zero) in case of error, TRUE (nonzero) otherwise. If the call is successful and this function had already been called before, the stored parsed data is replaced.
Get (full) list of C/I/O files	`void osfiCLPGetConfFiles (char *confFiles[], int *noFiles)` Same signatures: **osfiCLPGetInputFiles** and **osfiCLPGetOutputFiles**.	If the buffer is NULL, it is not touched and only the number of files is returned in *noFiles. Otherwise, the function **assumes that the number of slots available is enough** (*noFiles is not checked on entry) and that every slot in the buffer is preallocated to a capacity of MAX_LENGTH_FILE_NAME+1.
Legacy: get single configuration file	`void osfiCLPGetConfFile (char *fileName, int *length)`	The buffer must be preallocated to MAX_LENGTH_FILE_NAME+1. The actual length of the file name is returned in *length.
Language-specific: free OSFI-C resources	`int osfiCommonClose()`	Destroys the held instances of **both** the C++ CLP and ParamReader objects. Returns zero on failure, nonzero on success.

Table 17: Functions of the EHLog module in C

General operation	C Prototype	Notes
Show I/W/E/D message (also, **extension**: format I/W/E/D message with extra data)	`void osfiLoggerInfo(char *format,...)` Same signatures: **osfiLoggerInfo**, **osfiLoggerError** and **osfiLoggerDebug**.	The user should not introduce newlines in the output given to those streams, because it will break the output format
Show progress indication	`void osfiLoggerprogress(int, int)`	No validation is performed.
Finish execution	`_Noreturn void osfiLoggerfinishExecution(int exitCode)`	Calls exit with the given value as exit code. The _Noreturn attribute is only declared if the C version reported by the compiler (__STDC_VERSION__) is at least C11 (value greater than or equal to 201112L).
Extension: show quality report	`void osfiLoggerQuality(char *name, double v)` `void osfiLoggerQualityDouble(char *name, char *v)`	The user should not introduce newlines in the strings, because it will break the output format.

Table 18: Functions of the ConFM module in C

General operation	C Prototype	Notes
Load config file	`bool osfiConFMReadConfigFile(const char *xmlFile)`	Returns false if errors prevent reading the file. If successful, parameters from any previous call to this functions are discarded and replaced from the values read from this file.
Validate against XSD	`enum osfi_confm_val_res osfiConFMValidateAgainst(const char* schemaFile)`	The result enum has three values for "validation passed", "validation ran but failed" and "validation could not run".
Get parameter by full path	-	All remaining functions in the ConFM module address a parameter by its full path.
Query parameter existence	`bool osfiConFMExistParameter(char* paramName)`	-
Query parameter element type	`ParamType osfiConFMGetElementType (const char *paramName)`	ParamType is an enum with the E2E-ICD simple types, that is, it does not represent types ARRAY or MATRIX.
Query parameter structure type	`bool osfiConFMIsArray(const char* paramName)`	-

Query parameter dimensions	`void osfiConFMGetDimension (char *paramName, int index, int *size)`
	`int osfiConFMGetRows (char *paramName)`
	`int osfiConFMGetColumns (char *paramName)`
		If index is past the dimensionality of the parameter, 0 is returned.
		Rows and columns are defined as dimensions #2 and #1, respectively, according to E2E-ICD.

Query parameter attributes	`void osfiConFMGetParameter (osfiParameter *param, char *paramName)`	Both param and all fields of *param must point to properly allocated buffers of the right sizes.

General operation	C Prototype	Notes

Get parameter parsed value (scalar)	`bool osfiConFMGetBoolValue (char *paramName)`
	`void osfiConFMGetIntegerValue (int *value, char *paramName)`
	`void osfiConFMGetDoubleValue (double *value, char *paramName)`
	`void osfiConFMGetStringValue (char *value, int *length, char *paramName)`
	`void osfiConFMGetFileValue (char *value, int *length, char *paramName)`
		For all versions taking a pointer, it must not be NULL. Furthermore, for strings, the length is written to *length on output but not checked in input: the code always assumes that buffers are preallocated to hold MAX_x+1 characters, where x is LENGTH_STRING or LENGTH_FILE_NAME.

Get parameter parsed value (vector)	`void osfiConFMGetVector#T#Values (#V# *vals, int *size, char *paramName)`	#T# is one of Integer, Double, Boolean, String or File, and #V# is the corresponding C type, namely int, double, bool, char* and char* respectively. If vals is NULL, only the required size is returned in *size. Otherwise, the array is **assumed to be large enough**. For string types, the same consideration as for the scalar getters operates: every slot of the buffer array must be preallocated to the mentioned size.
Get parameter parsed value (matrix)	`void osfiConFMGetMatrix#T#Values (#V# *vals, int *rows, int *cols, char *paramName)`	Same as for the vector getters. The buffer used is a 1-D array, not an array of arrays (except for strings). Matrix elements are written to it in row-major order.
Extension: query file existence	`void osfiConFMfileExist(bool *booleanMatrix, int *rows, int *columns, char *paramName)`	For FILE parameters. If the parameter is a scalar or vector, a 1x1 or single-row matrix is written, respectively.

General operation	C Prototype	Notes

ARRAY access – getArrayValue API	`void osfiConFMGetNodeDimension(char *paramName, int node[], int depth, int *size)`
	`bool osfiConFMIsLeaf(char *paramName, int node[], int depth)`
	`void osfiConFMGetLeafVector#T#Values (#V# *vals, int *size, char *paramName)`
		In all cases, "depth" is the number of elements in the input array "node". Together, both parameters represent the concept of a list if ints.
		For the getLeafVector#T# functions, same notes as for the non-ARRAY vector getters. Furthermore, they do **not** flatten sub-elements: they only return non-empty values for leaf nodes.

Language-specific: free OSFI-C resources	`int osfiCommonClose()`	Destroys the held instances of **both** the C++ CLP and ParamReader objects. Returns zero on failure, nonzero on success.

4.3.3. Fortran Programming Language

The OpenSF integration libraries for Fortran language are designed to comply with the [F2003] standard. They 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.

If should be noted that OSFI-Fortran libraries use standard Fortran 2003 features thoroughly. In particular, FINAL subroutines are used to provide automatic clean-up of OSFI objects, and both the implementation and the examples use automatic left-hand side reallocation on assignment. This feature means that it is not necessary to explicitly allocate an ALLOCATABLE variable or array that is being assigned: if it is not allocated or the size is incorrect, it is automatically deallocated (if needed) and reallocated to the new, correct size. Before Fortran 2003 this was not the case, and some compilers keep the old behavior even when compiling new code. Make sure to look at your compiler documentation to enable "realloc_lhs" or "Fortran 2003 standard compliant mode".

General description:

- Five modules are provided: OSFI, OSFI_base, OSFI_EHLog, OSFI_CLP and OSFI_ConFM. The first includes all others, while the second declares some shared elements and data types.
- Types in E2E-ICD are mapped as follows: INTEGER to (default) integer, FLOAT to double precision and BOOLEAN to (default) logical. The STRING, FILE and FOLDER types are all mapped to a character variable with LEN=* on input and LEN=: on output (as a deferred-length allocatable output argument or return). In future versions, it is possible that the arithmetic types will be defined to use a specific KIND (instead of default integer).
- 1-D "list" and 2-D "list of list" types are generally mapped to arrays and matrices of the corresponding Fortran type, except in the case of strings. Since an array or matrix of character variables would all share the same length (because the length is part of the type), a derived type OSFI_Str is provided whose only member is a deferred-length allocatable character variable. Thus, a 1D list of strings is a Fortran array of OSFI_Str instances.
- Access to ARRAY-typed parameters is provided using the "getArrayValue" approach described in §4.2.3.4. It is implemented in the form of getVectorT functions that take the desired slice indices.
- Error conditions are reported in one of two ways: some functions return an object instance (e.g. of type CLP, Parameter) and this object instance has an isValid method that returns a logical value. In other cases, the Fortran functions of which the C++ equivalent can raise them have an optional output parameter called "stat". If this parameter is given, it will have a value of zero on successful execution, and nonzero on error.

Known issues:

- 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

The CLP module API is provided by module OSFI_CLP. The implementation takes the form of a single class "CLP" which parses command line arguments as provided by the Fortran runtime. Unlike in other OSFI implementations, this one does not allow user code to replace the arguments to be parsed. No global state is stored and thus multiple instances can coexist. Furthermore, all parsing is done by the "constructor", and accessor functions only return copies of stored data.

Table 13 details the interface of the CLP module in OSFI-Fortran.

4.3.3.2. EHLog

The EHLog module API is provided by module OSFI_EHLog, which contains free subroutines (not type-bound procedures). Relevant status variables (debug/color) are initialized at first execution of an output, and stored as global data from them on. Furthermore, functions operate on the global output and error streams (from C++). Thus, thread safety is not guaranteed in these functions.

Note that no custom formatting routines are available: in order to write a formatted string to the OSFI log in Fortran, use code must first render it into a string by using Fortran internal-file write statements.

Table 14 details the interface of the EHLog module in OSFI-Fortran.

4.3.3.3. ConFM

The ConFM module API is provided by module OSFI_ConFM. It defines the two derived types OSFI_ParamReader and OSFI_Parameter.

