Deliverable Reference	:	D4.4
Title	:	Preliminary Design Document
Confidentiality Level	:	PU
Lead Partner	:	SPACEAPPS
Abstract	:	This deliverable gathers the outcomes of the preliminary design of InFuse, consisting in a large extent by material released in D4.2 as well as D4.1 and D4.3.
EC Grant N°	:	730014
Project Officer EC	:	Christos Ampatzis (REA)

InFuse is co-funded by the Horizon 2020

Framework programme of the European Union

DOCUMENT APPROVAL SHEET			
	Name	**Organization**	**Date**
Prepared by:	Raul Dominguez Shashank Govindaraj Jeremi Gancet Simon Lacroix Fabrice Souvannavong Michal Smicek Mark Post	DFKI SPACEAPPS CNRS-LAAS MAG DLR USTRATH	22/06/2017
Cross-reviewed by:	Jeremi Gancet Shashank Govindaraj	SPACEAPPS	23/06/2017

Using InFuse Preliminary Design deliverables in Perspective of the H2020 Space Robotics Technologies SRC Call 2

Overview of the Preliminary Design WP deliverables' content and purpose/logics within InFuse

D4.1: The goal of D4.1 is to present the choices of the data processing and data fusion functions that will be developed within InFuse.

After an analysis of operational scenarios which exhibit the required InFuse outputs, the list of these outputs ("data products") is defined. For each data product, a brief analysis of possible solutions is made, and a solution is selected. Each solution is defined by a "Data Fusion Process Compound" (DFPC), and is described with a coarse level of precision. The refinement of the DFPCs into a series of elementary "Data Fusion Nodes" (DFNs, i.e. elementary data processing functions), and the definition of the organisation of the DFNs to compose a DFPC is matter for further work for the InFuse project.

The DFPCs are defined for both the planetary and orbital reference implementations. While both implementations mostly require different DFPCs, many DFNs will be shared between the two targeted contexts.

The document also presents a sets of operational scenarios for each reference implementation. They have the purpose of putting in context the proposed DFPCs.

D4.2: This deliverable focuses on the one hand on software architecture considerations, with the preliminary specification of all the key components that the InFuse CDFF consists of, and on the other hand on the identification of all relevant interfaces, both internal to InFuse and the external ones with respect to other OGs (OG2-ERGO and OG4-I3DS) in particular.

Note that the architecture and ICD material introduced in this document are essentially application independent – application specific considerations will be introduced at a later stage, during the detailed design of the InFuse CDFF (each of the orbital track and plenatery track will then be tailored, accordingly).

This deliverable will serve as a starting point for the detailed design work, in the following work package.

D4.3: The purpose of this document is to define the strategy and overall approach for the testing activities to be carried out internally (OG3), so that to ensure that the developed software is sound while meeting the requirements expressed in the earlier phases of the project. The content is orbital / planetary independent: it is not in the scope of this deliverable to specify the testing approach with facilities and EGSEs specific to either the orbital or planetary scenarios. These aspects will instead be addressed in the detailed design activities that are just starting as this document is being written, and will be released by the CDR milestone.

D4.4 (this deliverable): This deliverable is basically a compilation of essential information from D4.1, D4.2 and D4.3 into a single document. Note that some parts of D4.1, D4.2 and D4.3 may therefore not be appearing in the D4.4.

Relation between WP4 deliverable's content and other "common building block" OGs

The scope of OG3-InFuse lies essentially between OG4-I3DS, that produces raw sensor data, and OG2-ERGO, which controls all the rover activities. The interfaces with these two OGs are defined in the deliverable D4.2:

- The interfaces with OG4-I3DS are defined by sensor data types
- The interfaces with OG2-ERGO are defined on the one hand by the types of data products (mainly terrain maps and localization information related to the rover and the terrain, and to the chaser and target satellites), and on the other hand by requests made by OG2-ERGO to OG-InFuse.

OG2-ERGO is interfaced with OG3-InFuse at the granularity of the DFPCs: OG2-ERGO has no view of the inner mechanisms of OG3-InFuse that assembles DFNs into a DFPC. An OG2-ERGO request triggers the activation of a DFPC (which can either be synchronous or not), along with given parameters, and the DFPC returns the requested data products with an execution report. Within a DFPC, the DFNs are assembled, sequenced and triggered via pre-defined scripts, which are configured according to the parameters associated to the OG2-ERGO requests.

The definition of the DFPCs provided in D4.1 is generic, and makes no hypothesis regarding the integration middleware within which they will be developed. The way the developed functions will be integrated within the OG1-ESROCOS framework (and potentially other target middlewares) is depicted in the document D4.2.

Finally the content of D4.3 is largely centered on InFuse internal testing and validation activities – in that context, “integrated test plans” deal with the joint testing of several sub-parts of the InFuse framework, not InFuse and other OGs. Still, several components of the InFuse CDFF have interfaces with other OGs – mainly OG2-ERGO and OG4-I3DS. For these ones, it is foreseen in the test plan to develop specific components as placeholders of ERGO and I3DS, exposing the interfaces that are assumed to be the ones with which InFuse should integrate in the upcoming Space Robotics SRC projects. We call them M-OG2 and M-OG4 (M standing for Mock). Their purpose again is only to make it possible, internally, to carry out end-to-end tests and ensuring the soundness of the CDFF interfaces.

Applicability to the H2020 Space Robotics Technologies SRC upcoming calls (i.e. OG7 to OG11a/b).

Environment perception, be it to model the environment or to localize the controlled robotic platform within this environment, is at the core of autonomous operations, and is therefore required in all the applications defined by the upcoming calls.

For each of the future Operational Grants, a set of relevant DFPCs identified in D4.1 document is identified below as having particular relevance (note however that this list is based on very preliminary, and still limited knowledge of the content and scope of each upcoming OG).

OG7: (Orbital Support Services): Long/Mid/Close-range Tracking and Detection, 3D Target reconstruction, Point Cloud based Localisation

OG8: (Modular Robotized Assembly): Mid/Close-range Tracking and Detection, Point Cloud based Localisation

OG9: (Satellite Re-configuration): Mid/Close-range Tracking, Point Cloud based Localisation

OG10: (Advanced Autonomy): DEM Building + Soil Type Map, and all the rover localisation DFPCs: Visual Odometry, Visual/LIDAR based SLAM, Scientific Area Localisation, Visual Map-based Localisation, Absolute Localisation.

OG11a: (Advanced Mobility): DEM Building + Soil Type Map and Visual Odometry for extreme terrain mobility, plus all the localisation DFPC for the coordination of multiple platforms.

OG11b: (Robotized Construction): The required DFPCs for this OG are certainly similar to some of the ones required for the orbital OGs: Long/Mid/Close-range Tracking and Detection, 3D Target reconstruction, Point Cloud based Localisation – though in planetary context. DEM building and rover localization DFPCs remain relevant.

Deliverable D4.2 provides the latest baseline about the InFuse CDFF architecture and ICD. In perspective of the next OGs, it is essential to understand the proposed architecture and mechanisms to handle data, and the proposed approach to generate middleware specific reference implementations from the vanilla (middleware independent) CDFF environment.

In perspective of the upcoming OGs, D4.3 has limited relevance as its purpose is essentially to define the InFuse internal strategy to test the various components of the CDFF. It is therefore of little use for what concerns the preparation of bids for the next call, and similarly would have limited relevance for the future implementation of the next OGs.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose and scope

This document provides a compilation of the key outcomes of InFuse, as of the PDR milestone. It essentially deals with the architecture of the InFuse CDFF (i.e. D4.2 related material), but also contains relevant information in preparation to the PDR review. More material is available in other deliverables – D4.1, D4.2 and D4.3.

1.2 Document structure

In short this deliverable is structured as follows:

Section 1: this introduction.

Section 2: Operational scenarios

Section 3: Architecture design rationales

Section 4: Proposed architecture

Section 5: Detailed description of the architecture

Section 6: Inter-OGs interfaces

Section 7: Recommended coding style

1.3 Applicable documents

AD1	InFuse Grant Agreement (Confidential document)
AD2	InFuse Consortium Agreement (Confidential document)

1.4 Reference documents

RD1	D4.1 Technical Tradeoffs Analysis (Public – available on demand)
RD2	D4.2 Advanced CDFF Architecture and ICD (Public – available on demand)
RD3	D4.3 CDFF Unitary and Integrated Test Plan (Public – available on demand)

1.5 Acronyms

API: Application Program Interface

CDFF: Common Data Fusion Framework

DEM: Digital Elevation Map

DF: Data Fusion

DFN: Data Fusion Node

DFNCI: Data Fusion Node Common Interface

DFPC: Data Fusion Process Compound

IMU: Inertial Measurement Unit

RCOS: Robotics Control Operating System (e.g. ROS, Rock, GenoM)

MW: Middleware. In this document is used as synonym for RCOS.

PCL: Point Cloud Library

RI: Reference Implementation

2 Operational scenarios

Elements of scenarios can be found in the SRC Compendium of activities [Compendium]. For the planetary track, the following operational requirements are defined:

- Depart from a landing spacecraft,
- Reach towards another planetary asset several kilometers away,
- Perform autonomous science while traversing,
- Rendezvous with another planetary asset.

For the orbital track, the defined operational requirements are:

- Rendez-vous and capturing, which encompasses close perception, capturing and docking
- Exchange of a payload module between the chaser and the target.

From these operational requirements, the Infuse Consortium has proposed five scenarios in the systems requirements document [InFuse_D3.2], aiming at allowing the verification that the identified requirements are suitably implemented.

Besides, two reference scenarios for planetary exploration and in-orbit servicing have been defined by the OG2-ERGO consortium in the system requirements document [ERGO_D1.2]. These two complete scenarios integrate all the required functionalities for the two reference implementations, and depict the products that OG2-ERGO requires from InFuse (note that other elements of scenarios are defined in [FACILITATORS_D1.2], mostly for validation purposes).

Here, we complement the description of the scenarios introduced in [InFuse_D3.2], aiming at enumerating all the data products that InFuse has to deliver. This list of data products is the basis upon which the technical trade-offs are made, in section 3, “Baseline solutions".

2.1 Planetary track

The planetary track reference implementation encompasses a large variety of functionalities. For the ease of the analysis, instead of defining a general scenario that requires all these functionalities, we have chosen in [InFuse_D3.2] to define 5 different scenarios, each exhibiting a specific set of functionalities. The first scenario (“Long traverse”) is naturally the most important one, the four others defining extensions of this one:

- Long traverse: the objective of the mission for the rover is to autonomously reach a target located about 1km away, defined by its absolute coordinates;
- Autonomous science: while navigating, the rover processes acquired data so as to detect and localize potential scientific targets;
- Rendez-vous: the goal is here to precisely position the rover with respect to an existing asset (*e.g.* the lander);
- Getting back: this consists in getting back to the initial position after a long range traverse;
- Cooperation: this scenario involves the use of a second mobile robot (*e.g.* a scout-rover, or a UAV) to assist the rover during long traverses.

2.1.1 Long traverse

This scenario is the basic one upon which all the others are defined.

2.1.1.1 Mission definition

The objective of the mission for the rover is to autonomously reach a target located about 1km away, defined by its absolute coordinates.

The mission is defined by an path specified for the traverse, with constraints to satisfy:

- Localisation constraints: each part of the path (or each waypoint to reach) is associated to a requirement on the precision of the localisation;
- Time constraints: the path is specified with bounds on time to satisfy

The path is specified either by a corridor or a series of waypoints.

2.1.1.2 Mission preparation

The mission is defined by an operator on the ground (possibly assisted with planning tools), who considers a series of constraints to satisfy and criteria to optimize. For this purpose, various information have been considered, such as terrain traversability, sun illumination, possibility to absolutely localize the robot. These information come from orbiter data and a priori knowledge and models, but can also encompass information gathered by the rover in previous missions.

2.1.1.3 Mission execution

The rover follows a nominal navigation cycle, that is sequences of short term path planning and trajectory following phases.

- Short term path planning. This requires the "Fused Rover Map" (Digital Elevation Map structure);
- Trajectory following:
    - Localisation at 10Hz to ensure satisfactory path following;
    - Hazard avoidance. This requires the production of the "Rover Map".

In addition to the nominal navigation cycle, the following phases are also planned:

- Upon request from OG2, or automatically triggered according to some criteria (e.g. precision of the current global position estimate, length of the executed path): absolute localization is made avaiable;
- End of the mission: If specified in the mission, build a Fused Total Map. This map may serve as initial information for further mission planning, *e.g.* for the getting back scenario.

2.1.1.4 Associated data products

The data products associated to this scenario are the following:

- Three kinds of Digital Elevation Maps, that differ from the amount of fused data, their spatial extent, their reference frame, and the frequency of their update:
    - ”Rover Map”;
    - “Fused Rover Map”;
    - “Fused Total Map”.
- Two localization services, that differ for the frame in which they estimate the rover pose and the frequency at which the output pose estimates:
    - @ 10Hz when driving;
    - Absolute localization (using the orbiter map).

Notes:

- each data that has to be integrated within the DTM must be precisely localized, so that the produced DTM is spatially consistent.
- If required by OG2-ERGO, in order to help the production of Navigation Maps, the DTM can be complemented with a Soil Type Map (mostly derived from luminance information).
- the "getting back" scenario (section 2.1.4) introduces the need to build a specific data structure (a "localization map") during the execution of the long range traverse. This data structure (which can be defined upon the built DTMs) is not exported to OG2-ERGO.

Section 2.3.2 discusses these points.

2.1.2 Autonomous science

Rather than an actual operational self-contained scenario, here "Autonomous science exploration" is considered as an additional functionality that can be triggered during the "Long traverse" scenario.

2.1.2.1 Mission definition

The mission consists in activating a process of scientifically relevant target detections, in given areas during the achievement of an autonomous traverse scenario.

2.1.2.2 Mission preparation

The details of the mission are defined by an operator (possibly assisted by mission planning support tools), along with the definition of the long traverse scenario. On the basis of similar information that are used to prepare the long traverse, the operator defines:

- The areas within which the scientific target detection process must be activated;
- The specification of the kind of scientific target to be detected.

The specification of the kind of target to be detected are limited to the sensorry information available and corresponds to the selection of the sensor(s) and detection process(es) to activate, because there may co-exist a variety of detection processes within the system. Note that the data acquisition may require pointing the sensor(s) or making a panorama: the priori planning and on-line adaptation of these options, if retained, does not pertain to OG3-InFuse, but to OG2-ERGO.

2.1.2.3 Mission execution

The mission execution simply consists in activating the selected target detection process and corresponding modules in the defined corresponding areas.

2.1.2.4 Associated data products

- The single data product associated to this scenario is the localization of the detected scientifically relevant target(s). A target is localized with respect to the rover, its spatial extent is provided (bounding box).

2.1.3 Rendezvous

2.1.3.1 Mission definition

The mission consists in reaching a precise position and orientation with respect to a man-made asset, e.g. the sample analysis module, for instance so that a sample can be transferred.

2.1.3.2 Mission preparation

The rover is supposed to be in the close vicinity to the asset. In the general case the terrain to traverse to achieve the Rendezvous with the asset is not known - yet it could have been previously modeled, and hence may be known, this does not change the gist of this scenario. The mission preparation consists in specifying a given position and orientation relatively to the asset to reach, as well as the trajectory the rover must follow to reach it. The preferred rendezvous feature on the asset will also need to be specified.

2.1.3.3 Mission execution

The rover executes the planned trajectory (using the basic nominal navigation cycle if the terrain is known). To insist on the specificity of this scenario, we considered that the rover is in a close range situation with respect to the asset to reach, and so that it must localize itself with respect to the asset, using specified features that are detected on the asset.

2.1.3.4 Associated data products

- The single data product associated to this scenario is the relative pose of the robot with respect to the asset to approach, produced at a frequency allowing fine trajectory control (10 Hz, to be confirmed).

Note: this localization is produced by a dedicated localisation service, which exploits features detected on the asset to reach. This service is analogous to the one required by the Orbital RI scenarios, and hence shares common processes.

2.1.4 Getting back

This scenario is the reverse traverse that can be defined after the execution of a long traverse mission, or any other trajectory formerly executed.

2.1.4.1 Mission definition

The objective of the mission for the rover is to autonomously execute rearward a trajectory that has been executed beforehand (*e.g.* after a long traverse, or after having fetched a sample or performed a scientific analysis). The benefit of this ability is to achieve traverses without resorting to path planning, nor absolute localization: in theory only the trajectory following process, supported by the knowledge of potential hazards identified previously in the Rover Maps, drives the rover, thus enabling faster motions (yet, for safety reason the hazard detection process can be required, thus requiring the production of Rover Maps).

The mission is defined by:

- The trajectory to follow;
- Constraints on the localisation precision defined along the trajectory;

2.1.4.2 Mission preparation

The mission is prepared on the basis of data collected during the forward traverse, upon which the trajectory to execute is defined, and with which localization can be ensured with the required accuracy during the execution of the trajectory. The later is encoded within a *Localization Map*, a specific data product that is not exported to OG2-ERGO.

Note: the trajectory to execute might be slightly different from the one that has been executed: it can indeed be replanned by OG2-ERGO using the Total Rover Map.

2.1.4.3 Mission execution

The execution of the mission consists in following the defined trajectory exploiting the Localisation Map and the Fused Total Map from the previous trip before going back. Possibly, a local Rover Map can be produced at a given spatial frequency.

2.1.4.4 Associated data products

The single data product associated to this scenario is:

- Localisation with respect to the “localization map”;

Note: Additionally, if required local Rover Maps can be produced and exported to OG2-ERGO at a given spatial frequency.

2.1.5 Cooperation

This scenario is the most complex one, and also the most prospective. Its definition remains quite open (it has not been depicted in any other OG deliverables), and instead of a complete scenario definition, we briefly depict some cooperation schemes, aiming at defining the associated InFuse products.

The consideration of such schemes is here mainly to assess that the actual definition and achievement of such scenarios can be supported by InFuse, without requiring to update the InFuse architecture or to develop brand new functionalities. The structure of this section is therefore less precise that for the other scenarios.

2.1.5.1 Perception and navigation functionalities for cooperation

Cooperative schemes can be defined to achieve the whole variety of missions that a single rover can be tasked with, such as exploring / mapping an area, long range traverse, or a declination of the getting back scenario in which one robot executes a trajectory, formerly executed by another. The cooperating robots may be heterogeneous, *e.g.* a UAV can act as a scout for a rover, and the cooperation schemes are of course not limited to two robots. Without going into further details, the functionalities related to perception (terrain modelling) and navigation (localization) that can be used to build cooperation scenarios are the following:

- Fusion of data gathered by the different robots into a Digital Elevation Map. Here the fusion can be made along a tight scheme (incorporating point-clouds), or a loose scheme (integrating

DTMs independently produced by the robots with their own-acquired data). Note that the data / DTMs need to be registered beforehand (see below)

- Inter-robot localization. This can be achieved by two different means:
    - Direct localisation of one robot with respect to the other (when both robots are in sight);
    - Indirect localisation, by registering point-cloud data (of DTMs, or point-cloud data with a DTM). This can be achieved even when the robots are not in direct sight.

  These inter-robot localization information are used to refine each robot position estimates.

2.1.5.2 Associated data products

The data products associated to cooperation scenarios are exactly the same than for the former scenarios: DTMs (Fused Rover and Total Rover Maps), and localization estimates at low frequency (akin to Absolute Localization). The way to generate these data products will not be further detailed, yet they will be slight derivatives from the DFNs and DFPCs retained for the basic scenarios.

2.2 Orbital track

Here, instead of associating a scenario to the various operational concepts, a single global scenario is depicted. Also, since the validation of the developments will require a dedicated setup (robot manipulators), the use of this setup is mentioned in the various scenario phases.

The scenarios in orbital track can be found in the SRC Compendium of activities [Compendium], where the following operational requirements are defined:

- Detection of a target spacecraft;
- Far-range rendezvous ;
- 3D mapping (3D model);
- Close-range approach.

2.2.1 Mission definition

The orbital track reference implementation consists in approaching a target spacecraft (mainly at close range), which can be simulated with a hardware in the loop simulator. The chaser spacecraft needs to identify, track and approach a target satellite to perform in orbital servicing operations under the following constraints:

- localization constraints: each part of the path is associated to a requirement on the precision of the localisation. Thus, the localization constraints are defined at various ranges, which may require bearing only tracking, 3D tracking and detection;
- range constraints: the localization is specified with bounds on range to satisfy approach requirements;
- time constraints: the range constraint is specified with bounds on time.

2.2.2 Mission preparation

With pre-conditions described in [InFuse_D3.2] such as preparation of dynamic test bench, calibration and pre-configured OBC with CDFF software, the target satellite will be first detected, knowing its geometric model and using calibrated cameras. The satellite model will be pre-processed offline to reduce computational complexity of the tracking software. In order to achieve the defined scenarios, robot manipulators in the dynamic test bench will be powered up along with all the supporting motion control system, real time simulator, monitoring and debugging system and the OBC running the CDFF software stack. The sensors are activated at the start of the test run and the CDFF software is started to perform a sanity check on the flow of sensor data to the CDFF software and forward it to a pre-defined set of data fusion methods that begins to acquire sensor data for processing.

2.2.3 Mission execution

The robot manipulators of the dynamic test bench perform a series of pre-defined trajectories and goes to a pre-defined starting position. The ground truth which is obtained from robot forward kinematics and if available by an additional tracker system provides the absolute pose of the spacecraft mockups.

The mockup of a chaser spacecraft will be positioned initially at the maximum possible distance from that of a target spacecraft. The illumination of the setup is activated and if applicable together with the multiple-axis free rotations of the mockup to replicate a freely floating satellite in orbit. The demonstration of the CDFF will begin by evaluating a baseline implementation of localization and target satellite state estimation data processing chains with respect to the target spacecraft. This task would involve obtaining a dynamic model of the target spacecraft using LIDAR complemented by stereo/optical cameras, detecting and classifying designated landmarks for grasping and stabilization. Relative localization of the target satellite or a designated target point provides inputs for planning trajectories of the robot manipulator.

The final phase of operation requires vision-aiding markers or a designated target structure for accurate pose estimation under simulated in-orbit illumination conditions. The localization chain can be switched to a slower rate but higher accuracy of output to enable a close and safe approach to the target. Once the two spacecrafts are close enough (< 1m), the manipulator onboard the servicer spacecraft will be commanded most likely in open loop wrt the CDFF software. The quantitative evaluation of the CDFF will be primarily offline on a recorded dataset obtained from the on-orbit servicing simulator to allow multiple parameterizations to be tested under the same conditions in reasonable time, and to minimize the effect of possible synchronization issues that may be insufficiently addressed by the hardware setup. However, the CDFF will be demonstrated for qualitative analysis as integrated on the OG6 platform. The data fusion processing chains will be evaluated based on the localization precision w.r.t to ground truth measurements.

2.2.4 Associated data products

Which state information is estimated depends on the range to the target and the same applies for the possible and required accuracy.

At far range, the state contains bearing only information, i.e. the position of the target relative to the servicer.

At mid- and close range the state of the target satellite consists of its position and orientation, i.e. the 6 DOF pose, relative to the servicer. By detecting the pose over a sequence of inputs it might be possible to determine the relative speed between the target and the servicer spacecraft. State estimation over periods of time assumes that no external forces are applied to the target (constant speed model). It is possible to determine whether sudden changes have occurred (e.g. a collision) by comparing the estimated state with the sensed state at each step and determining a residual.

The data products associated to this scenario are the following:

- Target satellite detection (far-range range and bearing localization);
- Target satellite state estimation (full localisation and speeds wrt. the chaser);
    - For the mid-range rendezvous;
    - For the close range approach;
- 3D reconstruction of the target.

2.3 Synthesis: list of InFuse Data products

The set of data products generated by InFuse and delivered to OG2-ERGO is split along the two planetary and orbital reference implementations.

The products delivered in the planetary track are:

- Digital Elevation Maps (DTMs):
    - Rover map;
    - Fused Rover Map;
    - Total Rover Map;
- Scientific target localisation;
- Localization;
    - At 10Hz while driving;
    - Absolute localization;
        - With respect to the orbital map (case of the long traverse scenario)
        - With respect to a localization map formerly built by the robot (case of the getting back scenario)
    - Relative localisation with respect to a man-made object.

The products delivered in the orbital track are:

- Target satellite detection (far-range range and bearing localization)
- Target satellite state estimation (relative localisation and speeds wrt. the chaser)
    - For the mid-range rendezvous
    - For the close-range approach
- 3D reconstruction of the target

Notes:

- This list associates a name for each of the data products, that are used throughout the rest of the document - yet these names may not match the ones defined in the ICD document, the final product naming convention will be adopted when this ICD document is finalized.
- Besides these data products exported to OG2-ERGO, various data products are built and maintained *within* InFuse for its own purposes (*e.g.* the Localisation Map for the getting back scenario). These private products, which are managed by the Data Product Manager, are introduced in Section 3 "Baseline solutions", along with the processes that generate and/or exploits them.

3 Architecture Design Rationale

3.1 CDFF objectives

The main objective of the CDFF is to provide a framework for the development, evaluation and deployment of space robotics data fusion technologies in the context of the PERASPERA cluster of project.

For the case of the relation between InFuse and ESROCOS this means that any data fusion process developed using InFuse should be portable to ESROCOS with minimal efforts. This also means that within InFuse it would be redundant to implement another RCOS or aim for taking decisions that are within the realm of the RCOS (e.g. processes priorities). Similarly, InFuse does not take decisions on the sequencing and control of its processes, as these pertain to the ERGO autonomy framework. Yet, InFuse has mechanisms to select and parametrize its processes on the basis of requirements expressed by ERGO. Finally regarding our relationship with the sensors system (OG4) InFuse does not pursue to control the internal state of the sensors but rather knows a set of operational modes that can be accessed and which provide data streams with different characteristics.

3.1.1 Data Products Management and Orchestration

InFuse has the focus set on the generation and management of a consistent and coherent environment representation which can be accessed externally. In this direction the *Data Products Management* stores in a consistent manner the environment representation data products and the *Orchestrator* analyzes which of the available sensor's operational modes and data fusion processes are the most adequate to successfully process data product requests.

3.1.2 RCOS Independence

InFuse provides a set of libraries and development tools to support the work done by data fusion solutions developers. InFuse aims for RCOS and robotic platform independence but pursues an easy transition to the target RCOS in various ways (e.g. Compatible interfaces, provision of deployment recommendations and middleware facilitators for partially automating deployment in a target RCOS). The decision of implementing the current set of utilities and not a complete inter-process communication and logging mechanism was taken after analyzing what could be useful for robotics perception researchers, the context of OG3 - responsibilities with respect to OG1 - and the amount of available resources.

Researchers on perception often have the goal of analyzing how a given algorithm or library performs on given logged data, most often using multiple specific parameterizations. For such analyses, the researcher is normally not interested on the details of the platform it could run on, or the target RCOS due to the inherent independence of the algorithm or library itself. Currently, one of the problems a researcher is facing when evaluating algorithms is the requirement of installing and running a RCOS (which might not be stable), taking the developer into long periods of working out of his realm of interest and expertise. *InFuse aims to facilitate the data fusion expert's work by providing a framework to evaluate data fusion solutions with little effort in a simple as possible environment.*

The transition between the developer's environment and the target system raises challenges which are platform and RCOS specific. Thus, this is out of the scope of the project as we aim for RCOS independence. It is arguable that it is possible to ease even more the work in this direction by providing a more realistic communication system between data fusion nodes so as to bring it closer to the final RCOS setup but this would imply the simulation of multiple RCOS communication mechanisms in InFuse, and furthermore the final system's communication will still require testing. It is also possible to think of solutions which could be directly deployed in target systems without any transition, but again these are RCOS specific solutions which, given the stage of OG1 development, does not seem realistic (e.g. some deployment steps in TASTE have to be done currently via GUI).

3.2 Architecture Design Process

The process of design has followed these stages: First, the requirements of the framework in its designer environment as well as for the deployed data fusion solution were collected. Second, an initial components diagram of the overall architecture was designed. Third, the design was further detailed by designing the sequence diagrams of some of the identified use cases.

In the second iteration of the architecture design further insights on the implementation of the use cases are done by focusing on the design and the deployment of a Data Fusion Processing Compound for a scenario of long term navigation.

The process of designing the architecture started with collection of the requirements which was started in the previous work package (refer to D2.2). In parallel many concepts were discussed to achieve a common understanding among the members of the consortium. As result of the discussions among other documents, a glossary incorporating all the important concept to understand the architecture designed has been produced.

Once the requirements were identified, the initial phase of architecture design produced a sketch of the system architecture in a components diagram. This represents software modules in an organized manner so that each module has a particular identity defined by the features it provides to the overall system.

The initial architecture is a sketch and a more in depth insight to how the requirements are to be provided must be done. For this purpose, use cases which require multiple functionalities from different components were selected, designed and analyzed with the partners. This process leads to a deeper understanding of what to do and how to approach the software implementation.

In order to further validate the approach and the adequacy of third party tools, critical use cases have been selected and prototyped to verify the validity of the proposed architecture. This approach aimed to identify at early stages of the development the real efforts to implement the architecture features, and to find the most appropriate way to implement them before completely defining the architecture.

Another important part of the design of the architecture is the definition of the interfaces with other elements of the final system. In the case of InFuse these are the autonomy framework (OG2), the sensors suite (OG4) and the RCOS (OG1). In this context various meetings were held with the corresponding OG representatives. As result of these meetings, in the first iteration of the architecture design phase an initial ICD was defined. With respect to the integration with OG1, the interfacing is of a different nature as the other ones. In this case, the interface defines how any solution defined by the Common Data Fusion Framework can be ported to the target RCOS. Discussions with the leading partner from OG1 helped to understand the data types that will be supported in the communication between processes of the target Robotic Framework.

4 Proposed architecture

The proposed process of developing data fusion solution solutions consist of two steps: (1) development and initial evaluation on the developer's environment and (2) deployment and testing on the target system. These two steps can be reiterated: after evaluating the initial solution on the target system, the logs generated by the target system's RCOS logging module can be used in the developer's environment to analyze and improve the solution.

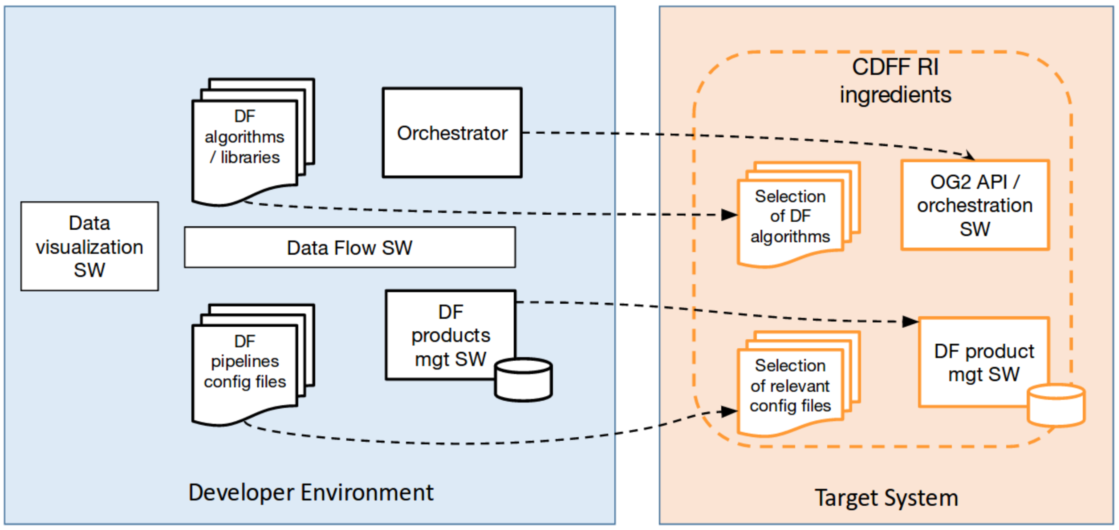

DF algorithms / libraries
Orchestrator
Data visualization SW
Data Flow SW
DF pipelines config files
DF products mgt SW
Developer Environment
CDFF RI ingredients
Selection of DF algorithms
OG2 API / orchestration SW
Selection of relevant config files
DF product mgt SW
Target System

Figure 1: InFuse approach towards the development and deployment of data fusion solutions. On the developer's environment a set of utilities for design and evaluation are available. The developed perception solution is design to be easy to integrate in any target system, independently of the RCOS and the specific hardware.

An important distinction must be done between features implemented by components that belong to the Developer's Environment uniquely (CDFF-Dev) and components that belong to both the Developer's Environment and the Target System (CDFF-Core and CDFF-Support). In Table 1 a comprehensive list of the different component and the set to which they belong is provided.

Feature	CDFF-Core	CDFF-Support	CDFF-Dev	On Target System	On Developer Environment
Core Libraries	X			X	X
Common Interface (C++ Level)	X			X	X
Data Fusion Processing Compound		X		X	X

Orchestrator		X		X	X
Data Product Management Tool		X		X	X
Common Interface (Python Level)			X		X
Middleware Facilitator			X		X
Logs and Data Flow Management			X		X
Visualizer			X		X
Data Analysis and Performance Tools			X		X

Table 1: List of features implemented in the CDFF. CDFF-Core and CDFF-Support incorporate all features that are included both in the target system and in the developer's environment. CDFF-Dev includes features which are useful for the developer to design a data fusion processing solution as well as to evaluate existing ones using logged data. None of the features of CDFF-Dev will be deployed in the target system.

InFuse is not conceived to be an RCOS: some features that an RCOS deal with are left aside: realtime communication layer, hardware interaction, intelligent processes deployment... In our view these are features that the RCOS should provide and eventually will aim at providing, as it is the only subsystem with the complete picture of the system running components. Nevertheless, the developer can provide helpful information in the Configuration Files to facilitate a smart deployment while the DFN Common Interface (DFNCI) is conceived to ease the integration process in different RCOS.

4.1 CDFF-Core

The CDFF-Core provides the "core" data fusion set of state of the art algorithms and techniques currently in use and applicable to the Planetary and Orbital RIs within the scope of Infuse, as a collection of ready-to-use libraries or packages. These libraries represent the CDFF's processing methods necessary to fuse sensory data. The CDFF-Core is to be implemented in a modular fashion with a common interface to allow high flexibility in configuration as well as in operation as a distributed system on multiple platforms. The core libraries are to be deployed on the ***target system*** (robotic platform) as well as on the ***designer's environment***.

Examples of core libraries include low-level functions such as feature detection, registration and recognition, data association, state estimation, outlier removal, and filtering as well as building environmental representations, achieving 3D object reconstruction or SLAM…

4.2 CDFF-Support

The CDFF-Support provides the necessary tools for the instantiation and execution of Data Fusion Processing Compounds. Under the CDFF-Support functionalities can be found the Data Fusion Processing Compound (DFPC), the Orchestrator, and the Data Product Management Tool (DPM).

CDFF-Core entities will be deployed with no change into the final target system. All components of CDFF-Support will be deployed in the Target System and are also available for programming and testing in the Developer Environment.

1. The **Data Fusion Processing Compound** (DFPC) is a combination of several nodes designed to provide a certain data product (e.g. pose estimation, map...). The DFPC defines the connections between input and output ports of different data fusion nodes.
2. The **Orchestrator** is the component that deals with the activation and deactivation of DFPCs and configures the data connections between the nodes. It is also the component that receives and answers the requests from the Autonomy Module (OG2). The selection of one or another DPCH is done based on availability and quality of data sources and on the data product required.
3. The **Data Product Management** (DPM) is a tool which acts as a long term memory for the data fusion products being generated by DFPCs to be used by OG2 or other DFPCs.

4.3 CDFF-Dev

The CDFF-Dev provides the tools to develop, test, visualize, and perform analysis on data fusion products. The CDFF-Dev include tools such as a DFPC inspector, data log replay, visualization tools, and data analysis tools. None of the components of the CDFF-Dev are deployed on the target system.

1. The CDFF-Dev **Common Interface (Python level)** are python bindings provided for the DFN common interface. This provides the developer the possibility to evaluate the data processing solution (or library) directly from Python without the need to port the solution to a specific RCOS.
2. The **Middleware Facilitator** provides the CDFF the capability to partially convert a DFPC from the designer's environment in the corresponding DFPC on the target RCOS. This utility is foreseen to provide a nominal level of corresponding target RCOS specific wrapper code for the CDFF-Core and CDFF-Support components.
3. The **Logs and Data Flow Management** is a tool that allows to inject logged data to a data fusion process in a chronological fashion. It is envisioned to allow loading of log data produced by different RCOS[1]. It replaces the communication layer of the RCOS but as this is not a tool that will be deployed in the target system, there are no requirements regarding real time communication. Nevertheless, the data will be timestamped so that a realistic estimation of the processing time required by the DFNs is available to the designer. The outputs of the nodes in the DFPC are also stored in logs which are used as input for posterior nodes in the processing compound.
4. The **Data Analysis and Performance Tools** are comprised of statistical analysis tools and graphical representations necessary for comparison of data fusion products resulting from different processing chains. The goal is to provide the designer methods to assess the quality of the data products and of the DFPC. The logs of the internally generated data are for example useful to compare data products generated using different DFPCs.
5. The **Visualizer** is responsible for presenting graphical representations of the different data products (e.g. 2D/3D plots, maps, camera images...).

Note that during the design of the architecture other features were identified with potential interest, which could be implemented in the future. For instance, the **Data Fusion Processing Compound**

[1] To achieve this, specific converters for each RCOS to the CDFF types have to be implemented. In the course of the project, converters to ROCK will optionally be implemented.

Configurator is a tool that enables the designer to define data fusion processing compounds (i.e. DFNs used, order, connections, frequencies, priorities etc.). The result of using this tools is a DFPC Configuration File. This configuration file is then used by the Middleware Facilitator to generate the correspondent DFPC for the target RCOS. The DFPC Configuration File is also required to design the operation logic of the Orchestrator which might handle multiple DFPCs. CDFF-Dev could provide in the future monitoring of the execution time and memory consumption of individual components in order facilitate the design of an embedded DFPC.

5 Detailed description

In the following sections a more detailed review of the already presented components is presented.

5.1 CDFF-Core

5.1.1 Data Fusion Node Common Interface

The Data Fusion Node Common Interface (DFNCI) provides the abstraction layer to encapsulate each data fusion node internal processing. The DFNCI is designed to be general enough to standardize all cases of data fusion processing nodes, but on the other hand, it is also specific for the context of perception and pose estimation which are at the core of data fusion in robotics.

In InFuse, the aim is to provide a RCOS independent framework. Thus, it is aimed to leave as much as possible RCOS dependent features out of the DFNCI. On the other hand, it has been kept in mind that any final perception solution developed and evaluated will eventually be deployed in a target RCOS. The DFNCI eases the RCOS integration process by capturing information regarding the DFN libraries and provides a standardized interface to harmonize different libs for future porting to target middleware and test using different programming languages and test utilities (e.g. Python Pandas).

At the moment, RCOS components tend to have a dependency on the framework in which they were initially developed. Furthermore, a component performing the same task might be available in more than one RCOS, with different interfaces, and implemented with an ad-hoc glue code and logic. Our approach to achieve framework independence is to move the logic from the RCOS interface to the library level and to provide the generation of the glue code as much as possible automatically and supplemented by the developer.

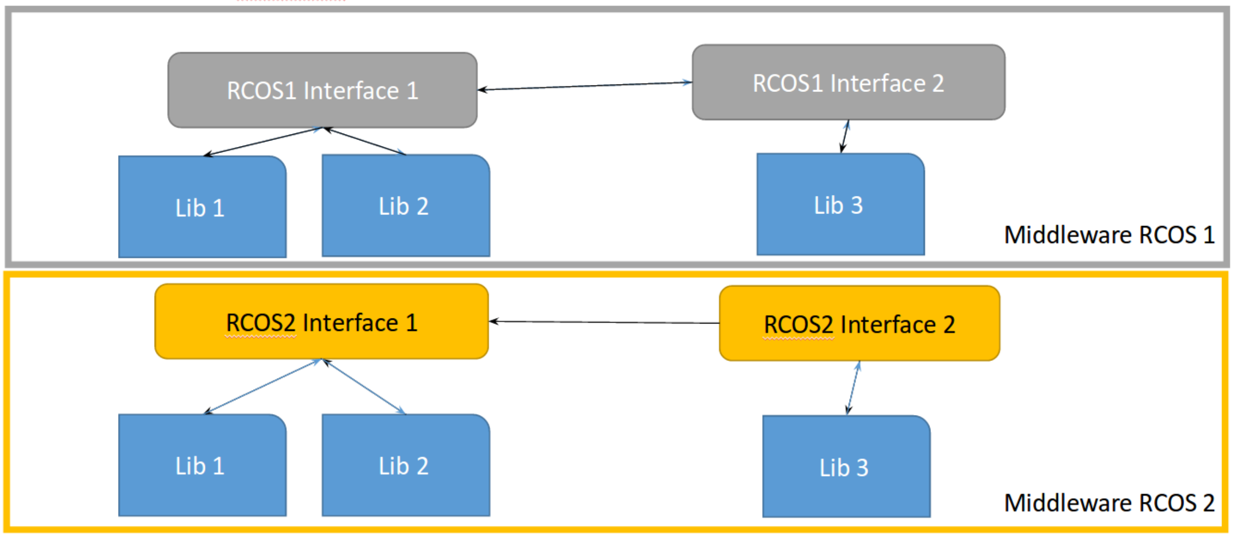


Figure 2: Currently Components performing a same task and using the same libraries might have implemented at the RCOS dependent level many logic.

For CDFF-Core, current RCOS-specific implementations of data fusion nodes will be replaced by an RCOS-independent implementation that contains the component logic. The component logic will expose a common interface that can easily be wrapped with code for the target RCOS (e.g. for type conversions in any RCOS). The component logic will also contain additional glue code for inter-library method invocations, data sharing etc.

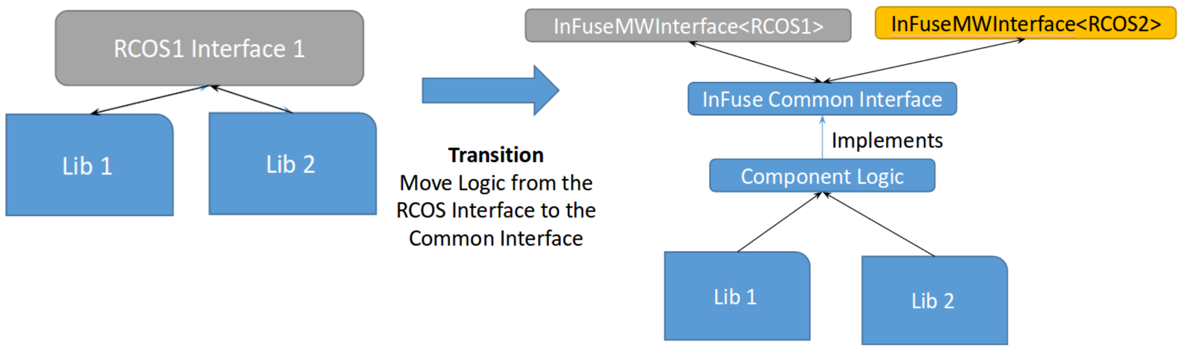


Figure 3: Proposed approach towards RCOS independent Data Fusion Processing Nodes: Component logic should be moved to the library level and a common interface (DFNCI) will facilitate the integration into different RCOSs.

The DFNCI includes:

- Methods to receive and output data
- Methods for configuration of internal DFN parameters (optional)
- Methods for handling of Meta-Information of the processed data (optional). We consider these three as the minimally required metadata fields:
    - Spatiotemporal information (transformation and timestamp)
    - Source information (sensor + parameters)
    - Processing steps gone through
- Allows to capture information regarding processing requirements.
- Allows to capture information regarding timing requirements (e.g. periodicity, synchronicity)
- Provides optional interfaces for control of the internal states by the orchestration

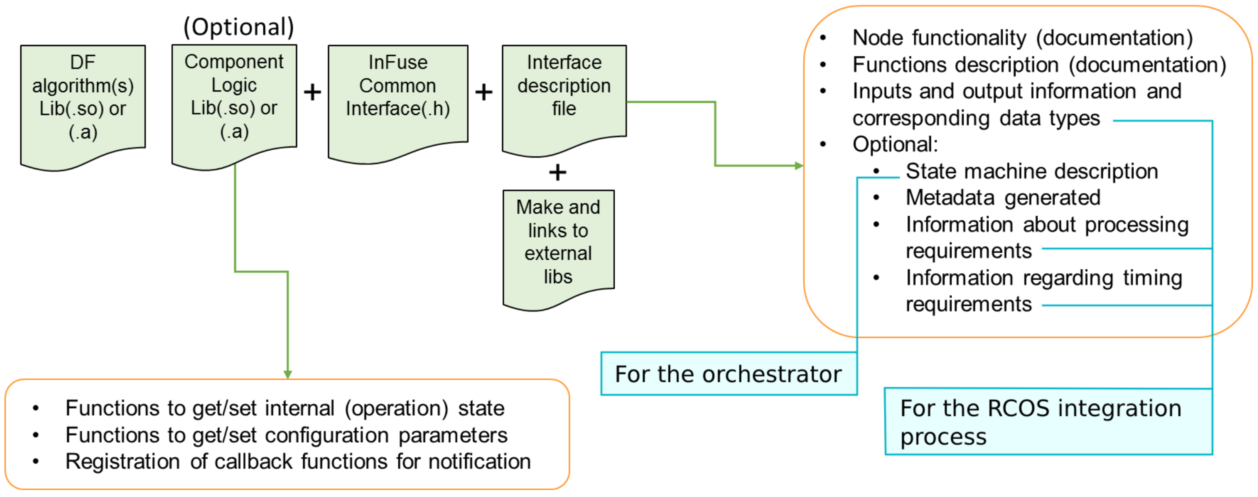

(Optional)
DF algorithm(s) Lib(.so) or (.a)
Component Logic Lib(.so) or (.a)
+
InFuse Common Interface(.h)
+
Interface description file
+
Make and links to external libs
Node functionality (documentation)
Functions description (documentation)
Inputs and output information and corresponding data types
Optional:
State machine description
Metadata generated
Information about processing requirements
Information regarding timing requirements
For the orchestrator
For the RCOS integration process
Functions to get/set internal (operation) state
Functions to get/set configuration parameters
Registration of callback functions for notification

Figure 4: Artifacts of a Data Fusion Node Common Interface (DFNCI). Information in the Interface Description will ease the process of integration in the target RCOS as well as in orchestrated DFPCs.