The ParamReader class is the main access point to the module interface. Each instance is independent and holds no global state, so several instances can be kept (e.g. for the global and local configuration files). The instance holds ownership of and provides access to a set of Parameter instances, which are likewise independent of any other instance. However, thread safety is not guaranteed in any of the functions because they may call the Logger functions to report errors.

Table 15 details the interface of the ConFM module in OSFI-Fortran.

Table 19: Functions of the CLP module in Fortran

General operation	Fortran procedure "interface"	Notes
Parse command line arguments ("constructor")	`Function OSFI_CommandLineParser() Result(newClp)` `  Type(OSFI_CommandLineParser) newClp`	On error, the returned object has newClp%isValid set to false and calling any getter functions returns empty arrays.

Get (full) list of C/I/O files	`Subroutine getConfFiles(this, files[, stat])`
	`  Class(OSFI_CommandLineParser),Intent(In) ::this`
	`  Type(OSFI_Str), Allocatable :: files`
	`  Integer, Optional :: stat`
	`  Intent(Out) :: files, stat`
	Same signatures: **getInputFiles** and **getOutputFiles**.
		If no file of a type is provided, the subroutines still allocate "files" (to an empty array) and set "stat" to zero if present.
		On error, "files" is not allocated and "stat" (if present) is set to nonzero.

Table 20: Functions of the EHLog module in Fortran

General operation	Fortran procedure "interface"	Notes

Show I/W/E/D message	`Subroutine osfi_info(message)`
	`  Character(len=*), Intent(In) :: message`
	Same signatures: **warning**, **error** and **debug**.
		The user should not introduce newlines in the string, because it will break the output format.

Show progress indication	`Subroutine osfi_progress(n, m)` `  Integer, Intent(In) :: n, m`	No validation is performed.
Finish execution	`Subroutine osfi_finishExecution(errorCode)` `  Integer, Intent(In) :: errorCode`	Calls the C++ equivalent to this function – beware if Fortran runtime-specific termination actions are required.

General operation	Fortran procedure "interface"	Notes

Extension: show quality report	`Subroutine osfi_qualityReport(name, value)`
	`  Character(*), Intent(In) :: name`
	`  ! Either type is accepted (generic interface)`
	`  Character(*), Intent(In) :: value`
	`  Double precision, Intent(In) :: value`
		The user should not introduce newlines in the strings, because it will break the output format.

Query debug mode	`Logical Function osfi_logger_isDebugging()`	Status set from environment variables DEBUG_MODE on first query or write to OSFI output.

Table 21: Functions of the ConFM module in Fortran

General operation	Fortran procedure "interface"	Notes
Test if instance is valid	`Logical Function isValid(this)` `  Class(OSFI_Handle), Intent(In) :: this`	Both OSFI_ParamReader and OSFI_Parameter extend OSFI_Handle. Returns true if the handle is valid, that is, if the constructor completed without errors.

Load config file	`Function OSFI_ParamReader(fileName[, stat])`
	`  Result(this)`
	`  Character(*), Intent(In) :: fileName`
	`  Integer, Intent(Out), Optional :: stat`
	`  Type(OSFI_ParamReader) :: this`
		On success, returns a valid instance and, if stat is present, sets it to zero.
		On error, returns an invalid instance and, if stat is present, sets it to a nonzero value.

Validate against XSD	`Logical Function validateAgainst(pr, xsdFile`
	`  [, stat])`
	`  Class(OSFI_ParamReader), Intent(In) :: pr`
	`  Character(*), Intent(In) :: xsdFile`
	`  Integer, Intent(Out), Optional :: stat`
		Combinations of: (return value, stat) are (.true., 0) for a passed validation, (.false., 0) for a validation that ran but did not pass; and (.false., nonzero) if the validation could not run, e.g. because the XSD could not be parsed.

General operation	Fortran procedure "interface"	Notes

Extension: validate against internal schema	`Logical Function validateAgainstInternalSchema (pr[, stat])`
	`Class(OSFI_ParamReader), Intent(In) :: pr`
	`Integer, Intent(Out), Optional :: stat`
		Result as above.

Get parameter by full path	`Function getParamRef(pr, paramName[, stat])`
	`Class(OSFI_ParamReader), Intent(In) :: pr`
	`Character(*), Intent(In) :: paramName`
	`Integer, Intent(Out), Optional :: stat`
	`Type(OSFI_Parameter) :: getParamRef`
		On success, returns a valid instance and, if stat is present, sets it to zero.
		On error, returns an invalid instance and, if stat is present, sets it to a nonzero value.

Query existence	`Logical Function existParameter(pr, paramName)`	Same argument types as getParamRef.

Get parameter parsed value (scalar, vector, matrix)	`Function get#T#Value(p[, stat]) Result(out)`
	`#V# :: out`
	`Function getVector#T#(p[, stat]) Result(out)`
	`#V#, Allocatable :: out(:)`
	`Function getMatrix#T#(p[, stat]) Result(out)`
	`#V#, Allocatable :: out(:,:)`
	`! Common parameter "p" is the "this" argument:`
	`Class(OSFI_Parameter), Intent(In) :: p`
	`Integer, Intent(Out), Optional :: stat`
		#T# is one of Int, Double, Boolean, String or File, like in C++. #V# is the corresponding type according to the type mapping (integer, double precision, logical, character).
		Note that #V# for string types is:
		❑ For scalar getter: Character(len=:), adding the Allocatable attribute
		❑ For vector/matrix getters: Type(OSFI_Str)
		On a parsing error, a log message is emitted and a default value is returned (see docs). If stat is present, it is set to nonzero on error, or zero on success.

General operation	Fortran procedure "interface"	Notes

ARRAY access – getArrayValue API	`Integer Function getDims(p, node)`
	`Logical Function isLeaf(p, node)`
	`Function getVector#T#(p, node[, stat]) Result(out)`
	`  Integer, Intent(Out), Optional :: stat`
	`  #V#, Allocatable :: out(:)`
	`! Common parameters: ("p" is the "this" argument)`
	`  Class(OSFI_Parameter), Intent(In) :: p`
	`  Integer, Intent(In) :: node(:)`
		The vector represents the index of the desired slice. Slices of a parameter with dimensionality d must have at most d−1 indices, because a vector is returned for the last dimension.
		The getVector#T# functions do **not** flatten sub-elements: they only return non-empty values for leaf nodes, that is, those for which isLeaf returns true.
		#V# is the same as for the non-ARRAY vector getter functions, and the "stat" parameter of the getter function works the same as described there.

Extension: query file existence	`Function getFileExists(p, stat) Result(out)`
	`  Class(OSFI_Parameter), Intent(In) :: p`
	`  Integer, Intent(Out), Optional :: stat`
	`  Logical, Allocatable :: out(:,:)`
		For FILE parameters. If the parameter is a scalar or vector, a 1x1 or single-row matrix is returned, respectively.

Query parameter attributes	`Function getDims(p) Result(dims)`
	`  Integer, Allocatable :: dims(:)`
	`Function getElementType(p) Result(elType)`
	`  Type(OSFI_ParamElemType) :: elType`
	`Function getName(p) Result(name)`
	`  Character(len=:), Allocatable :: name`
	`Logical Function isArray(p)`
	`! The common parameters is the "this" argument`
	`  Class(OSFI_Parameter), Intent(In) :: p`
		OSFI_ParamElemType is a derived type with an integer field "repr" which approximates an enumeration. It represents the E2E-ICD simple types, that is, it does not contain values for types ARRAY or MATRIX.

4.3.4. Fortran 77 Programming Language

OSFI-F77 is deprecated. It is no longer in active development, and this section is no longer updated. Since almost no F77-only compilers remain in the market, it is recommended to use the OpenSF integration libraries for Fortran instead (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

OSFI-IDL is deprecated. It is no longer in active development, and this section is no longer updated.

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

OSFI-Matlab code is tested to work in Matlab 2013b. The implementation is pure Matlab, not depending any MEX code, although XML parsing does depend on Java and thus will not work if Matlab is started without a JVM. Its design is object-oriented, and in general error conditions are communicated through Matlab errors, although some methods just log errors to the OSFI log and return token values instead.

General description:

- Types in E2E-ICD are mapped to their reasonable equivalents: INTEGER to int32, FLOAT to double, BOOLEAN to logical and the STRING, FILE and FOLDER types to char array strings.
- 1-D "list" types are generally mapped to vector-shaped matrices or cell arrays containing the appropriate types. Matrices (described in the general API as "lists of lists") are represented by actual matrices or matrix-shaped cell arrays.
- Access to ARRAY-typed parameters is provided with the "getArrayValue" approach described in §4.2.3.4.
- In order to use OSFI in Matlab the library must be accessible through the Matlab path. Either it must be deployed in a folder in the path or it must be available in a folder known to the script so it can be added to it. Look at §3.4.3.3 for more information on this.

Known issues: none currently

4.3.6.1. CLP

The CLP module API is provided by the single class "CLP". The class parses command line arguments as passed. No global state is stored and thus multiple instances can coexist. Furthermore, all parsing is done by the constructor, and accessor functions only return copies of the stored data.

Table 28 details the interface of the CLP module in OSFI-Matlab

4.3.6.2. EHLog

The EHLog module API is provided by the single class "Logger". Unlike other OSFI implementations, functions are instance method, not static, and thus several instances of the logger can coexist with different settings. Output does not support coloring, and other relevant status variables (debug mode) are initialized on construction.

Table 29 details the interface of the EHLog module in OSFI-Matlab.

4.3.6.3. ConFM

The ConFM module API is provided by the ConFM and Parameter classes. ConFM is the main access point to the interface. Each instance is independent and holds no global state, so several instances can be kept (e.g. for the global and local configuration files).

The ConFM class provides access to a set of Parameter instances, which are likewise independent of any other instance. The main API they offer is the getValue function which returns the parsed value of the parameter. For ARRAY parameters, this is in the form of nested cell arrays reproducing the structure in the XML.

An additional way to access ARRAY parameters is provided through the getArrayValue function, which can be passed a set of indices and returns the corresponding slice of the structure that getValue would have returned. Calling getArrayValue with a series of indices is equivalent to calling getValue and then applying those indices to the result; the main difference is that with getArrayValue, the indexing is applied *before* parsing.

Table 30 details the interface of the ConFM module in OSFI-Matlab.

Table 22: Functions of the CLP module in Matlab

General operation	Matlab function	Notes
Parse command line arguments ("constructor")	`obj = CLP.CLP(confFiles, inputFiles, outputFiles)`	The function initializes the object by parsing the three arguments. The output files argument is optional.
Get (full) list of C/I/O files	`files = CLP.getConfFiles(clp)` Same signatures: **getInputFiles** and **getOutputFiles**.	The return value is a cellstr. If no file of a type is provided, an empty cell is returned.
Legacy: get single C/I/O file	`file = CLP.getConfFile(clp, index)` Same signatures: **getInputFile** and **getOutputFile**.	Returns files{index} where files is the corresponding full C/I/O files list.
Legacy: number of C/I/O files	`num = CLP.nConfFiles(clp)` Same signatures: **nInputFiles** and **nOutputFiles**.	Returns length(files) where files is the corresponding full C/I/O files list.

Table 23: Functions of the EHLog module in Matlab

General operation	Type-annotated Python Function	Notes
Initialize EHLog	`obj = Logger.Logger()`	Parses the DEBUG_MODE environment variable.
Show I/W/E/D message	`Logger.info(obj, message)` Same signatures: **warning**, **error** and **debug**.	The user should not introduce newlines in the string, because it will break the output format.
Show progress indication	`Logger.progress(obj, step, nSteps)`	No validation is performed.
Finish execution	`Logger.finishExecution(obj, [exitCode])`	If exitCode is provided, calls exit(exitCode). Otherwise, throws an error to exit.
Extension: show quality report	`Logger.qualityReport(obj, name, value)`	The user should not introduce newlines in the strings, because it will break the output format.
Extension: set debug mode	`Logger.setDebugMode(obj, debug)`	-
Extension: set output to file	`Logger.setStandAlonMode(obj, standAlone)`	If true, further outputs of the Logger object will be written to a file in the current working directory named ".tmpLogFile".

Table 24: Functions of the ConFM module in Matlab

General operation	Type-annotated Python Function	Notes
Load config file	`obj = ConFM.ConFM([fileName])` `ConFM.parseFile(obj, fileName)`	An error is raised if the file cannot be found, or if it cannot be parsed as a configuration file. Using parseFile replaces the stored parameter map in the ConFM object.
Validate against XSD	`valOk = ConFM.validateAgainst(obj, xsdFile)`	Throws an error if the schema file cannot be loaded or parsed.
Extension: validate against internal schema	`valOk = ConFM.validateAgainstInternalSchema(obj)`	Throws an error if the linked schema cannot be loaded.
Get parameter by full path	`P = ConFM.getParameter(obj, name)`	Returns a Parameter instance. Throws an error if the path is not found.
Get all parameters	`map = ConFM.getParameters(obj)`	Returns a containers.Map instance with string keys and Parameter values.
Get parameter raw value	`Parameter.getRawValue(self) -> RawType`	For non-ARRAY parameters, the return value is the unparsed string value (for matrices, rows are joined). For ARRAY parameters, a structure of cell arrays reproducing the XML structure is returned. Each cell may contain other cells or a 2-element cell {nElems, stringVal}.
Get parameter parsed value with automatic type	`val = Parameter.getValue(p)`	For non-ARRAY parameters, the return value is a matrix or cell array of the type corresponding to the declared type. For ARRAYs, a nested structure of cell arrays is returned, where the *last* level contains a vector of data elements.

General operation	Type-annotated Python Function	Notes
ARRAY access	`val = Parameter.getArrayValue(p, varargin)`	Each element in varagin is interpreted as a subindex into the structure of cell arrays returned by getValue. The last element may also index into the data itself.
Query parameter dimensionality	`ndims = Parameter.getNdims(p)` `dims = Parameter.getDims(p)`	The second returns a vector of sizes, while the first returns the length of that vector.
Query parameter element type	`pt = Parameter.getType(p)`	Returns a value from ParamType, an enumerated class that represents the E2E-ICD simple types, that is, it does *not* contain values for types ARRAY or MATRIX.
Query parameter attributes	`val = Parameter.getName(self) -> str` Same signature: **getPath**, **getDescription**, **getUnits**, **getMax**, **getMin**	All return strings except for getMin and getMax, which try to parse the related attribute with str2double.

4.3.7. Python Programming Language

OSFI-Python code is written to work in both Python 2.7 [Py2.7] and 3.5 [Py3.5]. The library is presented as a folder (currently not a package) containing several classes.

The implementation is pure Python, not depending on native OSFI libraries. Its design is object-oriented, with both the CLP and ConFM modules implemented using classes: CLP, ParamReader and Parameter are the main ones. On the other hand, the Logger module defines a set of functions and stores its state "globally" in the module itself. In general, errors are communicated through exceptions, although some methods just log errors to the OSFI log stream and return token values instead.

General description:

- Types in E2E-ICD are mapped to their reasonable equivalents: INTEGER to int, FLOAT to float, BOOLEAN to bool and the STRING, FILE and FOLDER types to str (in both Python 2 and 3).
- 1-D "list" types are generally mapped to either lists or tuples. Matrices (described in the general API as "lists of lists") are indeed represented by lists of lists.
- Access to ARRAY-typed parameters is provided with the "getArrayValue" approach described in §4.2.3.4.
- In order to use OSFI in Python the library must be accessible through the import path. Either it must be deployed in a folder in PYTHONPATH or it must be available in a folder known to the script, so that it can add the folder to sys.path as needed. Look at §3.4.3.3 for more information on this.

Known issues:

- The XSD validation functions require the presence of the lxml library. If this is not available, the xml.etree library will be used for parsing, but calling any XSD validation-related function will raise NotImplementedError.

Note that, in the detailed API tables, the functions are described with type annotations and (where applicable) keyword-only arguments as supported in Python 3.5, as if the typing and typing.io packages had been imported. However, the actual functions are Python 2.7-compatible, and thus do *not* carry type annotations. Similarly, keyword-only arguments are implemented as **kwargs where any unknown arguments trigger an error.

4.3.7.1. CLP

The CLP module API is provided by the single class "CLP" in module CLP. The class parses command line arguments as passed. No global state is stored and thus multiple instances can coexist. Furthermore, all parsing is done by the constructor, and accessor functions only return copies of the stored data.

Table 28 details the interface of the CLP module in OSFI-Python.

4.3.7.2. EHLog

The EHLog module API is provided by functions in module Logger. Relevant status variables (debug/color) are initialized at first execution of an output, and stored as global

data from them on. Furthermore, functions operate on the global streams sys.stdout and sys.stderr. Thus, thread safety is not guaranteed in these functions.

Table 29 details the interface of the EHLog module in OSFI-Python.

4.3.7.3. ConFM

The ConFM module API is provided by the ParamReader and Parameter classes. ParamReader is the main access point to the interface. Each instance is independent and holds no global state, so several instances can be kept (e.g. for the global and local configuration files).

The ParamReader class provides access to a set of Parameter instances, which are likewise independent of any other instance. However, thread safety is not guaranteed in any of the functions because they may call the Logger functions to report errors.

There is a generic function getValue which returns the parsed value of the parameter, considering the dimensionality declared in the XML. Thus, a 1x1 integer matrix with value "1" will return the list-of-list-of-int result `[[1]]`. If this is undesired, specific functions are available to override the dimensionality and the type. For ARRAY-typed parameters, getValue returns a structure of nested lists with the parsed values of each node from the XML.

Furthermore, sliced access to ARRAY parameters is provided through the getArrayValue function, which can be passed a set of indices and returns the corresponding slice of the structure that getValue would have returned. Calling getArrayValue with a series of indices is equivalent to calling getValue and then applying those indices to the result; the main difference is that with getArrayValue, the indexing is applied *before* parsing.

Table 30 details the interface of the ConFM module in OSFI-Python.