5.1.1.1 Metadata Handling

The three main information that the metadata provides along with the data products are: (1) *what:* type of the data product and what it refers to - this comes along with the necessary information to process or fuse the data, *e.g.* calibration parameters, (2) *when:* date of data used have been acquired, and (3) *how:* the DFN and DFPC that produced it. The first two types (*what* and *when*) of metadata are mandatory for the DFNs and the DPM to properly handle the data, while the third type (*how*) would rather be needed for execution control (orchestration) purposes, and certainly goes beyond the requirements of the CDFF architecture. Note also that the metadata could include more information depending on the application scenario, their definition and representation being to be defined by the final developer.

In InFuse we evaluate various potential approaches to manage the metadata and currently not a final decision has been taken to which one will finally be established. The following options are foreseen:

Option 1: Metadata in the data types

In this approach the metadata is already part of the types that the CDFF uses. The convenience of this approach lies in the ease with which the metadata can be manipulated in and accessed for every data product without the need for implementation of any additional metadata communication infrastructure. The main drawback that we see is that the incorporation of this metadata might not be well accepted by the maintainers of the types as it includes data that for other applications could be of no use.

Option 2: Compositions of data types

In this approach EnviRe could be used to implement the associations of data and metadata. EnviRe can associate multiple objects to a frame and in this frame store also structures defining relations of the objects in the frame. This solution is entire applicable internally in OG3 and would not imply any addition of data to the base types, but the use of this approach will be more complex when communicating this data through the middleware of OG1. Nevertheless the problem could be tackled.

Option 3: DFN for metadata association

In order to facilitate the integration of metadata along with the normal data products in a flexible and non resource consuming manner (i.e. minimum computations and communications), the generation and association of additional metadata (i.e. not included in the data type itself) could be done as an optional feature that any DFN can implement.

The metadata would be generated by data fusion nodes which will read streams of data and will generate -according to its internal implementation- the metadata associated to the data product or products. This approach facilitates the inclusion of metadata where the developer might want to add it, and also when it is relevant to generate (e.g. continuously, only when certain object is recognized…). Furthermore, this approach encapsulates additional handling from all other data management. This could be beneficial for instance for optimization. Figure 5 and Figure 6 present two data flow examples of how the MetaData Fusion Nodes can be set.

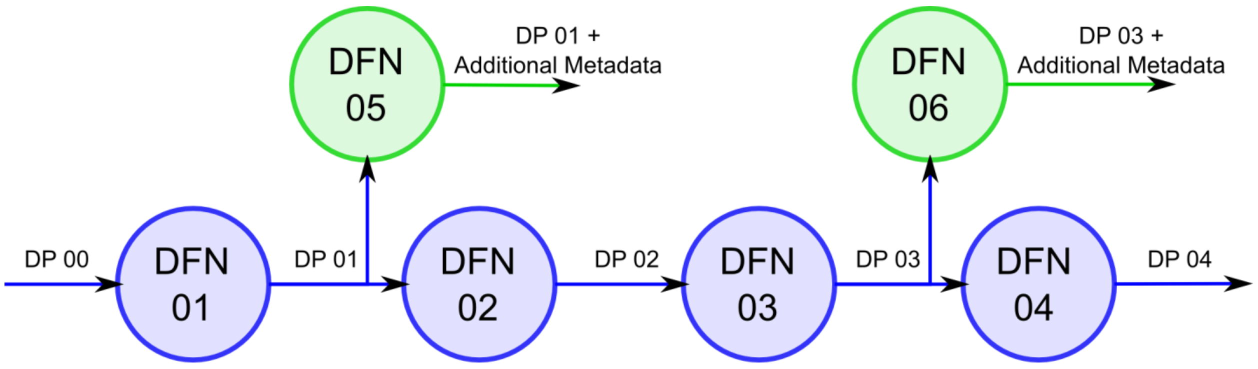


Figure 5: Data Fusion Nodes can be set to listen at the Data Products generated by other Data Fusion Nodes and associate to their output additional metadata. The Data Fusion Node that listens has implemented decision logic on when a data product should be incorporated with metadata. The output of the listener will contain the Data Product or a reference to it and relevant information about it (e.g. spatiotemporal, source, processing steps gone through).

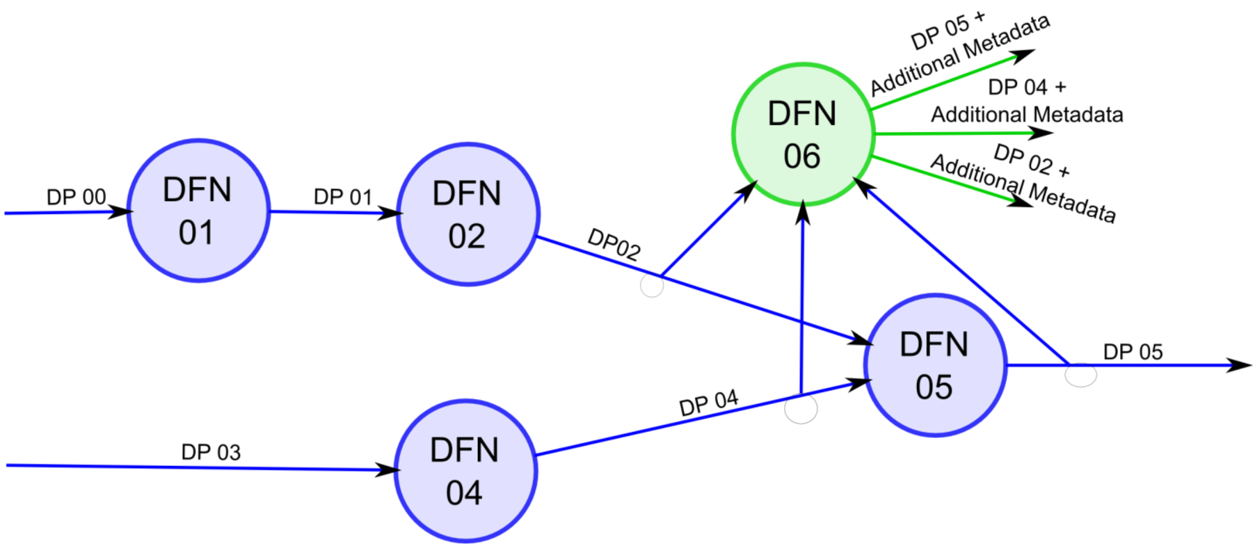

DP 05 + Additional Metadata
DP 04 + Additional Metadata
DP 02 + Additional Metadata
DFN 06
DP 00
DFN 01
DP 01
DFN 02
DP02
DFN 05
DP 05
DP 04
DP 03
DFN 04

Figure 6: Data Fusion Nodes can be set to monitor parts of the Data Fusion Process in different ways. In this case the Data Fusion Node 06 listens to three Data Product streams.

5.1.2 Nodes to be included

A large set of algorithms and technologies are available for inclusion in DFNs. To select the most appropriate and effective set of algorithms, test implementations will be created from existing code where available and from scratch where there is no available code. Test scenarios will create data for each type of algorithm as listed below. The most suitable methods for each application will be selected for C++ implementation as a data fusion node. In the ideal case, one or two algorithms will be included as the primary fusion in each category and others can be added optionally. Performance comparisons could be done in pySPACE in the Developers Environment -not directly on the target system- once actual outputs have been logged and assuming expected ones are also available.

The final list of DFNs to develop and integrate within InFuse will be defined after the precise analyses and trade-offs initiated in [RD5], which will be refined in WP5, tasks 5.1 and 5.2 (detailed specification of the orbital and planetary RIs). The following lists the DFNs that will most likely be integrated.

5.1.2.1 Feature Detection Nodes

The following feature detection methods will be evaluated for inclusion in nodes:

- Hough Transform
- Harris Detector
- ORB
- Optional: SIFT
- Optional: SURF

For three dimensional point clouds the following feature detection methods will be evaluated:

- Harris 3D
- SHOT (Signature of Histogram of Orientations)
- Optional: PFH/FPFH
- Optional: SURF 3D

5.1.2.2 Recognition and Registration Nodes

The following recognition and registration methods will be evaluated:

- ICP
- RANSAC
- Optional: Levenberg-Marquardt

In addition, the following general data association techniques will be evaluated:

- K-Nearest Neighbors
- Linear Classifier
- Bayesian Classifier
- Dense registration

5.1.2.3 State Estimation Methods

The following non-probabilistic state estimation methods will be evaluated:

- Optical Flow Estimation

- Fuzzy Logic
- Dempster-Shafer
- Optional: Dezert-Smarandache

The following probabilistic state estimation methods will be evaluated:

- Kalman Filter
- Unscented/Cubature Kalman Filter
- Optional: Particle Filter/SMC

5.1.2.4 Data Filtering and Outlier Removal Nodes

The following data filtering and preprocessing methods will be evaluated for inclusion in nodes:

- FFT High/Low/Band pass filters
- Variance filter
- Decimation
- Normalization

The following outlier removal methods will be evaluated:

- Interquartile range / Mahalanobis distance
- One-class support vector machine
- Optional: Gaussian Mixture Models
- Optional: k-means
- Optional: Minimum Volume Ellipsoid

5.2 CDFF-Support

The CDFF-Support consists of a set of components that will run on the target system. These components provide supporting tools to use multiple DFNs together and in a coordinated fashion. Furthermore CDFF-Support also provides the Data Products Manager which stores a consistent representation of the environment, a history of acquired pre-processed sensor data, estimated poses, and a selection of the generated fused data products, so as related to deliver them under request to OG2.

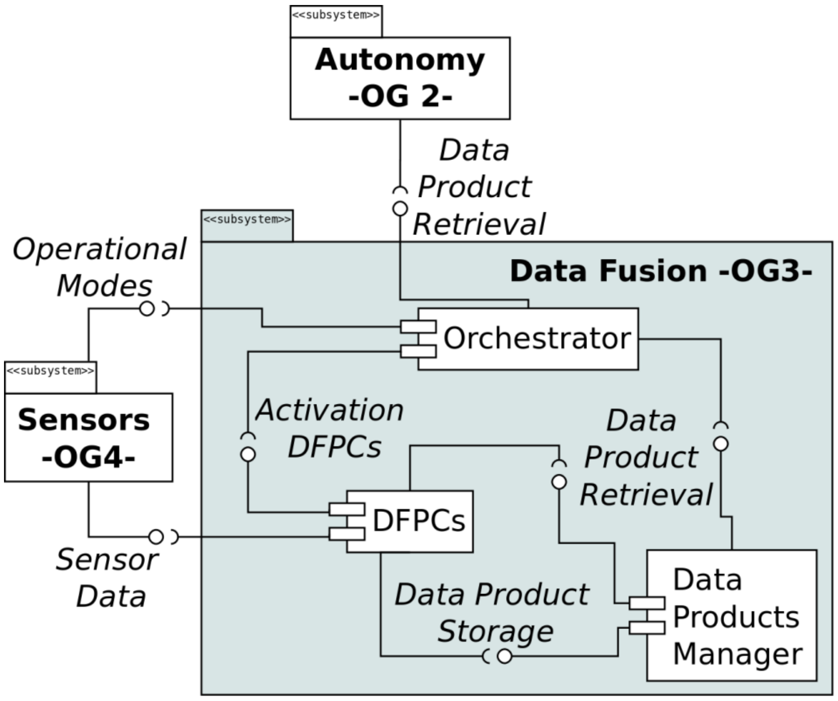


Figure 7: The diagram presents how the three components of the CDFF-Support interact and also the interfacing with OG2 and OG4. The *orchestrator* (1) performs request to the Data Products Manager based on what OG2 requires (2) sets the operational modes for the sensors and (3) activates or deactivates DFPCs. The *Data Product Manager* (1) stores under request the data products generated in the DFPCs and (2) retrieves under request data products for the Orchestrator or the DFPCs.

5.2.1 Data Fusion Processing Compound

The main feature of a Data Fusion Processing Compound is the Description File and the associated DFNs. It states the DFNs involved in the DFPC, their connections and the processing order. This defines the steps to produce a data product from a set of another data product(s).

In more details, the description file contains:

- Description of the data fusion functionality - intermediate processing steps, input data to the DFPC and fused data outputs of the DFPC.
- List of required DFNs and their corresponding libraries
- Configuration, periodicity and priority of each DFN in the DFPC
- Connections description between the DFNs
- Frames (static and dynamic transformations) applicable to each DFN
- Optional
    - Semantics
    - Deployment recommendation
    - Nominal frequency of operation
    - Recommended queues lengths
    - Build instructions

In the Data Fusion Processing Compound Description File, information about possible ways in which the DFNs can be distributed among threads in order to maximize the efficiency of the CDFF depending on the target system is optionally included. The format of the description file will offer the possibility to specify on which thread each DFN should be executed depending on the number of available threads. A format to describe this information will be proposed, in order to facilitate the deployment task to either a user, to a middleware facilitator or target RCOS that could automatically use this information to partially or fully deploy a DFPC in a desired RCOS.

In order to develop a DFPC to be integrated in a robotic platform, the model of the robot is needed: (1) The names of the frames that are associated to the sources of the data (sensors) and to the target frame on which the fused data should be given must match the frames of the robot model. (2) The robot model is needed to set up the transformation graph from which the transformations can be obtained and (3) the robot geometric model might be required for some DFNs (Ex. odometry).

5.2.1.1 Example of a data Fusion Processing Compound and the Description File

The following example was taken from an existing robotic software stack implemented in the RCOS Rock[2]. In the proposed example a lidar sensor is mounted on top of a tilting unit actuated via a dynamixel servo out, while moving up and down the lidar captures 2D scans of the environment. The DFPC performs the composition of the 2D scans taken with varying inclinations into a pointcloud. Furthermore a filter that removes parts of the robot from the 2D scans is included.

The corresponding DFPC describes a one layer lidar sensor (data source) connected to a lidar filter. The output of this last one is connected to a DFN node called Tilt Scan that produces the pointcloud. To produce this pointcloud the status -transformations for the tilt- of the dynamixel is passed to the Tilt Scan DFN.

[2] Rock RCOS website: http://rock-robotics.org/stable/

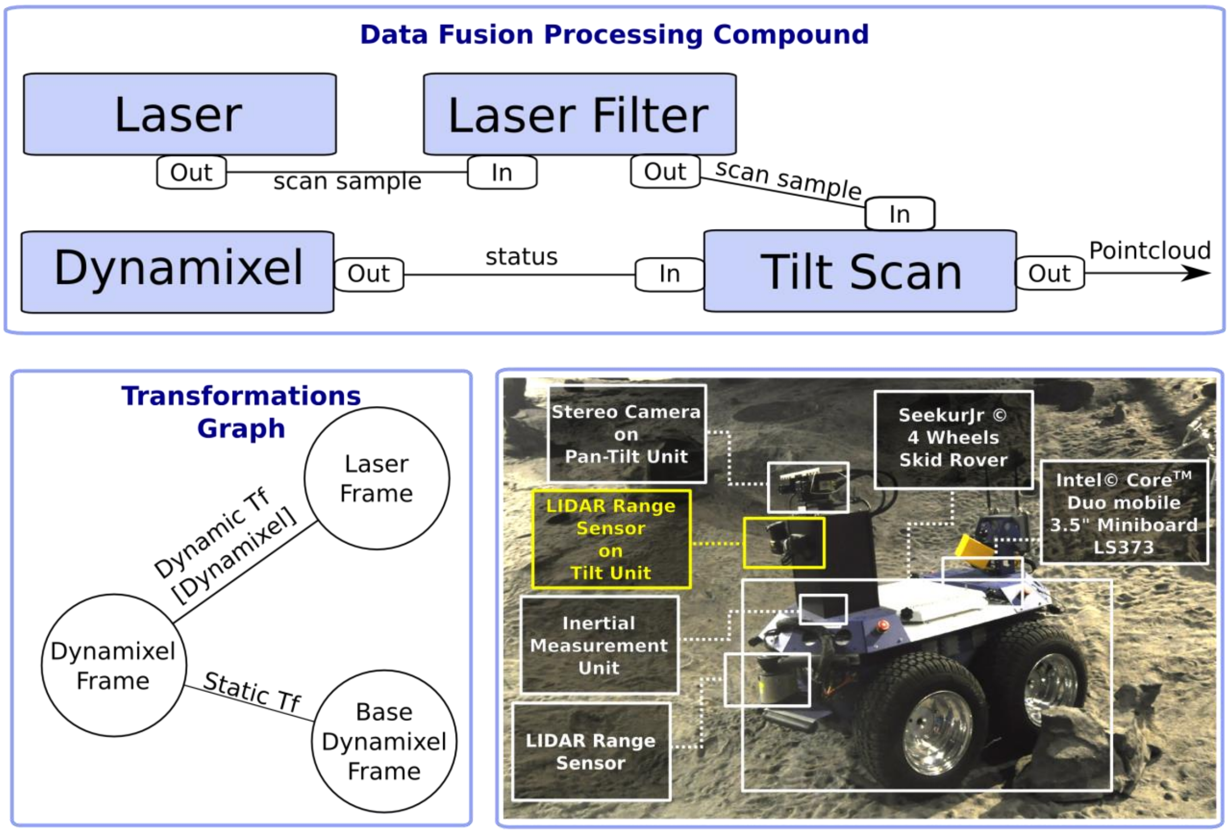


Figure 8: This figure describes an example of a DFPC, two data sources (laser and dynamixel) connect with two DFNs. The data product that is generated is pointcloud and the input data types are 2D scan samples and servo status (including motor position).

- Description of the functionality
  - Create pointclouds from of sensor data of a two dimensional lidar which is mounted on a servo and the status of the servo. The generated pointclouds can then be used for instance build DEMs which the autonomy framework will use for planning.
- List of required DFNs and libraries
  - Transformer, robot model and base-types
- Configuration, periodicity and priority of each DFN in the DFPC
  - Frame names:
    - Laser: Laser_frame <from robot model>
    - Dynamixel: Dynamixel_frame <from robot model>
    - Dynamixel Base: Dynamixel_base_frame <from robot model>
  - Periodicity:
    - Laser_Filter: scan_input_triggered
    - Tilt_Scan: scan_input_triggered
  - Priority:
    - Laser_Filter: 1
    - Tilt_Scan: 1
- Networks description
  - Inputs:
    - Laser: <2D scann_sample_type>
    - Dynamixel: <status_sample_type>

    - Outputs:
        - Pointcloud
    - Laser.out.scan_samples.connects_to(Laser_filter.in.scan_samples)
    - Laser_filter.out.scan_samples.connects_to(Tilt_Scan.in.scan_samples)
    - Dynamixel.out.status.connects_to(Tilt_Scan.in.joint_status)
- Frames (static and dynamic transformations)
    - Static:
        - Dynamixel_frame To Base_Dynamixel_Frame: <Path_to_robot_model>
    - Dynamic:
        - Laser To Dynamixel_frame: <Dynamixel.out.status>
- Optional
    - Semantics
    - Deployment recommendation
    - Nominal frequency, queues lengths
    - Build instructions

In this particular case the robot geometric model is needed to set the names of the source frame, the dynamixel frame and the target frame, and to set up the transformation graph with which the status of the dynamixel (dynamic transformations) will be feed.

5.2.2 Orchestrator

The orchestrator is an important functional component in the CDFF that has the main task of receiving queries from OG2 to activate certain DFPCs within OG3 and provide the fused data products to OG2 in the desired format. It acts as the central coordinator in the target system to control the activation states of DFPCs. The orchestrator has the following functions:

1. Interface between OG2-OG3
    a. Accepts requests from OG2 with a set of criteria to select an appropriate DFPC
    b. Provide a response about feasibility of the request
    c. Provide the fused data to OG2 as a response synchronously (preferred option by OG2)
    d. Notify OG2 that the fused data is available to be read i.e. asynchronously
2. Translate the perception and localization data into the format required by OG2 (or other DF product consumers) by accessing data products in the DPM.
3. Interface with OG4 Instrument Control Unit (ICU) to configure a limited set of sensor parameters (operational modes) influenced indirectly by OG2 requirement of data products with specific characteristics (resolution, range, data size, update rate etc.)
4. Interface with the Data Product Management (DPM) tool and provide mechanisms for querying fused data products
5. Activation and deactivation of DFPCs according to data product requests and operational modes of the sensors..

Optionally the orchestrator could provide additional functional capabilities:

1. Runtime monitoring of active DFPCs such as processing status, performance and memory constraints.
2. Functionality for selecting sub-components within a DFPC and initiating (or stopping) them for actual execution of the DFPC.
3. Provides a scripting (or user interface) mechanism for inserting or modifying DFPC descriptions that are stored in a repository

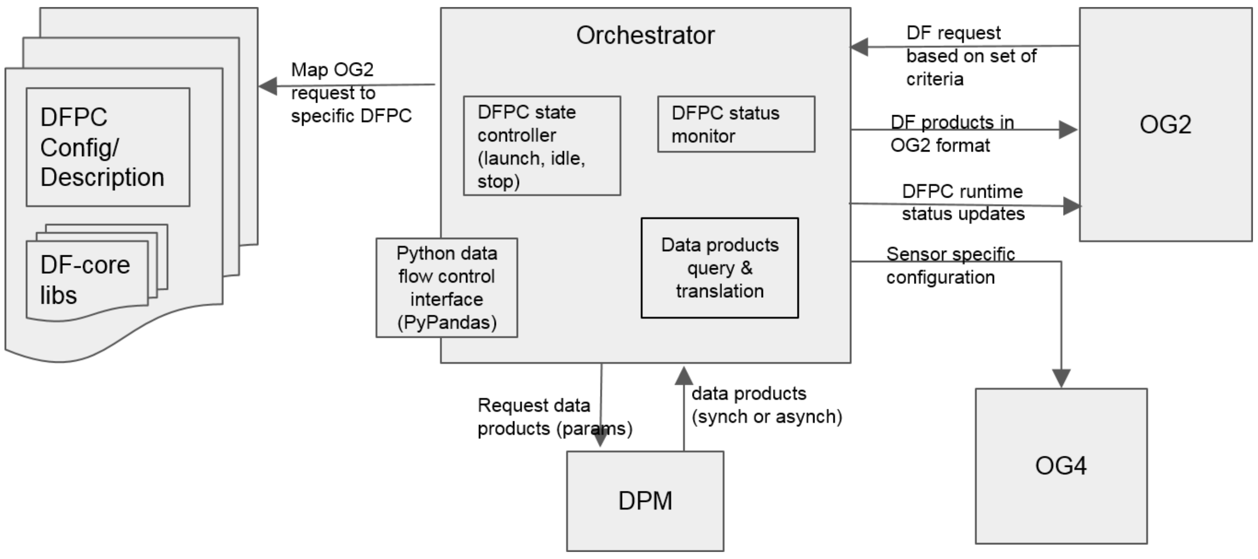

DFPC Config/ Description
DF-core libs
Map OG2 request to specific DFPC
Orchestrator
DFPC state controller (launch, idle, stop)
DFPC status monitor
Python data flow control interface (PyPandas)
Data products query & translation
DF request based on set of criteria
DF products in OG2 format
DFPC runtime status updates
Sensor specific configuration
OG2
OG4
Request data products (params)
data products (synch or asynch)
DPM

Figure 9: Internal software components of the Orchestrator and its interactions with other CDFF and external software modules

The diagram in Figure 9 illustrates the internal software components of the orchestrator and interactions with the DPM, DFPCs, OG2 and OG4. The DFPC state controller ensures that a selected DFPC is activated by loading the appropriate run time libraries with the associated executable. This includes the task of mapping an OG2 request to the most appropriate DFPC that can satisfy the OG2 criterion.

5.2.3 Data Products Management Tool

5.2.3.1 Motivations

Within the CDFF, a series of varied data and data products are processed and generated by a variety of DFNs. There is a need to store in a dedicated database a selection of these data structures, for different purposes such as:

- Serve OG2 requests for specific data products: for instance the Total Rover Map, which will certainly be requested at the end of a Long Traverse mission, requires the integration of (most of) the data gathered during the traverse, or of the various (Fused) Rover Maps that have been produced during the traverse.
- Serve the Operators, who may request specific data after the execution of a mission: for instance some data gathered in the vicinity of detected scientific target for further ground processing and analyses before commanding the rover to sample it.
- Serve internal CDFF purposes. Here various operations require the storage of data, either over the short or the long term, for instance:
    - To properly handle the asynchronicity of the data acquisition. Even if all the data are gathered in a synchronous manner, their processing require some time, and thus some data production may be delayed. For instance the precise position of the robot at the time of a 3D point cloud acquisition may be known only after a visual odometry process has refined the associated position.
    - By essence, SLAM solutions require the memorisation of environment related data (the maps of the M of SLAM, which are not necessarily products delivered to OG2). Depending on the selected SLAM implementation, such data can be landmarks, visual keyframes, point clouds keyframes, chunks of digital maps. etc. Most SLAM suites actually handle the management of such data internally, but there is an interest to expose them within a more principled spatial database management system.

- Prepare multi-robot cooperation schemes: similarly to the fact that environment related information are at the core of autonomous mobile robots, shared environment related information are at the core of the development of any multi-robot cooperation scheme. For instance a robot traversing an area already traversed by an other one may benefit from the information already gathered, be it for planning or localisation purposes: one robot's OG2 or OG3 sub-systems may require data stored by an other robots OG3 sub-system.

It is the role of the Data Product Management to memorise and manage the acquired data, the processed data and the generated data products that may be required for all these needs.

5.2.3.2 DPM implementation

Of course by no means all the data that is processed within the CDFF data can be stored: the DPM must manage the data stored so far, and be able to expose relevant data products to the various client's processes that require them.

The DPM can be seen a robotics-dedicated Geographic Information System (GIS). With respect to the activated DFNs and DFPCs in the CDFF, the DPM will process the data insertion requests. Internally, it manages all the spatial related data by implementing insertion, deletion or update functions, aiming at satisfying future needs for data products and storage constraints. With respect to the other systems and subsystems, the DPM acts as a server of data products for client processes, whatever they are (i.e. internal OG3-InFuse processes or OG2-Ergo processes).

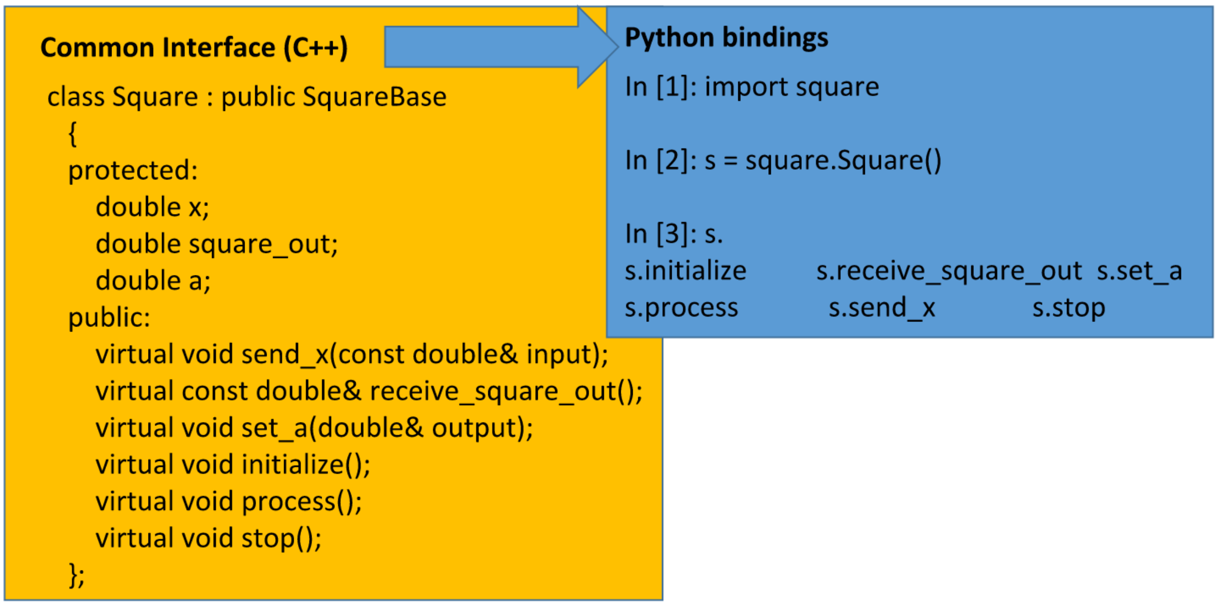
Internally the DPM will manage an EnviRe Graph in which multiple data products will be stored in a spatiotemporal consistent manner. Data Products might be serialized in this module to free memory usage. Given a request, the serialized data products shall be deserialized and deliver or the reference to a serialized object might be delivered instead. EnviRe currently provides serialization for various data types related to environment representation (e.g. maps) and it is extendable to enable serialization of further data types.

5.3 CDFF-Dev

Through the CDFF-Dev utilities, the developer will be able to: (1) connect DFNs and evaluate them with logged data, (2) define DFPCs and evaluate them with logged data, (3) test the orchestrator with logged data, (4) test the DPM tool with logged data, (5) run performance analysis to evaluate at least data filtering and outlier removal nodes and (6) visualize data communicated through DFNs using the Envire visualizer. Additionally, CDFF-Dev provides a Middleware Facilitator which eases the transition from the developer environment to the target system.

5.3.1 CDFF-Dev Common Interface

The DFNCI will provide Python bindings to allow prototyping of DFPCs or newly integrated DFNs. The bindings will be automatically generated, either based on the Node definition file or directly based on the C++ header of the DFN that implements the DFNCI. A tool to automatically generate wrappers for DFNs is provided in CDFF-Dev if the DFN only uses InFuse data types at its interface. The bindings will be based on Cython, which makes it easy to develop Python extensions that directly wrap existing C++ libraries. An example of a generated wrapper is shown in the next figure.



```
class Square : public SquareBase
  {
  protected:
     double x;
     double square_out;
     double a;
  public:
     virtual void send_x(const double& input);
     virtual const double& receive_square_out();
     virtual void set_a(double& output);
     virtual void initialize();
     virtual void process();
     virtual void stop();
  };
```

```
In [1]: import square

In [2]: s = square.Square()

In [3]: s.
s.initialize        s.receive_square_out  s.set_a
s.process           s.send_x              s.stop
```

Figure 10: An example of Python bindings generated for a DFN. A node that implements the DFNCI is displayed on the left side. On the right side, the auto completion from the IPython REPL is used to display the methods in Python.

5.3.2 Middleware Facilitator

The Middleware Facilitator component eases the process of transferring the data fusion solution from the developer's environment to the target system - RCOS and robot specific-.

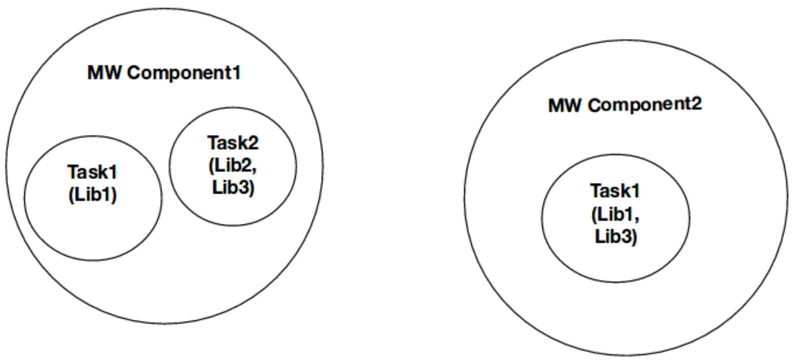


Figure 11: Terminology: MW Component - Library(s) + mw wrapper (system level process); Task - Threads running within the MW Component; DF-Lib - one or more DF algorithm implementations with appropriate interfaces

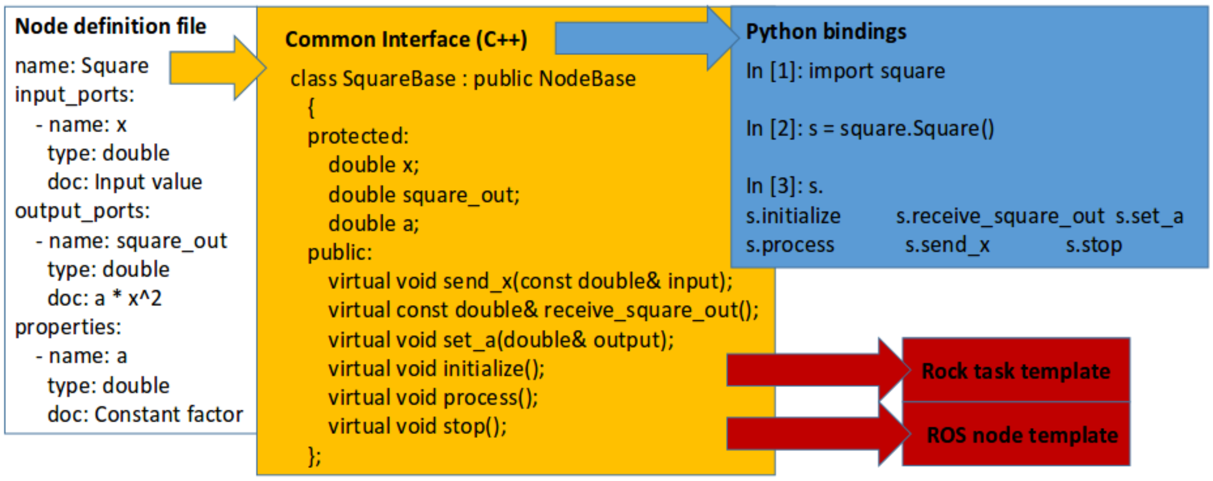
Figure 11 introduces some terminology that will ease the understanding of the following description. The middleware facilitator includes a code generator that will:

- Generate templates for the DFNCI
- Generate Python bindings for the DFNCI
- Generate, a template (e.g. for Rock tasks) wrapping the DFNCI, including
  - Port names and types

- Properties and types
- Type conversions

It will not contain temporal alignment of input ports, component frequencies, a description of the state machine or complex inheritance structures, and it will not generate the build configuration files because this is considered too complicated for a framework-agnostic code generation tool.

The Figure 12 gives an example for a very simple case. The developer writes a node definition file which defines inputs and outputs of the node, and configuration options. A base class that implements the DFNCI is generated automatically. The developer has to implement the logic that uses a RCOS-independent library to implement the DFNCI. From the DFNCI, Python bindings can be generated. In addition, templates for several RCOS like Rock or ROS could be generated. For the case of the RCOS TASTE the generation of such template might not be possible, because the classical design of modules involves the use of Graphical User Interfaces. The developer has to configure these templates, e.g. in order to align the inputs temporally, define the component frequencies, or configure the build process of the component. Libraries for type conversions between RCOS types and InFuse types must be available to facilitate the generation and development of the RCOS component.



```
name: Square
input_ports:
  - name: x
    type: double
    doc: Input value
output_ports:
  - name: square_out
    type: double
    doc: a * x^2
properties:
  - name: a
    type: double
    doc: Constant factor
```

```
class SquareBase : public NodeBase
 {
 protected:
   double x;
   double square_out;
   double a;
 public:
   virtual void send_x(const double& input);
   virtual const double& receive_square_out();
   virtual void set_a(double& output);
   virtual void initialize();
   virtual void process();
   virtual void stop();
 };
```

```
In [1]: import square

In [2]: s = square.Square()

In [3]: s.
s.initialize       s.receive_square_out  s.set_a
s.process           s.send_x           s.stop
```

Figure 12: The DFNCI template for a DFN will be generated from the node definition file. In addition, we can generate templates for RCOS components and a Python wrapper.

The Figure 13 presents a class diagram that shows the relations of the generated and manually implemented modules.

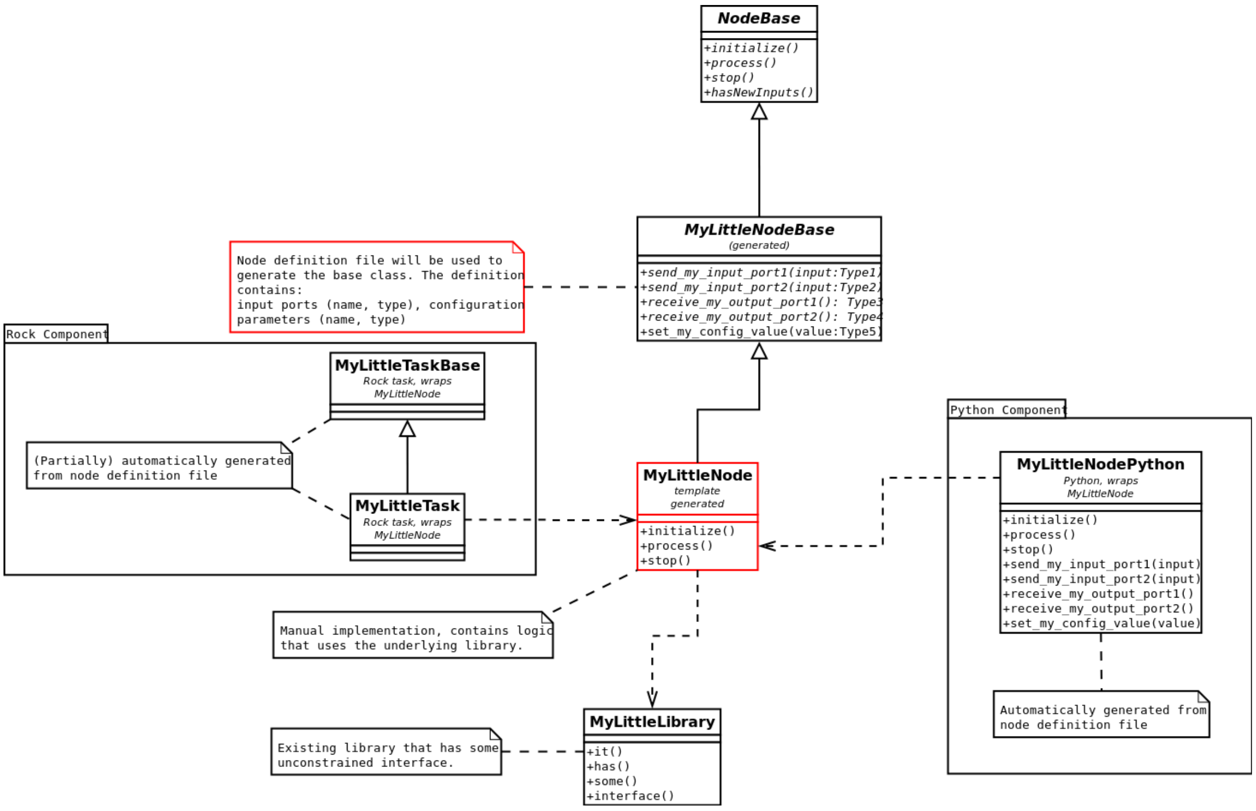

NodeBase
+initialize()
+process()
+stop()
+hasNewInputs()
MyLittleNodeBase
(generated)
+send_my_input_port1(input:Type1)
+send_my_input_port2(input:Type2)
+receive_my_output_port1(): Type3
+receive_my_output_port2(): Type4
+set_my_config_value(value:Type5)
Node definition file will be used to generate the base class. The definition contains: input ports (name, type), configuration parameters (name, type)
Rock Component
MyLittleTaskBase
Rock task, wraps MyLittleNode
(Partially) automatically generated from node definition file
MyLittleTask
Rock task, wraps MyLittleNode
MyLittleNode
template generated
+initialize()
+process()
+stop()
Python Component
MyLittleNodePython
Python, wraps MyLittleNode
+initialize()
+process()
+stop()
+send_my_input_port1(input)
+send_my_input_port2(input)
+receive_my_output_port1()
+receive_my_output_port2()
+set_my_config_value(value)
Automatically generated from node definition file
Manual implementation, contains logic that uses the underlying library.
MyLittleLibrary
+it()
+has()
+some()
+interface()
Existing library that has some unconstrained interface.

Figure 13: Class diagram of a node that implements the DFNCI. A Python binding and a template for a Rock component could be generated automatically. Red color indicates files that will be implemented by the developer:

5.3.3 Logs and Data Flow Management

The Logs and Data Flow Management (LDFM) module is a set of utilities that facilitates the Developer of DFPCs the connections between logs and core libraries (with the Data Fusion Node Common Interface) inputs and outputs. The LDFM belongs to the CDFF-Dev set of features and it is not deployed in the target system. The LDFM is not conceived as substitute of an RCOS but rather as an utility to evaluate - in the developer's environment - the adequacy of a DFN or a DFPC with logged data.

The Log Player is a tool that allows to replay a data fusion process in a chronological fashion following the timestamps of the data samples processed. The player should allow to play log data produced by different RCOS. To achieve this, specific converters for each framework to the intermediate log format have to be implemented. InFuse types must be serializable in the intermediate log format.

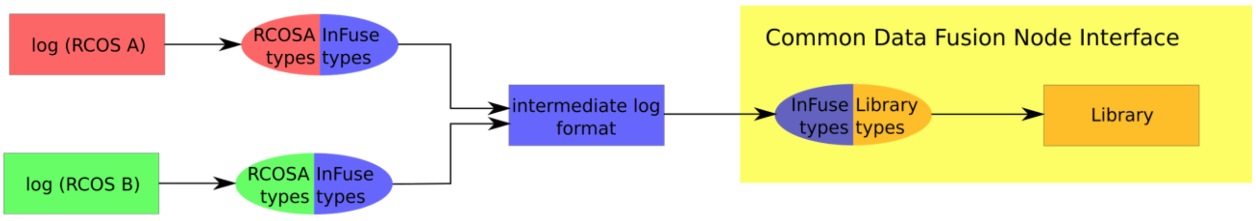


Figure 14: Conversions from RCOS-specific log formats to an intermediate log format that is independent of the RCOS have to be implemented. On the side of the DFNCI, conversions to the data types of the library have to be implemented.

The following subcomponents will be implemented:

- Data Flow Control: emulates the communication layer of an RCOS
- Log Player: allows to replay data fusion processes in a chronological fashion

The implementation of the data flow control is in a single process with one thread, everything is sequential. It will be lightweight; it should not be an RCOS replacement. It will have a simple API that can be directly controlled, e.g. from a Python REPL. An example of a sequence diagram in a simple scenario is shown in Figure 15.

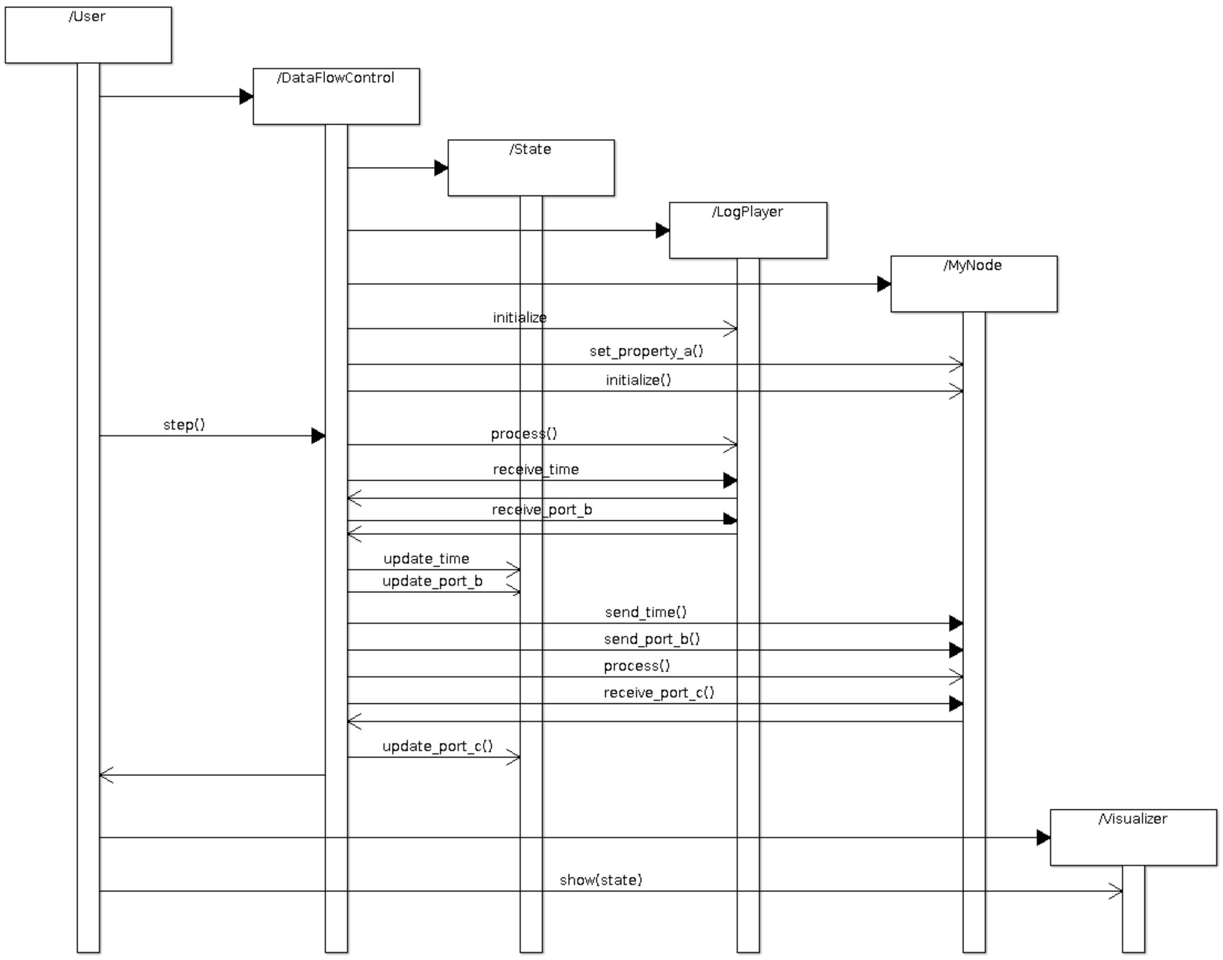


Figure 15: Sequence diagram that shows how a user could interact via REPL with the log player and the data flow control.

5.3.3.1 Python Bindings for EnviRe

Envire is a package for representing arbitrary information on the environment for robotics. The purpose is to have a common way of holding any information related to the environment of a robot and how the information relates to each other in a consistent representation. The applications of EnviRe varies between environment representation, navigation, planning and simulation. The model is a strongly connected directed graph which allows data acquisition, processing, and operations regarding autonomous systems demands.

In the context of InFuse, the EnviRe graph will be used as symbolic representation and structuring of the spatio-temporal information, as well as to facilitate the attribution of metadata. The visualization of EnviRe will be used to analyze data that is stored in the graph.

In order to use EnviRe as a tool in CDFF-Dev, Python bindings will be implemented as the graph and the visualization must become available in Python. The tool Cython will be used to wrap the C++ libraries directly. Data types that should be stored in EnviRe graphs must also be wrapped in Python. Because Python has a dynamic type system and C++ has a static type system, the exact template specialization for EnviRe have to be known during compilation of the Python extension, hence, it is not possible to write generic Python bindings for EnviRe that accept arbitrary Python types.