Table 25: Functions of the CLP module in Python

General operation	Type-annotated Python Function	Notes
Parse command line arguments ("constructor")	`CLP.__init__(self: CLP, argv: List[str] = None)`	If None is given in argv, the function will use the value of sys.argv instead.
Get (full) list of C/I/O files	`CLP.getConfFiles(self: CLP) -> List[str]` Same signatures: **getInputFiles** and **getOutputFiles**.	If no file of a type is provided, they return an empty list.
Legacy: get single configuration file	`CLP.getConfFile(self: CLP) -> str`	If called in a case with two configuration files, returns the unparsed string ("file1,file2").

Table 26: Functions of the EHLog module in Python

General operation	Type-annotated Python Function	Notes
Show I/W/E/D message	`info(msg: str) -> None` Same signatures: **warning**, **error** and **debug**.	The user should not introduce newlines in the string, because it will break the output format.
Extension: format I/W/E/D message with extra data	`getInfoStream() -> TextIO` Same signatures: **getWarningStream**, **getErrorStream**, **getDebugStream**.	The user should not introduce newlines in the output given to those streams, because it will break the output format.
Show progress indication	`progress(step: int, nSteps: int) -> None`	No validation is performed.
Finish execution	`finishExecution(exitCode: int) -> None`	Calls sys.exit with the given value as exit code.
Extension: show quality report	`qualityReport(name: str, value: Any)`	The user should not introduce newlines in the strings, because it will break the output format.
Extension: format quality report	`getQualityStream() -> TextIO`	The user should not introduce newlines in the output given to those streams, because it will break the output format.
Query debug and colored output.	`isColored() -> bool` Same signature: **isDebugging**	Status set from environment variables DEBUG_MODE and OSFI_LOG_COLOR on first query.

Table 27: Functions of the ConFM module in Python

General operation	Type-annotated Python Function	Notes
Load config file	`ParamReader.__init__(self, xmlFile: str)`	An exception is raised if the file cannot be found, or if it cannot be parsed as a configuration file.
Validate against XSD	`ParamReader.validateAgainst(self, xsdFile: str) -> bool`	Raises an exception if the schema file cannot be loaded or parsed as valid XSD.
Extension: validate against internal schema	`ParamReader.validateAgainstInternalSchema(self)`	Returns failure if the schema cannot be found or parsed.
Get parameter by full path	`ParamReader.getParameter(self, name: str) -> Parameter`	Returns null if the name is not found.
Get parameters by partial path	`ParamReader.getParameters(self, groupName: str) -> List[Parameter]`	Returns an empty list if no parameter matches.
Get all parameters	`ParamReader.getAllParameters(self) -> Dict[str, Parameter]`	The map key is the full path, which is not accessible from within each Parameter item.
Query existence	`ParamReader.existParameter(self, name:str) -> bool`	-

Get parameter parsed value with automatic dimensionality and possibly type	`Parameter.getValue(self, asType: ParamType = None) -> ValType`
	
	`ValType`
	`= V # for scalars`
	`= List[V] # for vectors`
	`= List[List[V]] # for matrices`
	`= List[X] # for ARRAY, where X is V or List[X]`
		V is the parsed type corresponding to for asType, or, it if is None, to the declared element type.
		
		If the parsed element type is called T, the return type depends on the declared dimensionality, as described on the left. For ARRAYs, a nested structure of lists is returned, where the *last* level is of type List[T]

General operation	Type-annotated Python Function	Notes

Get parameter parsed value (scalar, vector, matrix)	`Parameter.get#T#Value(self) -> V`
	`Parameter.getVector#T#(self) -> List[V]`
	`Parameter.getMatrix#T#(self) -> List[List[V]]`
		#T# is one of Int, Double, Boolean, String or File, and V is the corresponding type (int, float, bool, str, str).
		On parsing error, a log message is emitted and a default value is returned (see docs). If the getVector#T# functions are used on an ARRAY parameter, it is flattened to 1D.

ARRAY access	`Parameter.getArrayValue(self, *indices, asType: ParamType = None) -> ValType`	ValType is as defined for getValue. Each element in indices is interpreted as a subindex into the structure returned by getValue. The last element may also index into the data itself.
Get parameter raw value	`Parameter.getRawValue(self) -> RawType`	For non-ARRAY parameters, the return value is the unparsed string value (for matrices, rows are joined). For ARRAY parameters, a structure of nodes reproducing the XML is returned.
Extension: query file existence	`Parameter.fileExists(self) -> List[List[bool]]`	For FILE parameters. If the parameter is a scalar or vector, a 1x1 or single-row matrix is returned, respectively.

Query parameter attributes	`Parameter.getNdims(self) -> int`
	`Parameter.getDims(self) -> List[int]`
	`Parameter.getElementType(self) -> ParamType`
	`Parameter.getName(self) -> str`
	Same signature as getName: **getDescription**, **getUnits**, **getMax**, **getMin**, **getType**
		ParamType is an enum with the E2E-ICD simple types, that is, it does not represent types ARRAY or MATRIX.

4.3.8. Java Programming Language

OSFI-Java code targets Java SE 7, with the language specification defined in [Java7]. The library is presented as a single JAR file which does not have any external dependencies. The JAR is also an OSGi bundle, which exports the package "esa.opensf.osfi".

The implementation of OSFI-Java is object-oriented, with both the CLP and ConFM modules implemented using classes: CLP, ParamReader and Parameter are the main ones. The Logger module is mainly static method-based and stores its state globally. In general, errors are communicated through exceptions, although some methods just log errors to the OSFI log stream and return token values instead.

General description:

- All classes are directly in the "esa.opensf.osfi" package.
- Types in E2E-ICD are mapped to their reasonable equivalents: INTEGER to int, FLOAT to double, BOOLEAN to boolean and the STRING, FILE and FOLDER types to java.lang.String.
- 1-D "list" types are generally mapped to either java.util.List<T> or arrays. Matrices (described in the general API as "lists of lists") are represented by arrays of arrays.
- Access to ARRAY-typed parameters is provided implementing the "getRootNode" approach described in §4.2.3.4.

Known issues: none for the moment

4.3.8.1. CLP

The CLP module API is provided by the single class "CLP". The class parses command line arguments as passed. No global state is stored and thus multiple instances can coexist. Furthermore, all parsing is done by the constructor, and accessor functions only return copies of the stored data.

Table 28 details the interface of the CLP module in OSFI-Java.

4.3.8.2. EHLog

The EHLog module API is provided by the "fake" class "Logger". The implementation provides the functions as static methods of that class.

Relevant status variables (debug/color) are initialized at first execution of an output, and stored as global data from them on. Furthermore, functions operate on the global streams System.out and System.err. Thus, thread safety is not guaranteed in these functions.

Table 29 details the interface of the EHLog module in OSFI-Java.

4.3.8.3. ConFM

The ConFM module API is provided by the ParamReader, Parameter and ArrayNode classes. ParamReader is the main access point to the module interface. Each instance is independent and holds no global state, so several instances can be kept (e.g. for the global and local configuration files).

The ParamReader class provides access to a set of Parameter instances, which are likewise independent of any other instance. However, thread safety is not guaranteed in any of the functions because they may call the Logger functions to report errors.

Parsing code employs primitive arrays as much as possible, avoiding boxing large quantities of data. Thus, the getVector and getMatrix functions return primitive arrays for types INTEGER, FLOAT and BOOLEAN.

Access to ARRAY parameters is provided through the ArrayNode class. It is an abstract class, defined as ArrayNode<Es,S>. Es is the data type contained, and S is the actual type of the node, since it will be a subclass of ArrayNode[4]. In particular, nodes are always instances of either of two concrete subclasses:

- `ArrayNode.Raw`, which contains String data. This represents the structure of the parameter read in the XML and has methods to parse the contents into either an array (flattening the structure to one dimension in depth-first order) or a parsed node which keeps the structure but contains parsed data.
- `ArrayNode.Parsed<A>` which contains data of type A, where A will be an array type. This is done because Java generics cannot be primitives, so A could not be e.g. "int", but it can be int[] because array types are objects.

The parent class ArrayNode, and thus both subtypes of nodes, contain methods to navigate the tree structure: getDataAt(indices) and getSubNodeAt(indices) are the main features, which can be explored in the documentation.

Table 30 details the interface of the ConFM module in OSFI-Java.

[4] S is sometimes called a CRTP type parameter, using terminology borrowed from C++ and its Curiously Recursive Template Pattern.

Table 28: Functions of the CLP module in Java

General operation	Java Method	Notes
Parse command line arguments ("constructor")	`CLP.CLP(String[] args)`	The input is the array provided to the entry point.
Get (full) list of C/I/O files	`List<String> CLP.getConfFiles()` Same signatures: **getInputFiles** and **getOutputFiles**.	If no file of a type is provided, they return an empty list.
Legacy: get single configuration file	`String CLP.getConfFile()`	If called in a case with two configuration files, returns the unparsed string ("file1,file2").
Extension: parse string to files	`List<String> CLP.parseFiles(String arg) throws Exception`	-
Extension: format files to string	`static String CLP.arrayToString (List<String> values)`	-

Table 29: Functions of the EHLog module in Java

General operation	Java Method	Notes
Show I/W/E/D message	`static void Logger.info(String msg)` Same signatures: **warning**, **error** and **debug**.	The user should not introduce newlines in the string, because it will break the output format.
Extension: format I/W/E/D message with extra data	`static OutputStream Logger.getInfoStream()` Same signatures: **getWarningStream**, **getErrorStream**, **getDebugStream**.	The user should not introduce newlines in the output given to those streams, because it will break the output format.
Show progress indication	`static void Logger.progress(int, int)`	No validation is performed.
Finish execution	`static void Logger.finishExecution(int exitCode)`	Calls System.exit with the given value as exit code.
Extension: show quality report	`static void Logger.qualityReport(String name, double value)` `static void Logger.qualityReport(String name, String value)`	The user should not introduce newlines in the strings, because it will break the output format.
Extension: format quality report	`static OutputStream Logger.getQualityStream()`	The user should not introduce newlines in the output given to those streams, because it will break the output format.
Query debug and colored output.	`static boolean Logger.isColored()` Same signature: **isDebugging**	Status set from environment variables DEBUG_MODE and OSFI_LOG_COLOR on first query.