5.3.4 Data Analysis and Performance Tools

The Python Signal Processing And Classification Environment (pySPACE) will be used to analyze and compare different data fusion processing compounds (DFPCs). In the context of InFuse the main aspect of pySPACE is to be used for benchmarking purposes and for the integration of general filtering and outlier removal for sensor data.

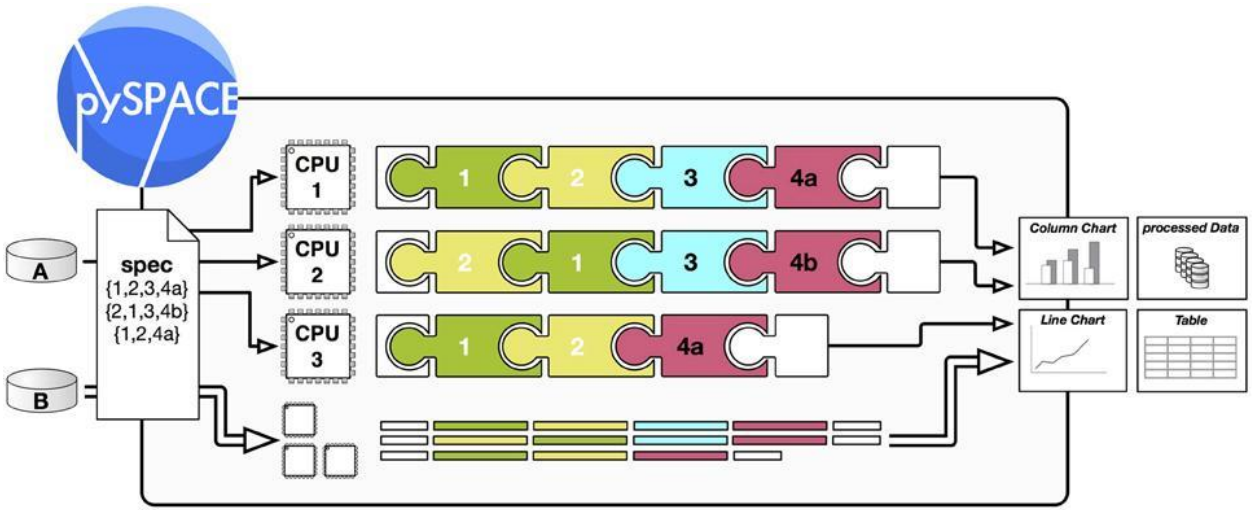


Figure 16: Visualization of pySPACE node chains. pySPACE can distribute the execution of independent node chains that will be compared over different processing units, e.g. CPUs.

PySPACE is a powerful tool that enables parallel process execution and offers more than 100 machine learning and signal processing algorithms that are implemented exclusively for pySPACE and wrappers for libraries like scikit-learn, Weka, and libsvm. It is easy to prototype and evaluate node chains that are not yet implemented for an RCOS.

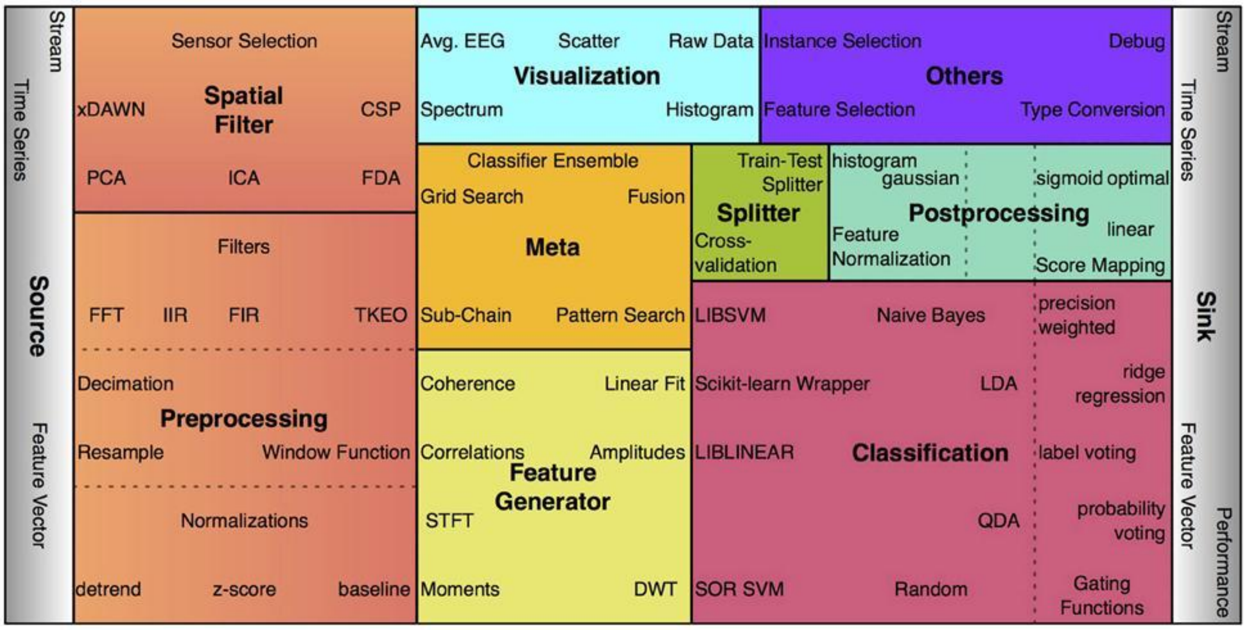


Figure 17: pySPACE provides a large number of nodes for preprocessing, feature generation, classification, visualization, etc.

PySPACE can be used to compare and analyse the performance (e.g., classification accuracy...) of several algorithms and parameter configurations easily in a graphical user interface. PySPACE also provides a 'launch_live' mode where you can directly execute signal processing as soon as you have the data in an online fashion. A multitude of metrics is already available for machine learning methods.

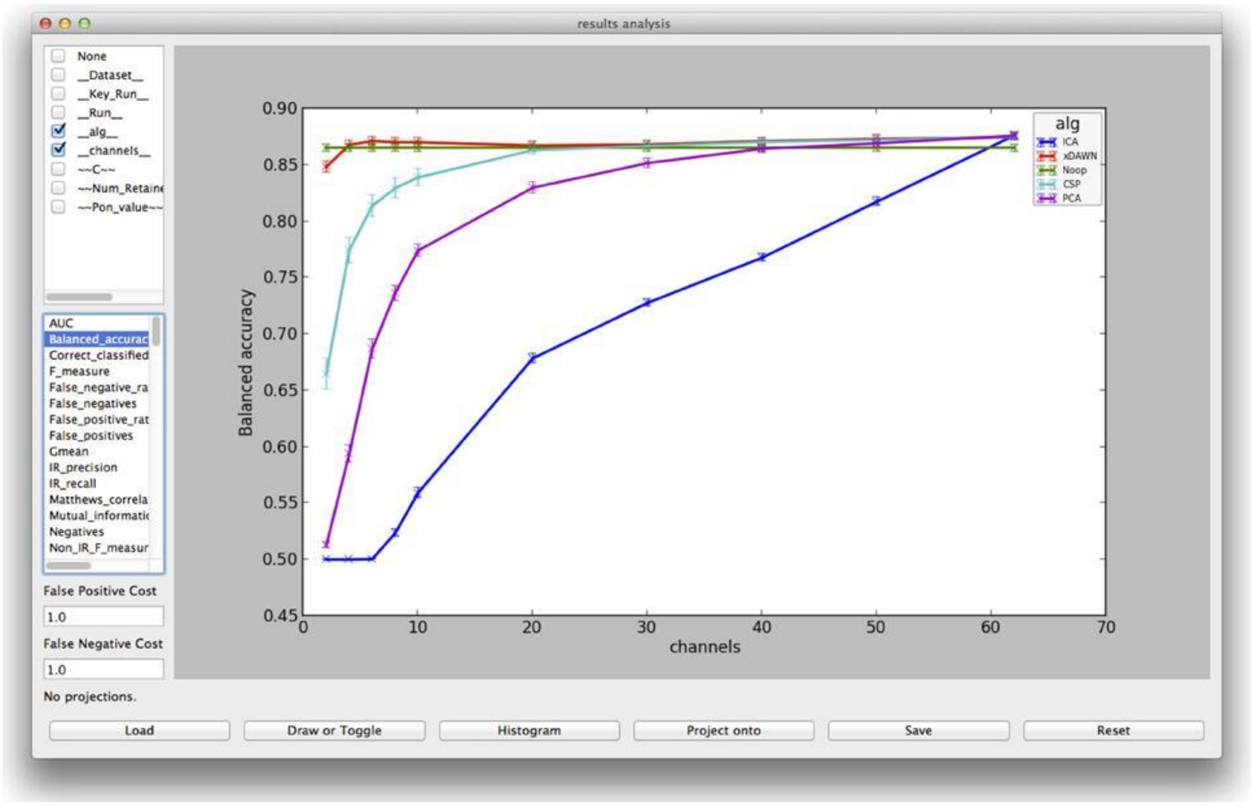
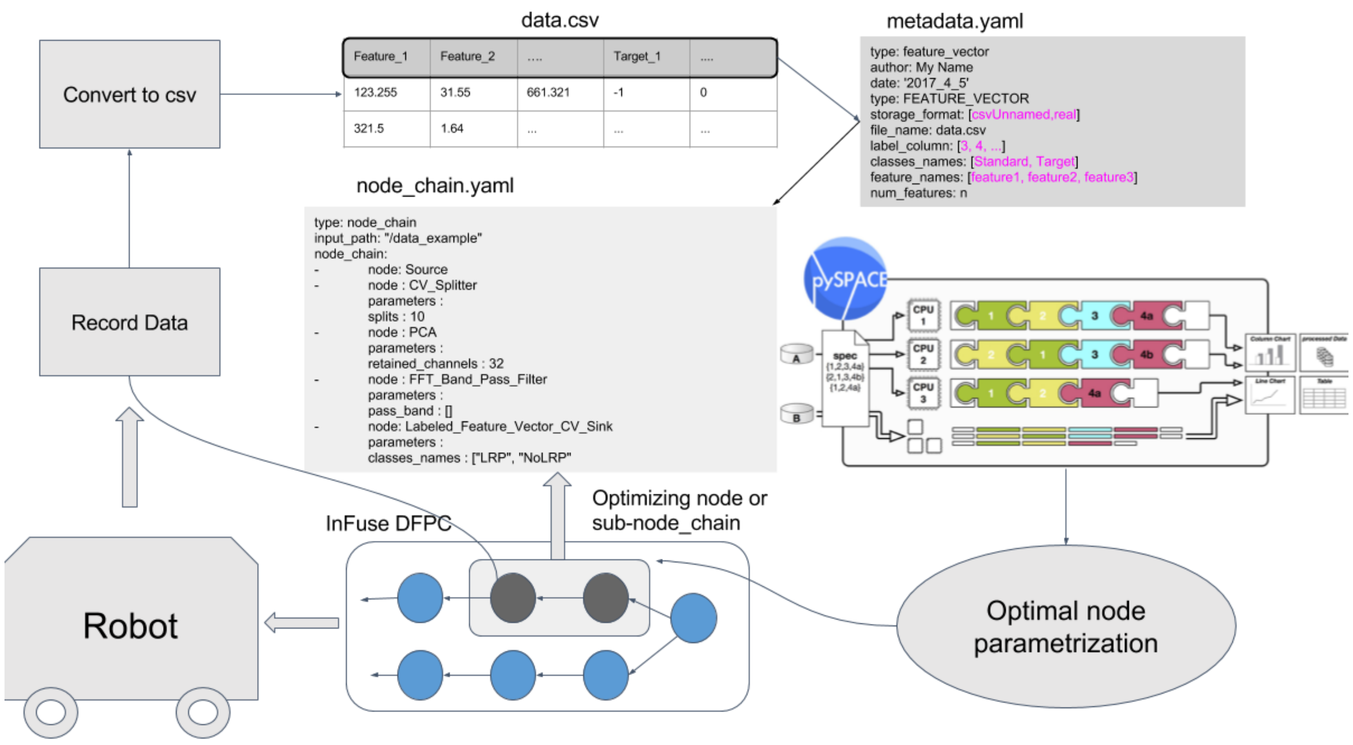


Figure 18: pySPACE offers various metrics and visualization tools to compare different machine learning and signal processing methods.

The general workflow for optimizing a DFN can be summarized as the following: Provided the data source to a specific node (which can be user specified or supplied by the previous node in the chain), a metadata.yaml file is setup which maps the header of the data being processed into a yaml file that is typically used in pySPACE. In the metadata file the user can choose between selecting all or part of the data features to be processed. Next a configuration file is setup, where the user can chain operation nodes selecting different algorithms as well as different parameter settings to be compared. It is also possible to compare the performance of full processing pipelines. After setting up the configuration file, the processing is then launched and the evaluation results are then synced together into a csv file. The results can be easily visualized using the performance analysis tool (as shown in the figure above). The parameter of the best performing node can then be exported to the corresponding node of the DFPC.



data.csv

Feature_1	Feature_2	....	Target_1	....
123.255	31.55	661.321	-1	0
321.5	1.64	...	...	...

metadata.yaml

```
type: feature_vector
author: My Name
date: '2017_4_5'
type: FEATURE_VECTOR
storage_format: [csvUnnamed,real]
file_name: data.csv
label_column: [3, 4, ...]
classes_names: [Standard, Target]
feature_names: [feature1, feature2, feature3]
num_features: n
```

node_chain.yaml

```
type: node_chain
input_path: "/data_example"
node_chain:
-       node: Source
-       node : CV_Splitter
        parameters :
        splits : 10
-       node : PCA
        parameters :
        retained_channels : 32
-       node : FFT_Band_Pass_Filter
        parameters :
        pass_band : []
-       node: Labeled_Feature_Vector_CV_Sink
        parameters :
        classes_names : ["LRP", "NoLRP"
```

Figure 19: pySPACE will be used as a tool to evaluate signal processing and machine learning nodes that will be deployed on the robot.

The workflow of developing machine learning and signal processing methods for DFPCs will include an initial evaluation of several methods that can be done in Python with pySPACE. Selected algorithms will be implemented in C++.

The implementation in C++ will have a constrained interface -DFNCI- which will be implemented by the developer of the node. For this interface, a Python wrapper can be generated automatically and a generic node for pySPACE is able wrap these nodes. The pySPACE node needs a serialization method, i.e. it automatically determines parameters of the model and stores them in a string. There is a distinction between transformation nodes (signal processing) and trainable nodes (machine learning). Method calls from the pySPACE node will be forwarded to C++.

Example: in practice, an algorithm can have an unconstrained implementation that is provided by some library (MyAlgorithmLibrary). The pySPACE interface (MyAlgorithmPySpace, C++) has to be implemented. For a transformation node, only the method execute(), which does the transformation, has to be implemented. For a machine learning node, the methods isTrainable(), isSupervised(), train(...), stopTraining(), and incTrain(...) must be implemented. Serialization and deserialization methods (e.g. fromYaml(...), toYaml()) are required. A tool to automatically generate the Python bindings (MyAlgorithmPySPACE, Python) will be available. For transformation nodes, the CppTransformationNode will be available in pySPACE, for machine learning nodes, the CppTrainableNode will be available.

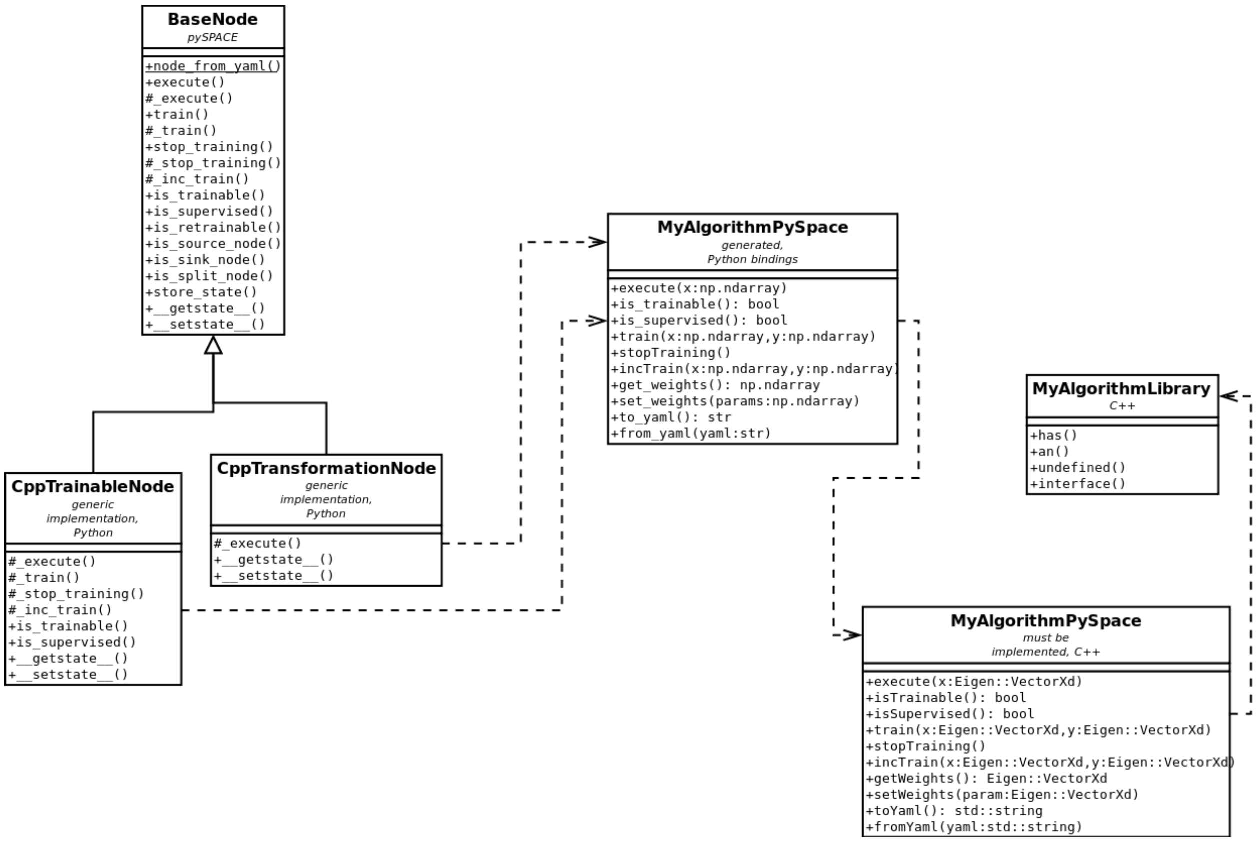


Figure 20: Class diagram that shows how a hypothetical algorithm *MyAlgorithm* can be integrated in pySPACE as a pySPACE node.

5.3.5 Visualization Tools

The EnviRe visualization tool will be used in the developer's environment to visualize the data products generated at different stages of the DFPCs. In order to visualize the data product Python bindings to the data type will be required as well as a visualization plugin for the type. Currently Envire Visualizer support Vizkit visualization plugins. There exist already Vizkit plugins for the base-types library[3] upon which the OG1 types are based, for map types compatible with those[4] and for other types commonly used in robotic applications.

[3] Base-types: https://github.com/rock-core/base-types

[4] Slam maps: https://github.com/envire/slam-maps

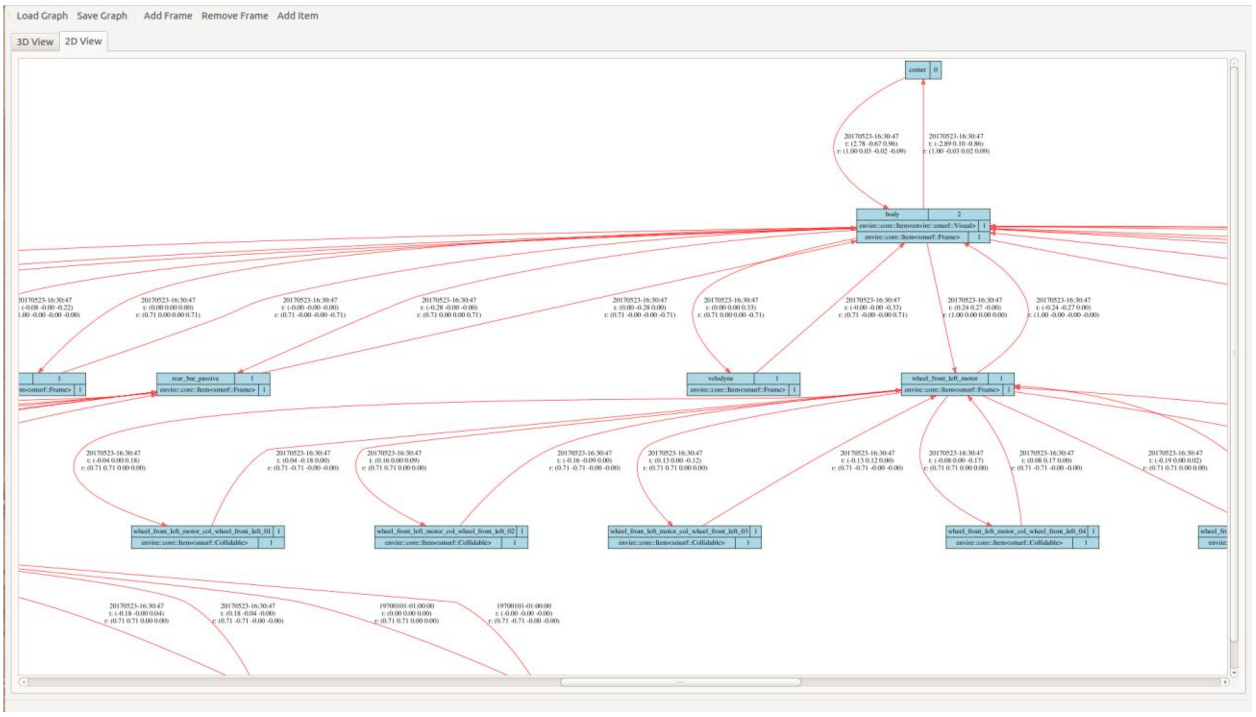
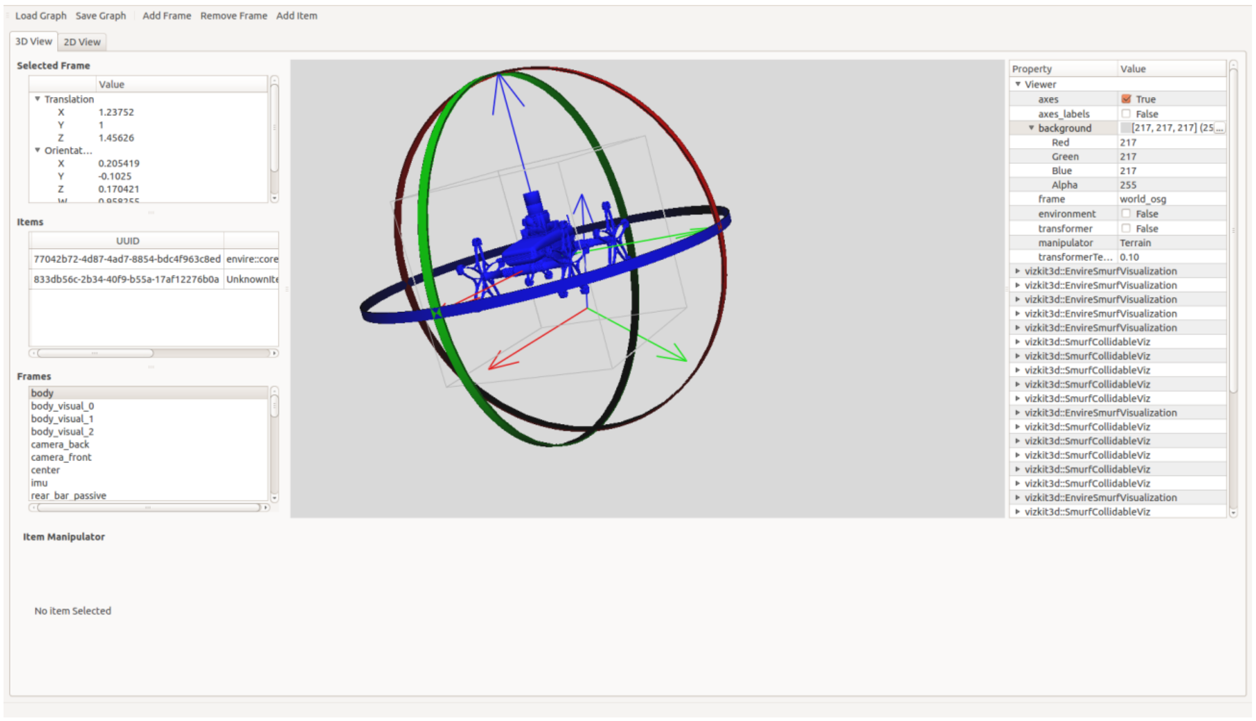


Figure 21: The 2D representation of the Envire graph can be visualized using Envire Visualization. The 2D view presents all the transformation as well as the objects stored in each frame with their correspondent type name. In this image a robot model is visualized.



Selected Frame

	Value
▾ Translation	
X	1.23752
Y	1
Z	1.45626
▾ Orientat...	
X	0.205419
Y	-0.1025
Z	0.170421
W	0.958255

Items

UUID	
77042b72-4d87-4ad7-8854-bdc4f963c8ed	envire::core
833db56c-2b34-40f9-b55a-17af12276b0a	UnknownIte

Frames

body
body_visual_0
body_visual_1
body_visual_2
camera_back
camera_front
center
imu
rear_bar_passive

Item Manipulator

No item Selected

Property	Value
▾ Viewer	
axes	✓ True
axes_labels	☐ False
▾ background	[217, 217, 217] (25...
Red	217
Green	217
Blue	217
Alpha	255
frame	world_osg
environment	☐ False
transformer	☐ False
manipulator	Terrain
transformerTe...	0.10
▸ vizkit3d::EnvireSmurfVisualization	
▸ vizkit3d::EnvireSmurfVisualization	
▸ vizkit3d::EnvireSmurfVisualization	
▸ vizkit3d::EnvireSmurfVisualization	
▸ vizkit3d::EnvireSmurfVisualization	
▸ vizkit3d::SmurfCollidableViz	
▸ vizkit3d::SmurfCollidableViz	
▸ vizkit3d::SmurfCollidableViz	
▸ vizkit3d::SmurfCollidableViz	
▸ vizkit3d::SmurfCollidableViz	
▸ vizkit3d::EnvireSmurfVisualization	
▸ vizkit3d::SmurfCollidableViz	
▸ vizkit3d::SmurfCollidableViz	
▸ vizkit3d::SmurfCollidableViz	
▸ vizkit3d::SmurfCollidableViz	
▸ vizkit3d::SmurfCollidableViz	
▸ vizkit3d::EnvireSmurfVisualization	
▸ vizkit3d::SmurfCollidableViz	

Figure 22: The 3D representation of the Envire graph can be visualized using Envire Visualization. The different transformations, frames as well as the objects contained them can be inspected. In this visualization the visual components of a robot model are shown.

6 Inter-OG Interface description

6.1 OG3 Product Definition

OG3 offers the Data Fusion Module for processing sensor data produce and generate from it the pose information and maps, required by the planning module (OG2). In Figure 23, the modules of autonomy (OG2), perception (OG3) and sensor (OG4) are presented as well as the RCOS (OG1).

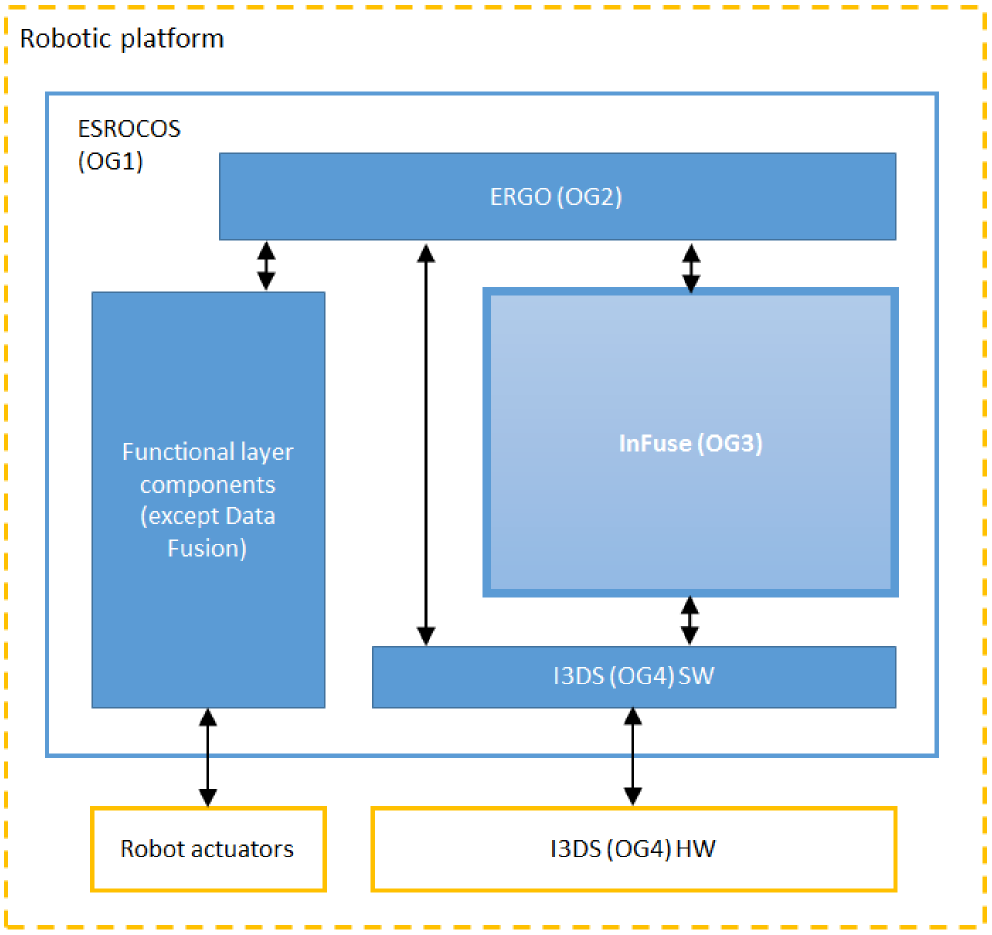


Figure 23:: The figure above allows to identify the potential set of interactions between OG3 on the one hand and OG2 and OG4 on the other hand. OG4 is understood as including a hardware part – i.e. the physical sensors, and a software part consisting of the drivers and pre-processing capabilities included in I3DS.

6.1.1 Identification of OG3 Interactions with other OGs

OG3 is controlled by OG2 in two ways: (1) OG2 is responsible for the initialization of OG3 and (2) OG2 through request of data products triggers the activation in OG3 of the most suitable Data Fusion Processing Compounds to satisfy the request. OG2 is also responsible for the activation of OG4 (sensor).

Although OG2 is responsible for the decisions in the overall architecture, OG3 can trigger the activation of operational modes of OG4. These operational modes determine the preprocessing steps, the sensor data generated and the interfaces through which is sent. Each pre-processed sensor data delivered by

OG4 is associated to its correspondent interface. Each interface is associated to only one specific type of sensor data and pre-processing steps.

The data products generated by OG3 will also be communicated through specific interfaces for each data type associated to the request received. In Figure 24 the consensual approach for the interactions of OG3 with other OGs is summarized.

The requests for sensor data will always go through OG3. In order to increase efficiency, only the fused data will come from OG3 and any direct output from OG4 requested by OG2 will come directly from OG4 to OG2.

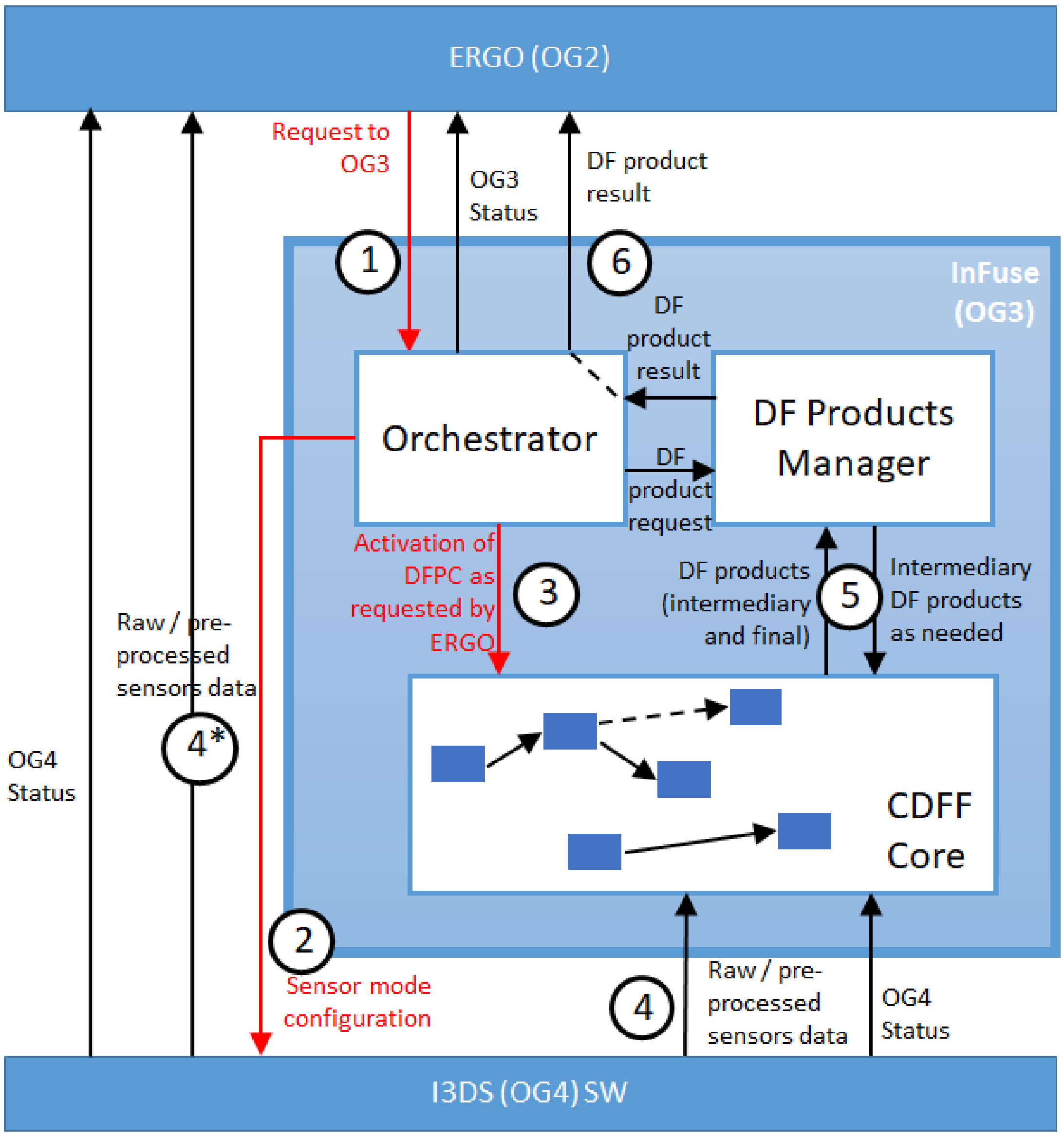

ERGO (OG2)
Request to OG3
OG3 Status
DF product result
1
6
InFuse (OG3)
DF product result
Orchestrator
DF Products Manager
DF product request
Activation of DFPC as requested by ERGO
3
DF products (intermediary and final)
5
Intermediary DF products as needed
Raw / pre-processed sensors data
4*
OG4 Status
CDFF Core
2
Sensor mode configuration
4
Raw / pre-processed sensors data
OG4 Status
I3DS (OG4) SW

Figure 24: Summary of the interactions of OG3 with other OGs

6.1.2 Data Products Generated by OG3

The data products generated by OG3 are specified by the needs expressed by OG2-ERGO in ERGO_D1.2 “System Requirements”. They are split in two families: information related to the environment (terrain or target), and information related to the position of the controlled system (rover or servicing satellite) with respect to the environment.

Note that besides the data products exported to OG2-ERGO, OG3 generates data products for its own needs: they are not listed in this document.

6.1.3 Planetary reference implementation

The main information related to the environment exploited by OG2-ERGO are Digital Elevation Maps (DEM), a 2.5D representation of the terrain defined over a regular Cartesian grid, which encodes terrain heights. These maps are possibly completed with an Uncertainty Map and a Soil Type Map, that share the same structure, but encode respectively the uncertainty on the height and the soil type.

Depending on the data that has been fused to produce these maps, their resolution, extent and reference frame differ. ERGO_D1_2 has defined three different maps:

- The *RoverMap* only integrates data perceived from a single rover position, and is expressed in the rover reference frame (this map is used to detect and avoid obstacles while driving)
- The *FusedRoverMap* integrates data perceived from various recent positions of the rover, and is expressed in a local terrain reference frame (this map is used to plan local rover trajectories)
- The *FusedTotalMap* integrates all the data gathered by the rover and other available information, and is expressed in a global absolute frame (this map is used to plan long term itineraries)

Note the associated data structures are defined by the following parameters: map extent (rectangular area, in meters), map resolution (size of the DEM cell, in meters), and map content (height, uncertainty, soil type).

Rover localization information is required by OG2 for the following purposes:

- At a rather high frequency (10Hz), to control the execution of the short term planned paths
- To plan local paths, so as to generate trajectories that follow the long term planned path
- To plan long term path using in particular the orbiter map

What distinguishes pieces of localization information (besides the inner OG3 processes that produce them) is the reference frame in which they are expressed. OG3 proposes the following reference frames:

- AbsoluteFrame: is the “geo-reference” frame attached to the planet
- GlobalTerrainFrame: is a reference frame in which is defined a whole navigation mission
- (a long term path). The Cartesian Coordinates of the frame should be x/y/z = East/Nort/Up.

- LocalTerrainFrame: is a reference frame in which are defined the local trajectories. For this reference frame the z component corresponds to upward.
- RoverFrame is the frame associated to the rover body. The localization information pertains to this frame. Its orientation is usually "x-forward, y-leftward, z-upward".

Given the fact that the the various localization information are produced by different processes, at different frequencies (and sometimes upon request) and express an information in different reference frames, we define three different interfaces:

- LocalPose: Pose of the BodyFrame in the LocalTerrainFrame
- GlobalPose: Pose of the BodyFrame in the GlobalTerrainFrame
- AbsolutePose: Pose of the BodyFrame in the AbsoluteFrame

Each of these pose consists of 3-axes attitude and 3-axes position information, with associated uncertainties (format to be defined: uncertainties on the 6 pose parameters, covariance matrix …).

6.1.4 Orbital reference implementation

The sole information related to the environment in the orbital scenario are the following:

1. The partial 3D model of the target spacecraft, built from data gathered by the servicing chaser spacecraft
2. The localization information is the relative position of the target Body Frame with respect to the Servicing Body Frame, associated to relative speeds.

6.2 Internal Interfaces

In this section the list of the methods identified for the internal interfaces is provided. In Figure 25 a component diagram showing the interfaces is presented.

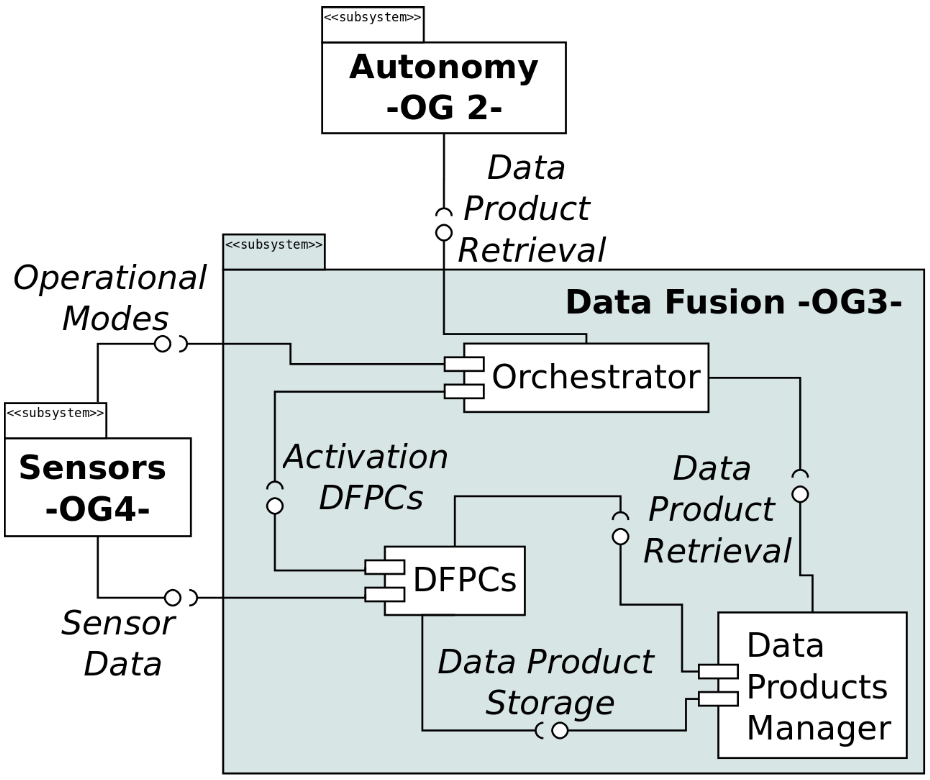


Figure 25: The internal interfaces are (1) Data Product Retrieval and (2) Data Product Storage implemented by the Data Product Manager and (3) Activation DFPCs implemented by the DFPCs

6.2.1 Data Product Storage

This interface is implemented by the DPM to receive requests for the storage of data products. Details for these requests are still to be defined internally.

Interface	Method	Description
DPMI	storeDEM	A DEM data product is passed as parameter which should be stored by the DPM
DPMI	storePose	A Pose data product is passed as parameter which should be stored by the DPM

Table 2: Interface Data Product Storage implemented by the Data Products Manager

6.2.2 Data Product Retrieval

This interface is implemented by the DPM to answer requests for the retrieval of data products. Details for these requests are still to be defined internally.

Interface	Method	Description

DPMI	getDEM	A DEM data product or the serialized reference to it is returned.
DPMI	getPose	A Pose is returned by the DPM.

Table 3: Interface Data Product Retrieval implemented by the Data Products Manager

6.2.3 Activation DFPCs

Table 4 presents the methods of the DFPC interface. In this case, to clarify the relationship with the Data Fusion Nodes the Data Fusion Node Common Interface related methods are also included.

Interface	Method	Description
DFNCI	initialize	Activates a DFN. It can be called multiple times, e.g. after stop has been called or before any other method has been called.
DFNCI	stop	Deactivates a DFN. Can only be called after initialize has been called.
DFPC	initialize	Calls 'initialize' for each DFN of the DFPC.
DFPC	stop	Calls 'stop' for each DFN of the DFPC.

Table 4: Interface of the DFPCs for the Orchestrator

6.3 Interface with OG1

The developer designs a deployment network. If necessary, type conversions between internal types and externally communicated types must be implemented by the developer. Transmitted data types can be custom designed by the developer (in agreement with other relevant OGs) or are predefined (as defined in OG1). The developer integrates a component network in the interface view. OG1 will generate code scaffolds for each OG1 component that will be implemented by the developer. The source code of the whole component or a static library must be provided by the developer. In the deployment view, the developer will specify how the components will be deployed on the target system. In Figure 25 the workflow to integrate the products from OG3 in OG1 is presented.

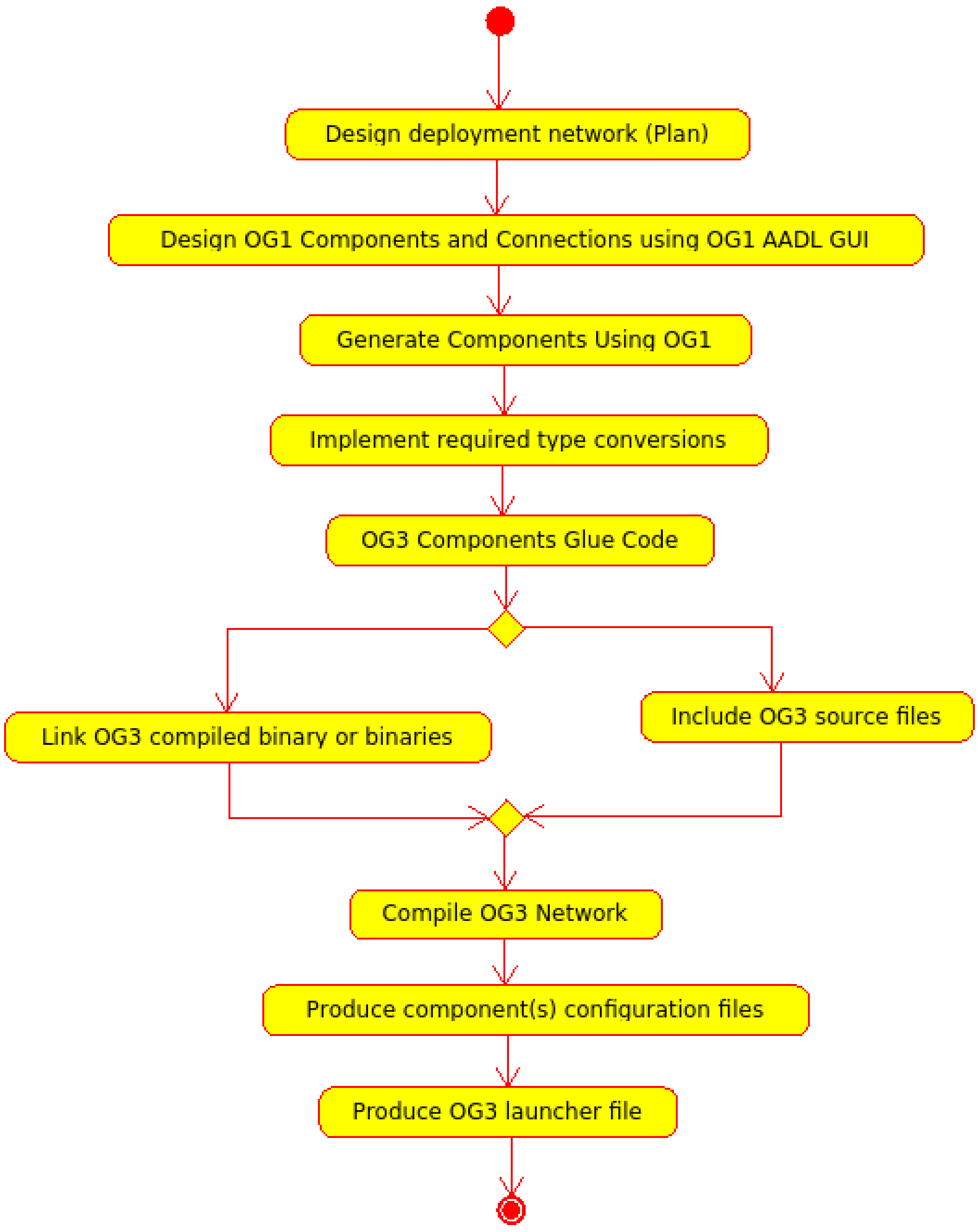

Design deployment network (Plan)
Design OG1 Components and Connections using OG1 AADL GUI
Generate Components Using OG1
Implement required type conversions
OG3 Components Glue Code
Link OG3 compiled binary or binaries
Include OG3 source files
Compile OG3 Network
Produce component(s) configuration files
Produce OG3 launcher file

Figure 26: Workflow to integrate OG3 in OG1

6.3.1 I/F Requirements

OG3-OG1/S/001/001/V0.1

Data View: Data that will be transmitted between OG1 components will be represented by ASN.1 types. OG1 provides an interface to design new data types that are transmitted between components.