Table 30: Functions of the ConFM module in Java

General operation	Java Method	Notes
Load config file	`ParamReader.ParamReader (String xmlFile) throws FileNotFoundException, XMLParser.ParseException`	The first exception is thrown if the file cannot be found, the second if it cannot be parsed as a configuration file.
Validate against XSD	`boolean ParamReader.validateAgainst (String xsdFile)`	Throws IllegalArgumentException if the schema file cannot be loaded or parsed.
Extension: validate against internal schema	`boolean ParamReader.validateAgainstInternalSchema()`	Returns failure if the schema cannot be found or parsed.
Get parameter by full path	`Parameter ParamReader.getParameter(String name)`	Returns null if the name is not found.
Get parameters by partial path	`List<Parameter> ParamReader.getParameters (String groupName)`	Returns an empty list if no parameter matches.
Get all parameters	`Map<String, Parameter> ParamReader.getAllParameters()`	The map key is the full path, which is not accessible from within each Parameter item.
Query existence	`boolean ParamReader.existParameter(String name)`	-

Get parameter parsed value (scalar, vector, matrix)	`#V# Parameter.get#T#Value()`
	`#V#[] Parameter.getVector#T#()`
	`#V#[][] Parameter.getMatrix#T#()`
		#T# is one of Int, Double, Boolean, String or File, and #V# is the corresponding type (int, double, boolean, String, String).
		On parsing error, a log message is emitted and a default value is returned (see docs). If the getVector#T# functions are used on an ARRAY parameter, it is flattened to 1D.

General operation	Java Method	Notes
Extension: query file existence	`boolean[][] Parameter.fileExists()`	For FILE parameters. If the parameter is a scalar or vector, a 1x1 or single-row matrix is returned, respectively.

Query parameter attributes	`int Parameter.getNdims()`
	`List<Integer> Parameter.getDims()`
	`Parameter.ParamType Parameter.getElementType()`
	`String Parameter.getName()`
	Same sig.: **getDescription**, **getUnits**, **getMax**, **getMin**, **getType**
		Parameter.ParamType is an enum with the E2E-ICD simple types, that is, it does not represent types ARRAY or MATRIX.

ARRAY access – ArrayNode API	`ArrayNode.Raw Parameter.getRootNode()`
	`ArrayNode.Parsed<#V#[]> ArrayNode.Raw.getTree#T#()`
	`#V#[] ArrayNode.Raw.getVector#T#()`
	`<Es> Es ArrayNode<Es,?>.getData()`
	`<Es> Es ArrayNode<Es,?>.getDataAtSub(int... idxs)`
	`<S> S ArrayNode<?,S>.getSubNodes()`
	`<S> S ArrayNode<?,S>.getSubNodeAt(int... idxs)`
	`int ArrayNode.getDim()`
	`List<Integer> ArrayNode.getDimsEnvelope()`
	`boolean ArrayNode.isDataNode()`
		The parameter returns an ArrayNode.Raw element, which contains String data (without splitting).
		Raw ArrayNodes can be parsed either by flattening to 1D (getVector#T#) or by keeping its structure (getTree#T#) obtaining an ArrayNode.Parsed<#V#[]> element, which contain arrays of type #V# (possibly primitive).
		The other methods can be applied to either subclass: Es will be String in Raw nodes and an array in Parsed nodes. S will be the same type of node that is receiving the call (Raw/Parsed).
		The return value of getDimsEnvelope is the rectangular envelope of the dimensions of the structure at that node.

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 <iomanip>
#include <string>

using namespace std;

int main(int argc, char * argv[])
try {
  CLP clp{argc, argv}; // Parse command line arguments

  cout << "input files = ";
  for (auto& if : clp.getInputFiles())
    cout << if << ", ";
  cout << endl;

  const string lcf = clp.getConfFiles().back();
  ParamReader reader (lcf); // Parse LCF
  Logger::info("Printing whole parameters file");
  reader.print();

  DynamicArray<int> mi = reader.getParameter("mat").getMatrixInt();
  for (int i = 0, n = mi.getRows(); i < n; i++) {
    for (int j = 0, m = mi.getColumns(); j < m; j++) {
      cout << setw(4) << mi[i][j] << '\t';
    }
    cout << endl;
  }

  Logger::info("vec");
  for(double d : reader.getParameter("vec").getVectorDouble())
    cout << fixed << setw(4) << setprecision(1) << d << '\t';
  cout << endl;
  return 0;
} catch (const std::exception& e) {
  Logger::getErrorStream() << "Module failed: " << e.what() << endl;
  Logger::finishExecution(1);
}
```

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 <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  osfiCLP(argc, argv); // Initialize CLP component
  char cfgFiles[2][MAX_LENGTH_FILE_NAME+1];
  int numCfgFiles;
  osfiCLPGetConfFiles(cfgFiles, &numCfgFiles);

  if (numCfgFiles != 2 || !osfiConFMReadConfigFile (cfgFiles[1])) {
    osfiLoggerError("Bad number of cfg files (%d) "
        "or error reading file %s", numCfgFiles, cfgFiles[1]);
    osfiLoggerFinishExecution(1);
  }

  double fltParVal; // Scalar parameter
  osfiConFMGetDoubleValue(&fltParVal, "los.LOS.initialTime.second");
  printf("Scalar float parameter = %g\n", floatParam, fltParVal);

  char *matPar = "matrix5x4"; // Matrix Parameters
  int m = osfiConFMGetRows(matPar), n = osfiConFMGetColumns(matPar);
  int *intMat = malloc(m * n * sizeof(int));
  int rows, cols;
  osfiConFMGetMatrixIntegerValues(intMat, &rows, &cols, matPar);
  for (i = 0; i < rows; i++) {
    for (j = 0; j < cols; j++)
      printf("%s[%d][%d] = %d\n", matPar, i, j, intMat[i*cols+j]);
  }
  free(intMat);

  osfiParameter param; // Parameter Attributes Retrieval
  param.name = malloc(MAX_PARAMETER_NAME_SIZE + 1);
  param.description = malloc(MAX_PARAMETER_DESC_SIZE + 1);
  param.value = malloc(MAX_PARAMETER_FIELD_SIZE + 1);
  param.units = malloc(MAX_PARAMETER_FIELD_SIZE + 1);
  param.min = malloc(MAX_PARAMETER_FIELD_SIZE + 1);
  param.max = malloc(MAX_PARAMETER_FIELD_SIZE + 1);
  osfiConFMGetParameter(param, matPar);
  printf("%s, raw value=%s\n", param.name, param.value);
  free(param.name);
  free(param.description);
  free(param.value);
  free(param.units);
  free(param.min);
  free(param.max);
  return 0;
}
```

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, fully or partially
  Use OSFI_ConFM
  Use OSFI_CLP
  Use OSFI, Only: osfi_error, osfi_info, osfi_finishExecution

  Type(OSFI_CommandLineParser) clp
  Type(OSFI_STR), Allocatable, Dimension(:) :: &
       cfgFiles, inputFiles, outputFiles
  Character(*), Parameter :: matPar = "sensor.NumericModel.polyParX"
  Double precision, Allocatable :: doubleMatrix(:,:)
  Character(1) tmp
  Integer :: i, err

  clp = OSFI_CommandLineParser()
  If (clp%isValid()) Call clp%getConfFiles(cfgFiles)
  If (.not. allocated(cfgFiles)) Then
    Call osfi_error('Command line arguments were not parsed')
    Call osfi_finishExecution(1)
  End If
  Do i=1, size(cfgFiles)
    Write (tmp, '(I1)') i
    Call osfi_info("Cfg File ("//tmp//") = "//cfgFiles(index)%str)
  End Do

  ! ConFM Module Example
  pr = OSFI_ParamReader(cfgFiles(1)%str)
  If (.Not. pr%isValid()) then ! Triggered if the file is not found
or cannot be parsed
    Call osfi_error('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(matPar, stat=err)
  If (err == 0) doubleMatrix = p%getMatrixDouble(stat=err)
  If (err /= 0) Then
    Call osfi_error('Could not find or parse ' // matPar)
    Call osfi_finishExecution(3)
  End If
  Do i=1,size(doubleMatrix,1) ! Write row by row
    Write(*,*) doubleMatrix(i,:)
  End Do
End Program
```

4.5.4. Fortran 77 Programming Language

Here is an example of Fortran 77 code that uses the different modules of the integration libraries. Note that the F77 interface is deprecated and no longer developed.