OG3-OG1/S/001/002/V0.1

A predefined set of types that can be used by OG3 will be provided by OG1 (base types).

OG3-OG1/S/002/001/V0.1

Interface view: OG1 provides a graphical user interface to design component networks. It will be possible to generate a C++ code scaffold for each component of the network. Code scaffolds will be filled by OG3 with glue code (e.g. type conversions) so that the components that will be developed in OG3 (Data Product Manager, the Orchestrator and Data Fusion Processing Compounds) are available in OG1. For the cases were InFuse components use internally ASN1 types, less glue code will be needed when integrating with ESROCOS (e.g. type conversions not needed).

OG3-OG1/S/002/002/V0.1

There are multiple options how the components developed in OG3 can be mapped to OG1 modules:

1. Data Product Manager (DPM), Orchestrator, and Data Fusion Processing Compounds (DFPC) form a single OG1 module.
2. DPM and Orchestrator are in separate OG1 module. Each DFPC is in a separate OG1 module.
3. The Orchestrator is in a separate module, the DFPCs and the DPM are in a single module.
4. DFPCs can be fragmented to the point of having each data fusion node as an OG1 module

OG3 won’t control the scheduling of any OG1 module (e.g. stopping a module).

OG3-OG1/S/003/001/V0.1

OG1 component lifecycle: OG1 components provide a predefined component life cycle that includes a configuration and a runtime state.

OG3-OG1/S/003/002/V0.1

It must be possible to set the configuration of a OG1 node at least once before processing begins.

OG3-OG1/S/004/001/V0.1

OG1 provides a library for geometric coordinate frame transformations.

OG3-OG1/S/004/002/V0.1

OG1 provides a forward and instantaneous kinematics computation library.

OG3-OG1/S/005/001/V0.1

Deployment view: The component network that has been defined in the interface view must be compiled to an executable program. OG1 must provide the functionality.

OG3-OG1/S/005/002/V0.1

Build system: Components that are developed in OG3 will either be compiled to a single static library that can be linked into an OG1 node or all the necessary C++ headers and implementation files will be available so that they can be used to compile the OG1 component.

6.4 Interface with OG2

OG2 is responsible for planning during the complete mission. The internal state of OG3 needs to be initialized to begin accepting requests to start DF processes. As long as the mission continues, OG2 can send to OG3 to generate fused data products (maps or poses). These request will be done through a generic querying method. Once the request has been received and processed, OG3 will either (1) confirm that the request will be performed, or (2) in case that the request can not be performed, report to OG2 with the correspondent message. After generating the correspondent data product, it is delivered to OG2. The delivered data product can be a periodic data update or a unique data product. The streams are provided to OG2 synchronously (possibly a polling mechanism to check if the data is available) interface which is univocally associated to the data product requested. At any point after performing a request, OG2 can send a cancellation of that same request. In Figure 26 the communication workflow between OG2 and OG3 is presented.

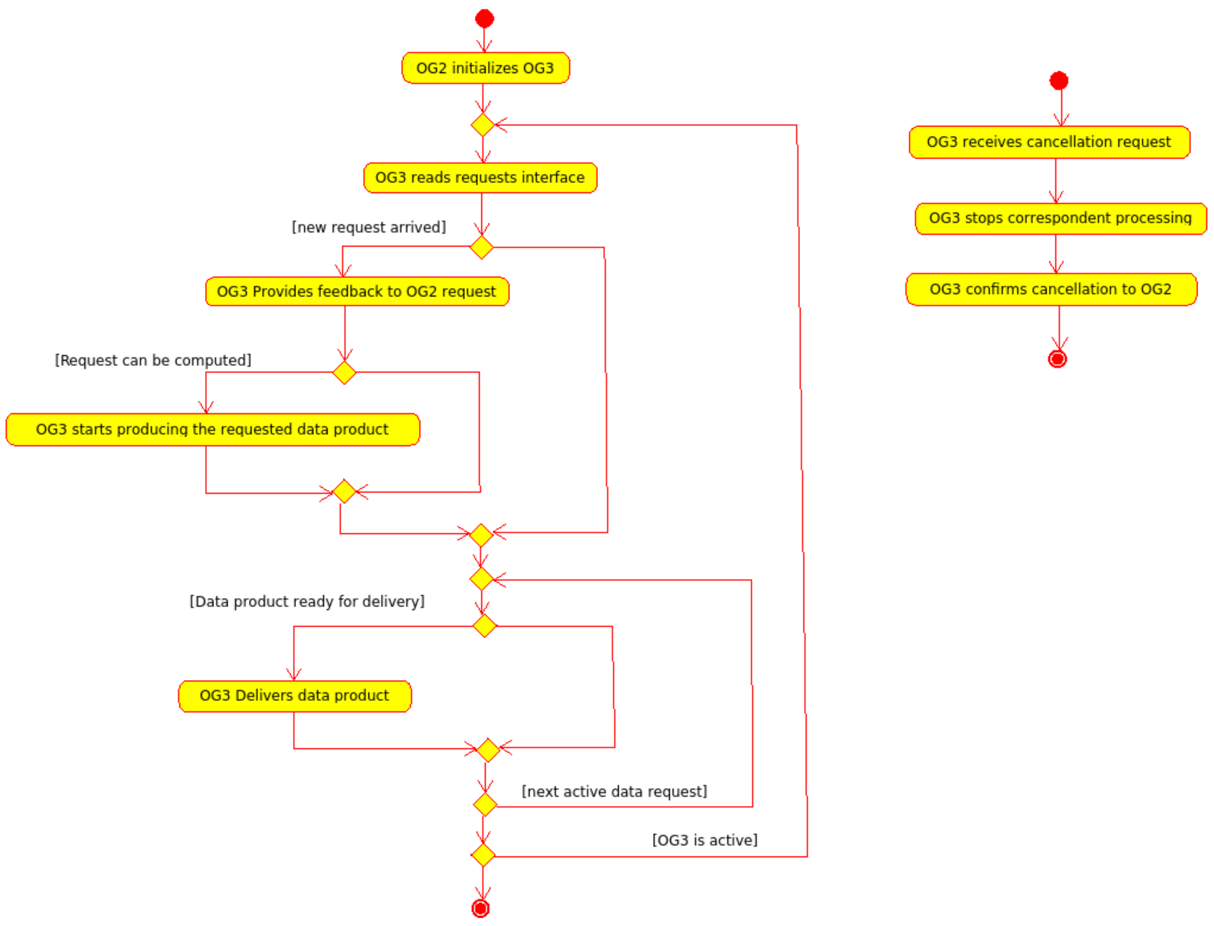

OG2 initializes OG3
OG3 reads requests interface
[new request arrived]
OG3 Provides feedback to OG2 request
[Request can be computed]
OG3 starts producing the requested data product
[Data product ready for delivery]
OG3 Delivers data product
[next active data request]
[OG3 is active]
OG3 receives cancellation request
OG3 stops correspondent processing
OG3 confirms cancellation to OG2

Figure 27: Communication workflow between OG2 and OG3. On the left diagram the main operational loop is presented. On the right one, the operation to stop a request.

6.4.1 I/F Requirements

OG2-OG3/S/001/002/V0.1

OG3 provides an interface to OG2 to initialize, reset or put the CDFF framework into an idle state (no processing). This can be used by OG2 to command the runtime internal state of the data fusion framework (not to be confused with the OG1 run time states). The specific parameters could be desired state of system:

1. Initialize - Inform CDFF to be prepared for accepting requests from OG2
2. Idle - Inform CDFF to stop processing DFPCs from previous requests
3. Reset - Inform CDFF to clear the previous OG2 requests that are currently being processed by stopping the corresponding DFPCs. This will enable OG2 to send a completely new set of requests.

OG2-OG3/S/002/001/V0.1

OG3 provides an interface for OG2 to query and get a notification on the current state of the CDFF framework to OG2. This can be used by OG2 to check the state before posting a new request that influences the internal runtime state of OG3 as indicated in OG2-OG3/S/001/001/V0.1**.** The return type is the current state of the system with associated data. This needs to be elaborated in further iterations.

OG2-OG3/S/003/001/V0.1

OG3 provides an interface for OG2 to query the CDFF orchestrator based on a set of criteria of parameters for activating DFPCs. OG3 internally maintains a list of possible CDFFs that can be selected indirectly by OG2 based on a specific set of parameters such as:

1. List of sensors activated by OG2
2. Default configurations or operational modes of sensors
3. Type of fused maps (rover map, fused rover map and fused total mapl)
4. Resolution of fused data (maps)
5. Update frequency (localization and map updates)
6. Area coverage (map)
7. Time bounds for providing specific fused data for aperiodic requests
8. Priority of request (to manage request queue in OG3)

The match between OG2 request and the selection of OG4's mode of operation will be clear and explicit. All of this will be pre-defined, for each reference implementation, so that in the scope of the project neither OG2 nor OG3 will have provision decision making mechanisms for what concerns the sensors selection. The decided approach eliminates the possibility of sending to OG4 conflicting operational modes commands from OG2 and OG3.

The orchestrator software component within CDFF has the task to map the criteria to a specific DFPC that can optimally satisfy the above requirements.

OG2-OG3/S/003/002/V0.1

This interface can be used by OG2 to get an update regarding the runtime status of the DFPC in terms of the quality of data inputs from OG4 (raw and pre-processed) and the quality of fused data products in terms of maps and localization data.

OG2-OG3/S/004/001/V0.1

Interface provided by OG2 to OG3 for providing fused data updates. There are 2 mechanisms that are foreseen:

1. For aperiodic requests from OG2 (Ex. absolute localization, global fused map etc.), OG3 will notify OG2 regarding the availability of fused data and provide access to the data buffer with corresponding information on the data type. The notification is foreseen to be asynchronous due the inherent delay in producing fused data for a specific query from OG2.
2. For OG2 requests that require OG3 to generate periodic data (local rover map, high frequency local pose estimates etc.), a predefined data stream for specific data types will be used to send the fused data periodically to OG2.

OG2-OG3/D/001/001/V0.1

This interface produces the Rover Map.

OG2-OG3/D/002/001/V0.1

This interface produces the Fused Rover Map

OG2-OG3/D/003/001/V0.1

This interface produces the Fused Total Map

OG2-OG3/D/004/001/V0.1

This interface produces the LocalPose Pose of the BodyFrame in the LocalTerrainFrame

OG2-OG3/D/005/001/V0.1

This interface produces the 3D model of the target spacecraft.

OG2-OG3/D/006/001/V0.1

This interface produces the GlobalPose Pose of the BodyFrame in the GlobalTerrainFrame

OG2-OG3/D/007/001/V0.1

This interface produces the AbsolutePose Pose of the BodyFrame in the AbsoluteFrame

OG2-OG3/D/008/001/V0.1

This interface is the relative pose (3 axes position and 3 axes attitude) of the target Body Frame expressed in the chaser Body Frame, with associated uncertainties (format to be defined)

OG2-OG3/D/009/001/V0.1

This interface is the relative speed (3 axes translation speeds and 3 axes rotation speeds) of the target Body Frame expressed in the chaser Body Frame, with associated uncertainties (format to be defined)

6.5 Interface with OG4

Before any interaction between OG4 and OG3 can take place, both modules have to be initialized. It is assumed at this stage that OG2 is responsible for the initialization of the different components and OG1 is the responsible of the initialization of OG2.

As long as OG3 is active, the activities of OG3 will can be summarized as follows: As long as there is an active Data Product Request, OG3 will keep activate the correspondent most suitable *preset* (i.e. configuration values agreed in advance) of OG4 and read the from the correspondent data stream. Each preset has associated OG4 Producer Interfaces (PI) in a unitary manner (i.e. one PI only produces one type of data according to the associated preset). OG3 will then the correspondent PI's associated to the preset. If there is not any active Data Product Request, nothing will be done.

In Figure 27 the communication workflow between OG3 and OG4 is depicted.

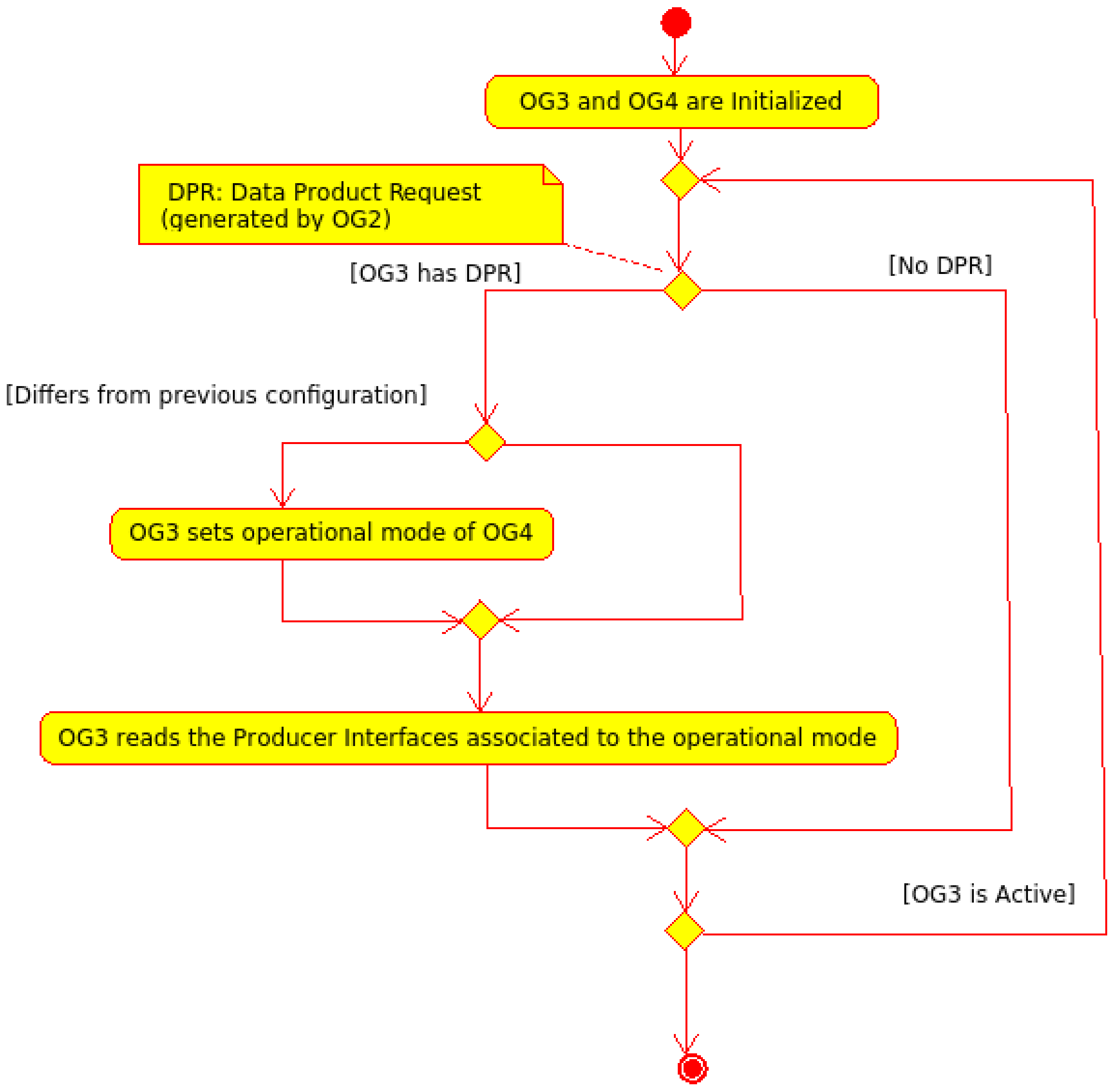

OG3 and OG4 are Initialized
DPR: Data Product Request
(generated by OG2)
[OG3 has DPR]
[No DPR]
[Differs from previous configuration]
OG3 sets operational mode of OG4
OG3 reads the Producer Interfaces associated to the operational mode
[OG3 is Active]

Figure 28: Communication workflow between OG3 and OG4

6.5.1 Sensor Data Inputs from OG4

Sensor Type	Data type	ESROCOS ASN 1 data type[5]	Metadata	Modification required in the ASN1 type
HR camera	Image (RGB) Transform metadata must be available.	asn/Frame.asn	frame-time, received-time, attributes, datasize, data-depth, pixel-size, row-size, frame-mode, frame-status, calibration parameters *Geometric frame name (not in type definition)*	Calibration parameters
Stereo camera	Disparity map	Similar to asn/Frame.asn type for frame but with float values	frame-time, received-time, attributes, datasize, data-depth, pixel-size, row-size, frame-mode, frame-status, *Geometric frame name (not in type definition)*	**The type does not exist**
	Depth map	asn/DepthMap.asn	ref-time, timestamps,	No modifications required

[5] ASN1 types defined in ESROCOS: https://github.com/ESROCOS/types-base

Sensor Type	Data type	ESROCOS ASN 1 data type[5]	Metadata	Modification required in the ASN1 type
			vertical-projection, horizontal-projection, vertical-interval, horizontal-interval, vertical-size, horizontal-size, remissions, *Geometric frame name (not in type definition)*	
	Point cloud[6] arranged in a so that correspondences between points and pixels and the reference camera (usually the left) are possible	asn/Pointcloud.asn	ref-time, colors, *Geometric frame name (not in type definition)*	No modifications required
	Raw images	asn/Frame.asn	frame-time, received-time, attributes, datasize, data-depth, pixel-size, row-size, frame-mode, frame-status,	Calibration parameters have to be included

[6] Pointcloud units should be meters

Sensor Type	Data type	ESROCOS ASN 1 data type[5]	Metadata	Modification required in the ASN1 type
			calibration parameters *Geometric frame name (not in type definition)*	
	Corrected images	asn/Frame.asn	frame-time, received-time, attributes, datasize, data-depth, pixel-size, row-size, frame-mode, frame-status, calibration parameters, Baseline of the stereovision bench, *Geometric frame name (not in type definition)*	Calibration parameters have to be included. Baseline of the stereovision bench has to be included.

Sensor Type	Data type	ESROCOS ASN 1 data type[5]	Metadata	Modification required in the ASN1 type
LiDAR[7]	Point cloud[8]	asn/Pointcloud.asn	ref-time, colors, **reflectances**, *Geometric frame name (not in type definition)*	The Reflectances are missing in the type, this metainformation about each point of the pcl can be used to determine information about the object. Some lidars provides sequences of reflectances and not just one. Thus, the sequence of Reflectance objects. The Reflectance object can be defined as: Reflectance SEQUENCE (SIZE(1..maxPointcloudSize)) OF float
	2D laser scan	types-sensor_samples/asn/LaserScan.asn	ref-time, start-angle, angular-resolution, speed, ranges, minRange, maxRange, remission, **reflectances** *Geometric frame name (not in type definition)*	The Reflectances are missing in the type, this metainformation about each point of the pcl can be used to determine information about the object.

Sensor Type	Data type	ESROCOS ASN 1 data type[5]	Metadata	Modification required in the ASN1 type
TIR camera	Image	asn/Frame.asn	frame-time, received-time, attributes, datasize, data-depth, pixel-size, row-size, frame-mode, frame-status, Calibration parameters **spectral_range** *Geometric frame name (not in type definition)*	The spectral range has to be included it can be defined as: SEQUENCE (SIZE(1..2)) OF T-Float

[7] Depth map if specific to the COTS sensor

[8] Pointcloud units should be meters

Sensor Type	Data type	ESROCOS ASN 1 data type[5]	Metadata	Modification required in the ASN1 type
ToF camera	Point cloud[9]	asn/Pointcloud.asn	Intensity Image, Confidence map, ref-time, colors *Geometric frame name (not in type definition)*	Intensity Image and Confidence map have to be included in a new type extending the Point Cloud For the intensity image and the Confidence map the asn/Frame.asn type can be used.
Radar	Point cloud[10]	asn/Pointcloud.asn	ref-time, colors *Geometric frame name (not in type definition)*	No modifications required

[9] Pointcloud units should be meters

[10] Pointcloud units should be meters

Sensor Type	Data type	ESROCOS ASN 1 data type[5]	Metadata	Modification required in the ASN1 type
Structured light camera	Depth map	asn/DepthMap.asn	ref-time, timestamps, vertical-projection, horizontal-projection, vertical-interval, horizontal-interval, vertical-size, horizontal-size, remissions *Geometric frame name (not in type definition)*	No modifications required
	Raw image	asn/Frame.asn	frame-time, received-time, attributes, datasize, data-depth, pixel-size, row-size, frame-mode, frame-status, calibration parameters, Baseline of the stereovision bench, *Geometric frame name (not in type definition)*	Calibration parameters have to be included.

Sensor Type	Data type	ESROCOS ASN 1 data type[5]	Metadata	Modification required in the ASN1 type
Structured light camera	Corrected images	asn/Frame.asn	frame-time, received-time, attributes, datasize, data-depth, pixel-size, row-size, frame-mode, frame-status, calibration parameters, Baseline of the stereovision bench, *Geometric frame name (not in type definition)*	Calibration parameters have to be included.
IMU	Inertial measurements	asn/IMUSensors.asn	Timestamp *Geometric frame name (not in type definition)*	No modifications required
End effector force/torque sensor	Linear forces or torques at end effector	Asn1 type for Wrench[11]	Timestamp, Geometric frame name	No modifications needed

[11] ESROCOS ASN1 Type for Wrench: https://github.com/ESROCOS/types-base/blob/master/asn/Wrench.asn

Sensor Type	Data type	ESROCOS ASN 1 data type[5]	Metadata	Modification required in the ASN1 type
Start tracker	Rotation with respect to tracked star.	asn/RigidBodyState.asn	timestamp, sourceFrame, targetFrame	No modification required

Table 5: Sensor data inputs from OG4

Metadata Type	Data Type	Asn1 Data Type	Modification required in the ASN1 type definition
Time	Time	asn/Time.asn	No modifications required
Frame-size-t	Width height	types-sensor_samples/asn/Frame.asn	No modifications required
Frame-mode-t,	mode-undefined, mode-grayscale, mode-rgb, mode-uyvy, mode-bgr, mode-rgb32, raw-modes, mode-bayer, mode-bayer-rggb, mode-bayer-grbg, mode-bayer-bggr, mode-bayer-gbrg, compressed-modes, mode-pjpg, mode-jpeg, mode-png	asn/Frame.asn	No modifications required
Frame-status-t	status-empty, status-valid, status-invalid	asn/Frame.asn	No modifications required
Frame-attrib-t	data T-String, att-name T-String	asn/Frame.asn	No modifications required
Calibration Parameters	4 floats for the intrinsic matrix: - alpha_u, alpha_v, u_0, v_0 5 floats for the distortion: - k1, k2, k3, t1, t2		Type doesn't exists in ASN1 definitions

Table 6 Metadata Data Types

6.5.2 I/F Requirements

OG3-OG4/D/001/000/V0.1

OG3 supports ASN.1 messages defined by OG4 following OG1 specification for all communication with OG4.