```
      program test
      implicit none
      INTEGER nconf, stat, i, j, nrows, ncols, p
      CHARACTER*255 fname, tmp
      LOGICAL*1 pexist, bmatrix(255)

c---- TEST OSFI COMMAND LINE PARSER
      call OCLP()
c     Get number of configuration files and print their names
      call OCLPNC (nconf)
      WRITE(tmp, '(I2)') nconf
      call OLINFO ('Number of configuration files:  '//tmp)
      DO i = 1, nconf
         call OCLPGC (i, fname)
         WRITE (tmp,'(I2)') i
         call OLINFO ('Configuration file '//tmp(1:2)//': '//fname)
      END DO

c---- TEST OSFI LOGGER
      call OLINFO ('      TESTING LOGGER')
      call OLPROG(3,4)
      call OLERR('Test OSFI Error Message')
      call OLWAR('Test OSFI Warning Message')
      call OLDEB('Test OSFI Debug Message')

c---- TEST OSFI PARAM READING
      call OPREAD('exampleFile.xml','',stat)
      if (stat.NE.1) call OLERR('Error Parsing '//cnfile)

c---- Check if parameter exists
      call OPEX ('los.LOS.polyParY', pexist)
      WRITE (tmp,'(L1)') pexist
      call OLINFO('Parameter exists: '//tmp(1:1))
c---- READING INTEGER SCALAR PARAMETER
      call OPINT(ivalue, 'earth.Earth.demType')
      WRITE(*,*) "Integer parameter value: ", ivalue
c---- READING BOOLEAN MATRIX PARAMETERS
      call OPBLM (bmatrix, nrows, ncols, 'los.LOS.flagsMatrix')
      DO i=1,nrows
         DO j=1, ncols
            p = (i-1)*ncols + j
            WRITE (*,*) 'row=', i, ' col=', j, ' ==> ', bmatrix(p)
         END DO
      END DO

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

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
```

4.5.7. Python Programming Language

Here is an example of Python code that uses the different modules of the integration libraries assuming that they are available either in the directory in which the interpreter is running, or in the list of directories contained in the PYTHONPATH environment variable or in the *sys.path* search path.

```
#!/usr/bin/env python

from __future__ import print_function # Py2/3 compatible code
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) # If given None, CLP will read sys.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))

    # Read the local configuration file
    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("[{0}][{1}] = {2}".format(i, j, mi[i][j]))

    vd = reader.getParameter(vectorDoubleParam).getVectorDouble()
    Logger.info(vectorDoubleParam)
    print(vd)

    return 0
  except Exception as e:
    Logger.error("TestModule failed: " + str(e))
    Logger.finishExecution(1)

if __name__ == "__main__":
  main() # CLP will read sys.argv itself
```

And here is the same example using the installed OSFI Python package as described in section 3.4.3.3.

As it can be seen, the only lines modified have been the *import* statements which are now done from the OSFI package, being all the rest of the code exactly the same as in the previous approach.

```
#!/usr/bin/env python

from __future__ import print_function # Py2/3 compatible code
from OSFI.ParamReader import ParamReader
from OSFI.CLP import CLP
from OSFI import Logger

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

  try:
    clp = CLP(argv) # If given None, CLP will read sys.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))

    # Read the local configuration file
    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("[{0}][{1}] = {2}".format(i, j, mi[i][j]))

    vd = reader.getParameter(vectorDoubleParam).getVectorDouble()
    Logger.info(vectorDoubleParam)
    print(vd)

    return 0
  except Exception as e:
    Logger.error("TestModule failed: " + str(e))
    Logger.finishExecution(1)

if __name__ == "__main__":
  main() # CLP will read sys.argv itself
```

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.List;

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

public class TestModel {

  public static void main(String[] args) {
    try {
      CLP clp = new CLP(args);
      List<String> cf = clp.getConfFiles();
      Logger.info ("Configuration files: " + cf);
      Logger.info ("Input files:         " + clp.getInputFiles());
      Logger.info ("Output files:        " + clp.getOutputFiles());

      ParamReader cfm = new ParamReader(cf.get(1)); // Parse LCF
      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");
      int[][] 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: " + e.getMessage());
      Logger.finishExecution(1);
    }
  }
}
```

5. COMPATIBILITY WITH PREVIOUS VERSIONS

Each release of OSFI is not guaranteed to be source or (for compiled languages) binary-compatible with previous versions. However, we do strive to keep source compatibility where possible, so that upgrading to a new version consists only of rebuilding against the latest OSFI. In particular, releases with the same minor version (e.g. 3.5.x) should be source-compatible.

In order to ease the upgrade path from previous versions of OSFI, this section details the incompatible API changes since the last version for each language. Note that, in general, only breaking changes are detailed here, with "breaking" defined as changes that cause a previously building source to fail to build, or to build but stop working. There may be other changes with a "soft" upgrade path, like deprecated functionality that raises a warning about the appropriate upgrade path. Such functionality will only appear in this section when it is finally removed from OSFI.

5.1. Migrating from OSFI 3.4 to 3.5

5.1.1. All/multiple Languages

Removal of support for attribute "ndims" (all languages)

Affected API: ParamReader/ConFM, Parameter constructors, Parameter.getDims

The "ndims" attribute was never part of [E2E-ICD], but it was recognized by and affected parsing behavior in several OSFI implementations. This attribute has been completely removed from OSFI 3.5, and is now ignored if present in configuration files. In particular, functions like Parameter.getDims now return consistent values across languages, while getNdims returns the length of the vector/list returned by getDims.

Given a file like with a parameter like the following:

```
<!-- Old-style array, one dimension but ndims=2 -->
<parameter name="x" ndims="2" dims="3" type="INTEGER">
    1 2 3
</parameter>
```

Implementations would previously have returned a variety of dimension arrays from getDims depending on whether "ndims" was being parsed or not (e.g. [3 0] in C++, [3 1] or [1 3] in other languages). In the new version, the "ndims" attribute is ignored and all implementations concur that the parameter shown is a **vector** of dimension 3.

As a consequence of the above changes, the constructors for Parameter instances in all languages no longer accept the "ndims" argument. In languages where arguments are purely positional, this is a breaking change that may either prevent building or fail at runtime, depending on the language. However, in most cases user code should not call Parameter constructors directly, so the impact to user code is likely to be small.

Parameter constructors now take different argument types (all languages)

Affected API: Parameter constructors

The specific effects depend on the language, but the "dims" argument is now a language-specific dynamic array instead of a string. For example, the argument has the type vector<int> in C++ and List<Integer> in Java, while in Matlab/Python an array/sequence of integral values is expected.

Scalar parameters now have zero dimensions (all languages)

Affected API: Parameter.getNdims, Parameter.getDims

For scalar parameters that do not have an explicit "dims" attribute, the default is to return zero from getNdims and the language-specific version of an empty integer list from getDims. This may be a breaking change for many users which may expect getNdims to never return zero, or for the dims return value to always have at least one element.

Xerces-C is now used privately (C++/FFI, C, Fortran)

Affected API: most OSFI-C++ headers

Xerces-C is the XML library used by OSFI-C++, and indirectly by OSFI-C, OSFI-Fortran and the deprecated OSFI-F77 to perform the low-level parsing of the configuration files. Until the previous version, this was an open fact exhibited by the OSFI CMake configuration and its headers. However, in the interest of encapsulation and a possible future move to a different XML library, the new version uses Xerces-C privately, as an implementation detail.

This means that, among other things, CMake target XercesC::XercesC is no longer part of the INTERFACE_LINK_LIBRARIES specification of OSFI-C++, so client executables or libraries declared to link against one of the mentioned OSFIs will **not** automatically get to the Xerces-C include path and library[5] injected in its own build settings.

This may be a significant breaking change if the user code attempts to perform its own XML parsing using Xerces but does not link against it itself, instead relying on the OSFI dependency. User code that uses Xerces should thus depend on it directly.

5.1.2. C++

Internal files and classes removed from public interface

Affected API: class ParameterParsingException (moved to different header), class StrX, class WriteErrorHandler, all functions in VectorTypes.h (removed), class XMLparser and macros in vt100.h (moved to private sources folder), class ParamReader and class UsageReader (inheritance tree modified).

The following headers have been removed, so trying to include them is now an error:

[5] If OSFI is built as a static library, it is possible that the Xerces-C library will still be a transitive dependency, since in many platforms static libraries are merely "object archives" and not truly "linked" until they are introduced into an executable or dynamic library.

- `ConFM/ParameterParsingException.h`, but **not** the class of the same name, which has been moved to the `ConFM/ParamReader.h` header
- `ConFM/StrX.h`, along with the class of the same name
- `ConFM/VectorTypes.h`, including functions intVector, doubleVector, fileVector, atob, str, boolVector and fileFormattedValue
- `ConFM/WriteErrorHandler.h`, along with the class of the same name
- `ConFM/XMLparser.h`, along with the class of the same name
- `EHLog/vt100.h`

These classes and/or functions were implementation details of other OSFI code, so they have been removed, either from the public API exported by the library, or altogether sicne they were made redundant by code rewrite. In particular, public API classes ParamReader and UsageReader no longer inherit from XMLparser, using it privately instead.

Xerces-C headers are no longer included by OSFI headers

Affected API: most OSFI-C++ headers

As detailed in the previous section, Xerces-C is now used privately in the CMake definition of the OSFI-C++ library. This means, among other things, that executables or libraries linking to OSFI will **not** get the Xerces include paths automatically, which forces OSFI to remove any mention of them from its own public headers.

Thus, user code using any Xerces type or function (e.g. XMLCh, DOMDocument) needs to ensure that the proper Xerces-C headers are included directly.

5.1.3. C

Removed some included headers from OSFIC.h, added include guard

Affected API: none directly ("collateral damage")

The OSFI C interface has been pruned and redundant code has been removed from the headers. In particular, the main C header is now wrapped in a double inclusion guard with the macro OSFI_C_INTERFACE. However, **references to stdio.h and stdarg.h have been removed** from OSFIC.h. Thus, code that inadvertently used functions or definitions from those files but did not include them directly will fail to build.

The solution is consistently including whatever headers your code uses, even if you think/know that they are already included by third party library headers.

```
#include "OSFIC.h" // No longer includes stdio.h (FILE, fopen)
#include <stdio.h> // Insert this or the module will no longer build

int main(int, const char**) {
    FILE* f = fopen("work.dat", "wb"); // Will no longer work
    //...
    return 0;
}
```

Bugfix in matrix getters output arguments for sizes

Affected API: all osfiConFMGetMatrix(T)Values functions, osfiConFMfileExist.

A bug causing matrix sizes to be incorrectly returned by OSFI-C matrix getters has been fixed. However, for code that depended on the returned (flipped) value of the "rows" and "columns" output arguments to those functions, the change will be breaking.

It should be noted that the osfiConFMGetColumns and osfiConFMGetRows functions did not exhibit this bug, so their return values have not changed.

5.1.4. Fortran

No breaking API changes exist between versions 3.4 and 3.5 of OSFI-Fortran, although the API continues to be fleshed out with new functions.

However, some bugs related to the bridge between Fortran and C++ (mostly off-by-one errors) have been fixed, which may be considered a breaking change if the code depended on a workaround.

5.1.5. Java

Renaming of OSFI package

Affected API: all of OSFI-Java

A major breaking change is that the OSFI-Java classes are now under a package named "**esa.opensf.osfi**" instead of simply "**osfi**", in application of ESA Java coding guidelines. This change obviously breaks both source and binary compatibility, but the fix is simply renaming references accordingly in both import statements and fully-qualified names:

```
import osfi.Parameter; // Remove this
import esa.opensf.osfi.Parameter; // Replace with this
```

Parameter returns primitive arrays where appropriate

Affected API: Parameter.getVectorT and getMatrixT, with T = (Int, Double, Boolean)

After an overhaul of parameter parsing, the Parameter class will no longer return arrays of boxed types like Integer or Boolean. Instead, those methods will return arrays of primitive types. Types that return arrays of strings are not affected. This change speeds up parsing of large parameter arrays, since the process directly generates primitive arrays with contiguous values, instead of arrays of references to possibly scattered values.

```
Parameter p = ...;
Integer[][] val = p.getMatrixInt(); // Remove this
int[][] val = p.getMatrixInt(); // Replace with this
```

A possible secondary effect of this change is that **primitive arrays do not play nicely with some collection-utility methods** such as java.util.Arrays.asList, so code that relied on such methods to wrap the returned arrays with a List<T> will no longer work.

The solution is twofold: if your module is using Java 8+, you can probably switch processing code to the Stream API, using Arrays.stream() and IntStream instead of

Arrays.asList() and List<Integer>. Otherwise, or if a List of a wrapper type is absolutely required, you can either do the conversion yourself or use a supporting library like Guava.

```
// This won't work anymore b/c java.util.Arrays does not produce
// a List<Integer> from an int[] argument – T must extend Object!
List<Integer> li = Arrays.asList(p.getVectorInt());

// Solution 1, for Java 8+: switch to Stream API and work with it
// either directly or as a way to obtain a List.
IntStream is = Arrays.stream(p.getVectorInt());
int minPosVal = is.filter(v -> v>0).min().orElse(0); // Work directly
List<Integer> li1 = is.boxed().collect(Collectors.toList()); // List

// Solution 2a: manual conversion to Integer array
List<Integer> li2a = new ArrayList<>();
for (int v : p.getVectorInt())
    li2.add(v);

// Solution 2b: automatic conversion/wrapping with utility methods
// from external libraries e.g. Guava or Apache Commons Lang.
List<Integer> li2b = Ints.asList(p.getVectorInt()); // From Guava
```

New API for ARRAY-typed variables

Affected API: Parameter.getArrayValue (removed)

The parsing of structured types in OSFI-Java has been rewritten in this version. The previous API was incoherent with the rest of the OSFI-Java interface, since unlike the other functions to retrieve a value, the getArrayValue function provided a single point without the possibility to get a typed result, always returning an Object array.

Furthermore, the previous implementation introduced a confusing permutation of dimensions for 3-D arrays, relabeling the outermost dimension of such an array as the "third" dimension instead of the first as would be customary in Java.

The new API is introduced under the name **getArrayRootNode**, and is similar in design to the C++ version. It exposes a tree of ArrayNode.Raw instances that can be navigated starting from the 1st dimension (formerly the 3rd) or parsed in a type-safe fashion into either a flattened array V[] or a parsed tree structure ArrayNode<V[]>. See §4.3.8 for the detailed interface.

Given a configuration file with a parameter like:

```
<parameter name="arr" dims="2" type="ARRAY" elementType="INTEGER"…>
    <parameter dims="3" type="ARRAY">
        <parameter dims="4" type="ARRAY">1  2  3  4</parameter>
        <parameter dims="4" type="ARRAY">5  6  7  8</parameter>
        <parameter dims="4" type="ARRAY">9 10 11 12</parameter>
    </parameter>
    <parameter dims="2" type="ARRAY">
        <parameter dims="4" type="ARRAY">-1 -2 -3 -4</parameter>
        <parameter dims="4" type="ARRAY">-5 -6 -7 -8</parameter>
    </parameter>
</parameter>
```

Previous versions of the OSFI-Java API would have consumed the parameter with code similar to the following:

```
Parameter p = pr.getParameter("arr");

// Both calls return Object[] with Integer elements, not Integer[]!
Object[] a = p.getArrayValue(0, 1)); // row=0, thirdDimension=1
System.out.println(Arrays.toString(a)); // Prints "[-1, -2, -3, -4]"

Object[] b = p.getArrayValue(1)); // row=1, thirdDimension=0
System.out.println(b[2]); // Prints "7"

// Obtaining an array of primitives requires copying the data
int[] b_prim = new int[b.length];
for (int i=0; i < b.length; ++i)
    b_prim[i] = b[i]; // Auto-unboxing in Java 5+
```

However, in the new API the array root node does return **primitive arrays** (int[], etc.) for the appropriate element types. User code may choose to retrieve and parse only a certain slice of the parameter, as in the first example; or to parse the full parameter and *then* access whatever slices are needed, as in the second example.

```
Parameter p = pr.getParameter("arr");

// Now, getArrayRootNode returns an ArrayNode.Raw instance, which
// can be indexed first and then parsed partially...
ArrayNode.Raw raw = p.getArrayRootNode()
int[] a = raw.getSubNodeAt(1,0).getVectorInt(); // layer=1, row=0
System.out.println(Arrays.toString(a)); // Prints "[-1, -2, -3, -4]"

// ... or parsed as a full tree and then indexed into.
ArrayNode<int[],?> parsed = raw.getTreeInt();
int[] b = parsed.getDataAtSub(0, 1);
System.out.println(b[2]) // Prints "7"
```

Collection classes replaced by interfaces in API

Affected API: CLP, ParamReader.getParameters, Parameter.getDims (ArrayList to List), Parameter.getOtherAttributes (HashMap to Map).

Instead of taking and/or returning concrete collection classes (ArrayList, HashMap), the API now works with the corresponding interface (List, Map) in order to improve encapsulation. In cases where the API *takes* one such argument, existing code is source-compatible, but for *return values* the compilation may fail if use code expects a collection class to be returned.

```
ArrayList<Parameter> params = pr.getParameters("group"); // Old
List<Parameter> params = pr.getParameters("group"); // New
```

Internal classes and methods removed from public interface

Affected API: Vt100, XMLParser, Logger.readFile

These classes were implementation details of Logger and ParamReader, respectively, so they have been removed from the public API exported by the library. If your code absolutely must use such a class, you can find it in the OSFI source.

5.1.6. Python

As in other languages, the parameter parsing code was rewritten in this version of OSFI-Python. Most of the changes are non-breaking, such as the fact that getMatrixX functions no longer raise Exception but the subclass TypeError.

Removal of the duplicated "constructor" in Parameter

Affected API: Parameter.init (removed), Parameter.__init__

As part of the removal of support for the "ndims" attribute, the function Parameter.init was also removed from the OSFI-Python interface. This was an unpythonic pseudo-constructor that allowed a Parameter object to be "reinitialized" after being created. Instead, if changes are needed to the basic attributes of a Parameter object, it should be replaced with a newly initialized one. The main breaking change could surface if it was paramount that the *same instance* of the class was modified.

For example, if a module variable is created and then the code wants to re-initialize the parameter, Python will assume that the assignment to the name (when using the new constructor-based syntax instead of the old function) creates a local variable instead. In this case, the workaround is simply telling Python that the variable to be assigned to is the module-scoped one, using the "global" keyword as in the example.

Note, however, that this is still a workaround, since the module-scoped variable no longer points to the same instance of Parameter as it did before.

```
# Create a module-level Parameter object and then write to it
p = Parameter(…) # Create with some data at module level

def func_that_alters_module_var(newVal, newLen):
    # Previously: reinitialize with special function
    p.init("name", "description", "INTEGER", newVal,
           "", "", "", "1", str(newLen), None)
    # New version: replace object with constructor
    global p # So that we don't create a local variable p instead
    p = Parameter("name", "description", "INTEGER", newVal, newLen)
```

Parameter constructor arguments renamed

Affected API: Parameter.__init__

The following arguments to the Parameter constructor have been renamed:

- aName → name
- aDescription → description
- aType → elType
- aValue → value

Since the mentioned arguments are compulsory, the renaming will not affect user code providing them via positional syntax, but it is a breaking change for code that tries to pass the old parameters via keywords.

Getters for scalar values now return None on failure

Affected API: Parameter.get(T)Value function, for all T

The previous version of OSFI returned a type-specific default value if the parameter value could not be parsed. This could result in correct but unintuitive behavior, like "true" being parsed as False (since the correct value is only "TRUE", in capitals). Instead, the new version returns None in such situations.

While this change is compatible in many common situations, like Boolean evaluation, it may be breaking depending on the usage made by user code and the version of Python, as shown in the example below:

```
val = pr.getParameter("param").getIntValue()

# This code for a user code-specific default will still be valid
v = val if val else 7
# The new None return also allows telling an actual "0" from an error
v = val if val is not None else 7

# However, this will fail in Python 3, since NoneType is no longer
# comparable to int
sig = 1 if val >= 0 else -1
```

New API for ARRAY-typed variables

Affected API: Parameter.getArrayValue, Parameter.getValue

Like in other languages, the parsing of structured types has been reimplemented in this version. In 3-D arrays, the first dimension in the file is no longer permuted to the 3rd dimension in the parameter, so for any code that read such arrays, this change is breaking.

Given a configuration file with a parameter like:

```
<parameter name="arr" dims="2" type="ARRAY" elementType="INTEGER"…>
    <parameter dims="3" type="ARRAY">
        <parameter dims="4" type="ARRAY">1  2  3  4</parameter>
        <parameter dims="4" type="ARRAY">5  6  7  8</parameter>
        <parameter dims="4" type="ARRAY">9 10 11 12</parameter>
    </parameter>
    <parameter dims="2" type="ARRAY">
        <parameter dims="4" type="ARRAY">-1 -2 -3 -4</parameter>
        <parameter dims="4" type="ARRAY">-5 -6 -7 -8</parameter>
    </parameter>
</parameter>
```

The indices used to address the sections of the array change so that the dimension that is actually first in the file also becomes the first in the code, instead of being permuted. The combination of that behavior with the defaulting of the "third dimension" (actually first) to 1 will cause different results, since the new API will, if insufficient indexes are given to return a single vector or element, return a cell array with subtree of values.

```
p = pr.getParameter("arr")

# Previous syntax with "third dimension" for the outermost layer
print(p.getArrayValue(0, 1)) # row, thirdDimension -> "[-1,-2,-3,-4]"
print(p.getArrayValue(1)) # row=1, thirdDimension=0 -> "[5,6,7,8]"

# Currently, the same slices must be addressed like:
print(p.getArrayValue(1, 0)) # Now prints "[-1,-2,-3,-4]"
print(p.getArrayValue(0, 1)) # Now prints "[5,6,7,8]"

# The function also allows returning larger slices or single elements
print(p.getArrayValue(1)) # Prints "[[-1,-2,-3,-4],[-5,-6,-7,-8]]"
print(p.getArrayValue(0,2,3)) # Now prints "12"
```

Furthermore, the getValue function that returns the unparsed value of the parameter will now return an ArrayNode instance that allows user code to examine the structure of the unparsed strings at each level of the parameter.

The getValue function now returns the parsed value

Affected API: Parameter.getValue

The function returning the raw (unparsed) value of a parameter has been renamed getRawValue for uniformity across OSFI implementations. The getValue function now returns the parsed value with the declared type (which can be overridden) and dimensionality/structure.

```
# Create a module-level Parameter object and then write to it
p = pr.getParameter('integerParam')

rawVal = p.getValue() # This was a string before, but is now an int
rawVal = p.getRawValue() # This is a string
intVal = p.getValue() # New interface for general parsed value
dblVal = p.getValue(asType=ParamType.FLOAT) # Type can be overridden
```

The actual return type of the new getValue depends on the dimensionality and ARRAY-ness of the parameter: a scalar parameter will return a single instance of the correct type, while a vector (matrix) will return a list (of lists) of such instances. The return value for ARRAY parameters is the same as calling getArrayValue, that is, nested lists representing the parameter structure in the XML.

5.1.7. Matlab

New Parameter parsing engine: removal of N/A and str2num usage

Affected API: ConFM, Parameter

Previously, many fields of a Parameter instance used the str2num function to parse inputs. In particular, the dims, min and max properties, and also the value for numeric and Boolean parameters, were parsed in this manner.

This implementation caused several Matlab-specific inputs to be allowed, like "1" and "0" for Boolean parameters or "6+2" for a numeric parameter. However, it is also an **important security problem**, since any Matlab code (possibly malicious) was *also* a

valid input. Thus, it has been removed as part of a complete overhaul of parsing code in the new OSFI-Matlab version, which is now a much closer match to [E2E-ICD]. This means that, among other changes:

- Structured types no longer consider the "N/A" string as a missing element placeholder. For string-like types, it is interpreted as a normal value, while for other types it triggers an error.
- Boolean parameters only accept "TRUE" or "FALSE" as values.
- Integer parameters reject non-integer values like "3.7", "Inf" or "NaN".

While not a syntactic API break in the sense of changing the names or arguments of the functions in the OSFI-Matlab interface; this is an important modification to the semantics of the API and thus may cause modules that depended on some specific behavior of the previous version to fail with an unchanged configuration file. In general, a file that was able to be parsed by a non-Matlab OSFI in a previous release should still be parseable with the new Matlab engine.

New Parameter parsing engine: types of values

Affected API: Parameter.getValue, Parameter.getArrayValue

In order to more closely match the specification in [E2E-ICD], the new parsing engine returns values of Matlab type int32 instead of double if a parameter is of type INTEGER. This change may be breaking in certain cases where arrays of values are compa

```
p = pr.getParameter('integerParam');

valueOk = p.getValue() - [1 2 3]; % Error: integer - double vector
valueOk = p.getValue() - int32([1 2 3]); % New format
```

New API for ARRAY-typed variables

Affected API: Parameter.getArrayValue

Like in other languages, the parsing of structured types has been reimplemented in this version. In 3-D arrays, the first dimension in the file is no longer permuted to the 3rd dimension in the parameter, so for any code that read such arrays, this change is breaking.

Given a configuration file with a parameter like:

```
<parameter name="arr" dims="2" type="ARRAY" elementType="INTEGER"…>
    <parameter dims="3" type="ARRAY">
        <parameter dims="4" type="ARRAY">1  2  3  4</parameter>
        <parameter dims="4" type="ARRAY">5  6  7  8</parameter>
        <parameter dims="4" type="ARRAY">9 10 11 12</parameter>
    </parameter>
    <parameter dims="2" type="ARRAY">
        <parameter dims="4" type="ARRAY">-1 -2 -3 -4</parameter>
        <parameter dims="4" type="ARRAY">-5 -6 -7 -8</parameter>
    </parameter>
</parameter>
```

The indices used to address the sections of the array change so that the dimension that is actually first in the file also becomes the first in the code, instead of being permuted. The

combination of that behavior with the defaulting of the "third dimension" (actually first) to 1 will cause different results, since the new API will, if insufficient indexes are given to return a single vector or element, return a cell array with subtree of values.

```
p = pr.getParameter('arr');

% Previous syntax with "third dimension" for the outermost layer
disp(p.getArrayValue(1, 2)); % row, thirdDimension -> "[-1 -2 -3 -4]"
disp(p.getArrayValue(2)); % row=2, thirdDimension=1 -> "[5 6 7 8]"

% Currently, the same slices must be addressed like:
disp(p.getArrayValue(2, 1)); % Now prints "[-1 -2 -3 -4]"
disp(p.getArrayValue(1, 2)); % Now prints "[5 6 7 8]"

% The function also allows returning larger slices or single elements
disp(p.getArrayValue(2)); % Prints "{[-1 -2 -3 -4] [-5 -6 -7 -8]}"
disp(p.getArrayValue(1,3,4)); % Now prints "12"
```

END OF DOCUMENT