OG3-OG4/D/002/000/V0.1

OG4 provides data samples (simulated and actual) to OG3 throughout the course of the project.

OG3-OG4/F/001/000/V0.1

OG4 I3DS ensures compatibility with OG3 Common Data Fusion Framework.

OG3-OG4/F/002/000/V0.1

OG4 supports operational modes as the primary way of controlling the sensor suite. Each mode consists of a set of active sensors with a given configuration that is verified to operate correctly with regards to throughput and latency.

OG3-OG4/F/003/000/V0.1

OG4 provides a list of specific modes per use-case to operate the sensor suite ensuring the limitations of available data rates, bandwidth and power consumption.

OG3-OG4/F/004/000/V0.1

OG4 receives commands from OG3 to enable pre-defined sensors configurations and sensors data provisions.

OG3-OG4/F/005/000/V0.1

OG4 provides sensor measurements and accepts commands through the OG1 middleware. The supported sensors are listed in the product tree.

OG3-OG4/F/006/000/V0.1

OG4 performs pre-processing for the Time Of Flight Camera, the Stereo Camera, the High Resolution Camera, the Thermal InfraRed Camera, the High Frequency Radar and the Lidar. **OG3 will have access to the intermediate products**.

OG3-OG4/F/007/000/V0.1

OG4 performs the following pre-processing for the Time of Flight Camera: Point Cloud Generation. **OG3 will have access to the intermediate products. In this case, Depth Map.**

OG3-OG4/F/008/000/V0.1

OG4 performs the following pre-processing for the Stereo Camera: Lens Vignetting, Histogram Equalisation, Optical Distortion (Lens radial and geometric distortion correction), Stereo Rectification, Dense Stereo Matching Disparity Map Production (TBD), Sparse Stereo Matching (TBD), Dense Depth Map Production (TBD), Sparse Depth Point Cloud (TBD). **OG3 will have access to the intermediate products.**

OG3-OG4/F/009/000/V0.1

OG4 performs the following pre-processing for the High Resolution Camera: Lens Vignetting Correction, Histogram Equalisation, Optical Distortion Correction, Feature Detection and Temporal Matching, Structured Light Pattern Detection, Depth Map Point Cloud Production from Structured Light (TBD). **OG3 will have access to the intermediate products.**

OG3-OG4/F/010/000/V0.1

OG4 performs the following pre-processing for the Thermal InfraRed Camera: Lens Vignetting Correction, Histogram Equalisation, Optical Distortion Correction, Feature Detection and Temporal Matching. **OG3 will have access to the intermediate products.**

OG3-OG4/F/011/000/V0.1

OG4 performs the following pre-processing for the HF Radar: Point Cloud Generation. The radar information could be used for long or short distance mapping.

OG3-OG4/F/012/000/V0.1

OG4 performs the following pre-processing for the LIDAR: Point Cloud Generation.

OG3-OG4/F/013/000/V0.1

OG4 provides to OG3 either raw sensors data or pre-processed data. Both cannot be performed simultaneously.

OG3-OG4/F/014/000/V0.1

OG4 performs time stamping for pre-processed and raw data.

OG3-OG4/O/001/000/V0.1

The operational mode can become parameters to be configured in the future.

OG3-OG4/O/002/000/V0.1

OG4 does not check the sender of command messages; therefore OG2 and OG3 must coordinate commanding of OG4 to avoid conflicting commands.

OG3-OG4/O/003/000/V0.1

OG4 does not store measurements for later retrieval; therefore OG3 receives the data at the specified rate[12].

OG3-OG4/P/001/001/V0.1

Before any interaction between OG4 and OG3 can take place, both modules have to be initialized. It is assumed at this stage that OG2 is responsible for the initialization of the different components and OG1 is the responsible of the initialization of OG2.

OG3-OG4/S/001/001/V0.1

[12] Related Issue: Not clear if OG1 or OG2 controls the communication buffers between OG4 and OG3.

An interface is required for OG3 to be able to configure a limited set of parameters of a group of sensors providing data to a DFPC or alternatively to select among pre-defined configuration modes that are compatible with a DFPC.

After OG2 triggers a set of sensors for a particular task, the default configuration of the sensors might not be optimal for data fusion due to changing conditions (Ex. ambient lighting, camera exposure etc.) that produce multiple outliers or unusable data (Ex. saturated images, lack of correspondence features etc.).

This interface aims to provide a mechanism for OG3 to set a suitable configuration for some limited parameters within each operational mode of OG4 for some sensors relevant for a DFPC to produce the required quality of fused data products (e.g. exposure of the camera or framerate).

OG3-OG4/S/001/002/V0.1

OG4 provides raw or pre-preprocessed sensor data to OG3 from the sensors through this interface. The sensors and the corresponding pre-preprocessing steps are configured by the function OG3-OG4/S/001/001/V0.1.

OG3-OG4/S/001/003/V0.1

The preprocessing steps that the data has gone through in OG4 is known to OG3 because it is associated to the OG4 sensors operation mode selected by function OG3-OG4/S/001/002/V0.1. The software interfaces through which each sensor data is delivered, is defined in the operation mode of OG4. For each pre-processed sensor data there is an associated software interface which is only used for that pre-processed sensor data and is not used in any other operation mode unless exactly the same pre-processed sensor data is produced.

OG3-OG4/S/002/001/V0.1

OG 4 provides metadata related to the pre-processing steps the sensor data has gone through. The metadata is provided through a different interface than the data itself and it is timestamped.

OG3-OG4/S/002/002/V0.1

OG4 metadata to OG3 from the sensors and the sensor filters as configured by the function OG3-OG4/S/001/001/V0.1. In the configuration is included the interface for each metadata product.

OG3-OG4/S/003/001/V0.1

Once OG4 sends any data over the data interface, if OG3 is not ready to process the data, this will be lost. An interface is therefore required for OG3 to inform OG4 that a specific DFPC has been selected based on an OG2 request (OG2-OG3/S/003/001/V0.1) and that the DFPC is in a ready state to process sensor data from OG4. Hence this mechanism allows OG4 to send data after the DFPC is initialized and ready to process sensor data to begin producing fused data as outputs.

6.6 Rover and Manipulation Interface

6.6.1 Sensor Data Inputs from OG6

Some sensors which are typically available in robotics systems are currently not included in OG4 sensor's suite. Thus, an additional interface is defined which should be provided for validation by either (1) OG6, the facilitators framework, by (2) OG2 or (3) OG4.

Sensor Type	Data type	ASN 1 data type[13]	Metadata	Modifications Required in the ASN1 Type
Manipulator joint states	Joint angle positions and velocity	Asn1 type for JointState[14]	Timestamp, Geometric frame name	Position resolution Velocity resolution
Rover wheel encoders	Position of the joints of the wheels	Asn1 type for Joints[15]	Timestamp, Names, Geometric frame names	Position resolution Velocity resolution
Manipulator force/torque sensor	Linear forces or torques at joint	Asn1 type for Wrench[16]	Timestamp, Geometric frame name	No modifications needed
Manipulator force/torque sensors	Linear forces or torques at joints composition	Asn1 type for Wrenches[17]	Timestamp, Names, Geometric frame names	No modifications needed
Wheel-Based Odometry	Global pose estimate and differential pose estimate (based on wheel encoder measurements and rover model)	Asn1 type for RigidBodyState[18]	Timestamp, Source Frame, Targe tFrame	No modifications needed

Table 7 Data inputs from the Rover and Manipulation interface

[13] ASN1 types defined in ESROCOS: https://github.com/ESROCOS/types-base

[14] ESROCOS ASN1 type for JointState: https://github.com/ESROCOS/types-base/blob/master/asn/JointState.asn

[15] ESROCOS ASN1 Type for Joints: https://github.com/ESROCOS/types-base/blob/master/asn/Joints.asn

[16] ESROCOS ASN1 Type for Wrench: https://github.com/ESROCOS/types-base/blob/master/asn/Wrench.asn

[17] ESROCOS ASN1 Type for Wrenches: https://github.com/ESROCOS/types-base/blob/master/asn/Wrenches.asn

[18] ESROCOS ASN1 Type for RigidBodyState: https://github.com/ESROCOS/types-base/blob/master/asn/RigidBodyState.asn

6.6.2 I/F Requirements

OG3-OG6/D/001/000/V0.1

OG6 supports ASN.1 messages defined by OG6 following OG1 specification for all communication with OG6.

OG3-OG6/F/001/000/V0.1

OG6 provides raw or pre-preprocessed sensor data to OG3 from the manipulator Joints through this interface.

OG3-OG6/F/002/000/V0.1

OG6 provides raw or pre-preprocessed sensor data to OG3 from the Rover Wheel encoders through this interface.

OG3-OG6/F/003/000/V0.1

OG6 provides raw or pre-preprocessed sensor data to OG3 from a particular manipulator force/torque sensor through this interface.

OG3-OG6/F/004/000/V0.1

OG6 provides raw or pre-preprocessed sensor data to OG3 from sets of manipulator force/torque sensors through this interface

OG3-OG6/F/005/000/V0.1

OG6 provides global pose estimation based on the rover encoders and on the robot geometric model to OG3 through this interface

OG3-OG6/F/005/001/V0.1

OG6 provides differential pose estimationa based on the rover encoders and on the robot geometric model to OG3 through this interface

OG3-OG6/F/001/000/V0.1

OG4 provides sensor measurements and accepts commands through the OG1 middleware. The supported sensors are listed in the product tree.

7 Coding Guidelines

7.1 InFuse Coding Guidelines for C

7.1.1 Language

Guideline 1: [required] C code must be fully compliant with the ISO/IEC 9899:1990 standard (also known as C90).

Rationale: The RTEMS C Cross Compiler available at the time of writing is based on GCC version 4.4.6. Such version of GCC fully supports the C90 standard, while it is reported to have incomplete support for the C99 standard.

Guideline 2: [required] Assembly language shall not be used.

Rationale: Use of assembly language would reduce the portability and the maintainability of the source code.

7.1.2 Program Structure

Guideline 3: [required] The source code shall be written in modules, each module consisting of two files: the program file (.c) and the header file (.h). The header file shall declare the public types and function prototypes representing the interface provided by the module. The program file shall contain the actual implementation of the module functions. Variables and constants, used locally only, shall be declared in the program file.

Guideline 4: [required] The header file of a given module shall be included by any program module that needs access to the types, variables and functions exported by that module.

Rationale: The interface file must contain declarations of data entities such as variables and functions and not their definitions. Variables and functions declared in a header file must be declared as "extern".

Guideline 5: [required] Declarations of "extern" variables and functions shall only appear in header files.

Guideline 6: [required] Each program file shall include its own header file.

Rationale: This provides consistency checking as well as making types and symbolic constants defined in the header file available also in the program file.

Example:

File data_flow_functions.c*:*

```
#include <stdio.h>

#include "data_flow_functions.h"

……
```

Guideline 7: [required] An appropriate #ifndef directive shall be placed at the beginning of each header file in order to avoid multiple inclusions of the same header file.

Example:

File header_file_name.h*:*

```
#ifndef header_file_name_h

#define header_file_name_h




…  /* body of header file */




#endif   /* end of header_file_name_h */
```

Guideline 8: [required] A full function prototype shall be declared in the header file (.h) for each globally available function.

Guideline 9: [required] The content of source files shall be structured according to the following order: include directives, constants and macros, types, structures, variables and functions.

Rationale: Symbolic constants and macros are likely to be used within type and structs declarations, which in turn are likely to be used for declaring variables. Such variables are likely to be used in the functions. Grouping each category of entities together allows the programmer to find a particular declaration much more quickly than if entities were declared "as needed".

7.1.3 Variables

Guideline 10: [required] Variables shall be declared at the smallest possible level of scope.

Rationale: The scope can be global scope, file scope, function scope. The scope of variables should be restricted to functions where possible. File scope should only be used when variables need to have a wider scope than a single function. Use global variables only when they are really necessary. It is good practice in general to limit as much as possible the use global variables. Global variables lead to implicit coupling between modules making it difficult to remember or reason about every possible use. Moreover global variables increase the difficulties of the test activities, for example when testing a single unit.

Guideline 11: [required] Global variables shall always be initialized (initialization value provided at the point of variable definition).

7.1.4 Preprocessor

Guideline 12: [required] The use of the preprocessor shall be limited to the inclusion of header files and simple macro definitions. The use of conditional compilation directives must be kept to a minimum. Token pasting, variable argument lists (ellipses) and recursive macro calls are not allowed.

Rationale: Heavy use of the C preprocessor can make the code hard to decipher. Concerning conditional compilation, we note that with N conditional compilation directives, there might be up to 2^N possible versions of the code, each of which would have to be tested, causing a huge increase in the required test effort. Note: the standard boilerplate that avoids multiple inclusions of the same header file (discussed in another guideline in this document) is allowed.

Guideline 13: [advisory] Functions shall be preferred to macros.

Rationale: While function-like macros can provide a speed advantage over functions, functions provide a more robust mechanism. This is particularly true with respect to the type checking of parameters.

Guideline 14: [required] All macro arguments shall be enclosed in parentheses.

Rationale: Not using parentheses to enclose macro arguments can lead to unexpected behaviours when the macro is expanded.

Example:

```
#define RECIPROCAL(x) (1/x)

// Wrong! RECIPROCAL(1+1) --> (1/1+1) --> result is 2
```

```
#define RECIPROCAL(x) (1/(x))

// Correct! RECIPROCAL(1+1) --> (1/(1+1)) --> (1/2)
```

Guideline 15: [required] Expressions with a side effect shall not be provided as parameters in macro invocation.

Rationale: A macro might expand the parameter more than once, resulting in multiple (unwanted) occurrences of the side effect.

Example:

```
#define SQUARE(x) ((x)*(x)) /* proper macro definition */

z = SQUARE(y++);   /* wrong! z = ((y++)*(y++)); --> not what we wanted! */
```

7.1.5 Error Handling

Guideline 16: [required] Each calling function shall check the return value of non void functions, in order to detect execution errors.

Rationale: This rule is especially important when a called function can return an error code value. Also, when a function returns a pointer and that pointer is subsequently de-referenced, the program should first check that the pointer is not NULL. In case the called function has continuous numeric output, only critical values (according to the algorithm being implemented) will have to be checked.

Example:

```
// Open a communications channel, and check the return type

CHANNEL_T channel;

CHANNEL_RESULT_T result;


result = open_channel(&channel);

if (result == CHANNEL_ERROR)

{

        // Handle error

}
```

Guideline 17: [required] Each called function shall check the validity of the parameters provided by the caller.

Rationale: This rule is meant to ensure that any constraints for proper function execution are met. A function may have constraints on some of its input parameters (for example in terms of allowed range): such constraints should be checked in order to avoid unpredictable function behaviour. All parameter validity rules and constraints shall be documented in comments.

Example:

```
CHANNEL_RESULT_T open_channel(CHANNEL_T* channel)

{

        if (channel == NULL)

        {

                return CHANNEL_ERROR;

        }



        ...

}
```

Guideline 18: [required] Explicit run-time checks shall be included to avoid arithmetic errors such as divide by zero, overflow, underflow and loss of significant bits through shifting.

Rationale: In order to minimize software failures, programmers should add dynamic checks wherever there is potential for run-time errors to occur.

Example:

```
if (int_var_1 != 0)

{

        div_res = int_var_2 / int_var_1;

}

else

{

        // Handle error
```

```
    }
```

Guideline 19: [required] Commands received from external software interfaces shall be subject, prior to execution, to integrity and parameter range checks.

Rationale: Invalid or corrupted commands could be received from external interfaces: such commands should be rejected.

7.1.6 Miscellaneous

Guideline 20: [required] Dynamic memory allocation shall not be used after initialization.

Rationale: This rule allows the use of dynamic heap memory functions (e.g. malloc, calloc, realloc and free) only during the initialization phase of the software application. This is to prevent memory management bugs and enhance code predictability (execution time and memory usage). Note that some library implementations may use dynamic heap memory allocation to implement functions (for example functions in string.h): if this is the case then these functions shall also be avoided. When the software application is developed using a real-time multi-tasking operating system, the latter usually provides facilities such as memory pools, which can be used by tasks to dynamically allocate/deallocate memory blocks, as their memory requirements vary during the course of execution. The use of this type of facilities is allowed because they work on fixed, preallocated areas of memory and are therefore completely predictable in terms of maximum memory usage.

Guideline 21: [required] Symbolic constants shall be used in the code. "Magic" numbers and strings are expressly forbidden.

Rationale: Code containing literal constants can be very hard to understand because there may be no obvious reason why a particular number, size or string appears in the code. This makes maintenance of the code difficult. Maintenance or future modification of the code is also complicated because whenever one of these literal constants needs to be changed the programmer must examine every occurrence of that literal constant in the code to determine whether it is being used for the same purpose before being able to update it. It is far better to define a symbolic constant once for a particular meaning of a literal constant and then use this symbolic constant. This improves readability of the code in the areas where the symbolic constant is used, and distinguishes between different symbolic constants, which may have the same literal constant value. Modification is also easier because there is only one place, which needs to be changed.

Guideline 22: [required] The value of an expression shall be the same under all permitted evaluation orders.

Rationale: The C language standard does not specify the order in which the operands of an operator are evaluated (with some exceptions, like `&&, ||, ?: and ‘,’)`*. For example, in a statement like*:

```
x = f() + g();   // not allowed!
```

f may be evaluated before g or viceversa; thus if either f or g alters a variable on which the other depends, x can depend on the order of evaluation. To solve this issue, intermediate results can be stored in temporary variables to ensure a particular sequence:

```
temp = f();

x = temp + g();
```

Similarly, the order in which function arguments are evaluated is not specified, so the following code:

```
x = f(i++, i);  // not allowed!
```

can produce different results with different compilers.

Guideline 23: [advisory] Parentheses shall be used whenever they can clarify the order of evaluation.

Rationale: It is easy to make a mistake with the rather complicated precedence rules of C. Parentheses can be used to override default operator precedence, but can also be used to emphasise it. This approach helps to avoid errors and makes the code easier to read.

Example: In the following example the use of parentheses avoids an error (the intention of the programmer is to perform bit-testing on variable `statusWord`*).*

```
if (statusWord & MASK != 0)   // wrong! & has lower precedence than !=

if ((statusWord & MASK) != 0) // ok! bit-testing correctly performed
```

*Example: In the following example, the intention of the programmer is to compute a + (b*c); the use of parentheses does not change the result (in fact, the + operator has lower precedence than the * operator), however it makes the code easier to read as it eliminates possible confusion about what may have been intended.*

```
x = a + b * c;     // not allowed

x = a + (b * c);   // ok!
```

Guideline 24: [advisory] The basic types of char, int, short, long, float, double and long double should not be used, but *typedefs* that indicate size and signedness should instead be used.

Rationale: The storage length of types can vary from compiler to compiler. It is safer if programmers work with types which they know to be of a given length. For example, on a certain 32-bit machine, the following definitions might be suitable:

```
typedef signed char    int8_t;

typedef unsigned char  uint8_t;
typedef signed short   int16_t;

typedef unsigned short uint16_t;

typedef signed int     int32_t;

typedef unsigned int   uint32_t;

typedef signed long    int64_t;

typedef unsigned long  uint64_t;

typedef float          float32_t;

typedef double         float64_t;

typedef long double    float128_t;
```

Such specific-length types could be already provided by the library `<stdint.h>`, in that case those types should be used. Otherwise the known-length substitute types should be defined in a header file which can then be included in all code files.

Abstract types can be defined in terms of specific-length types, for example:

```
typedef uint32_t   mass_kg_t;
```

Guideline 25: [advisory] The use of pointers should be limited. When pointers are used, more than one level of de-referencing should be avoided. Pointer arithmetic should not be used. Function pointers should also be avoided.

Rationale: Pointers are easily misused, even by experienced programmers. They can make it hard to follow or analyse the flow of data in a program, especially by tool-based analysers. Concerning pointer arithmetic, the rule can be relaxed to allow simple pointer increments (`p_var++`) or decrements (`p_var--`).

Guideline 26: [advisory] Restrict all code to very simple control flow constructs. Avoid *goto* and *setjmp/longjmp* constructs. Avoid direct or indirect recursion.

Rationale: This rule is meant to improve the code understandability, both for human readers and for static code analyzers.

Guideline 27: [advisory] Executable statements, such as assignments, shall be avoided in conditional expressions.

Example:

```
/* NOT ALLOWED */
if ((new_value = get_value()) > THRESHOLD)
{
   …
}


/* OK */
new_value = get_value();
if (new_value > THRESHOLD)
{
   …
}
```

Guideline 28: [advisory] The body of an if … else, switch, while, do and for statement shall be enclosed by braces even if it is only one line long.

Example:

```
/* NOT ALLOWED */
if (int_value > MIN_VAL)
       res = fun(int_value);


/* OK */
if (int_value > MIN_VAL)
{
       res = fun(int_value);
}
```

Guideline 29: [advisory] Type conversions shall be done explicitly. Never rely on implicit type conversion.

Example:

```
float32_t float_var;

int32_t int_var;


int_var = float_var;              /* implicit conversion: to be avoided */

int_var = (int32_t) float_var;    /* explicit conversion: ok! */
```

Guideline 30: [advisory] The length of a function shall be kept as short as possible, according with the principle of functional cohesion.

Rationale: Excessively long functions are often a sign of poorly structured code. Each function should be a logical unit in the code that is understandable and verifiable as a unit. Some texts define the maximum function length as what can be printed on a single sheet of paper in a standard format with one line per statement and one line per declaration. Other texts propose a threshold of 75 lines of code per function for space software of criticality category D.

Guideline 31: [advisory] Code shall not be "commented out". If during development some code needs to be commented out, idioms like *#if DEBUG ... #endif* should be used. In any case, no disabled code shall remain in the production version.

Rationale: Commenting out code creates the risks of allowing unexpected snippets of code to be compiled into the final executable.

7.1.7 Naming

Guideline 32: [required] The identifiers used for functions, variables, constants and types shall be meaningful and self-explaining in the English language.

Rationale: For example, the name given to a function should be related to the action performed by that function. Note: the names of source files should also follow the principle of this guideline.

Guideline 33: [required] Identifiers from different scopes (name spaces, blocks inside a module, etc.) shall not have the same name.

Rationale: Hiding identifiers with an identifier of the same name in a nested scope leads to code which is very confusing. Example:

```
int16_t i;


{

        int16_t i; /* not allowed, same identifier used at different levels*/

        i = 3;     /* it could be confusing as to which i this refers */

}
```

7.1.8 Code Documentation

Guideline 34: [required] All source files (both header and program files) shall start with a prologue containing the following information: file name, project name, company name, author, edit history (e.g. date of creation, date of last change) and abstract (i.e. a short description of the functionality).

Guideline 35: [required] All functions shall be preceded by a prologue containing the following information: function name, function abstract, function parameters description, return value description, calling constraints, function algorithm description.

Guideline 36: [required] Comments shall be used on declarations and statements to improve the readability and maintainability of the code.

Guideline 37: [advisory] A documentation generator tool (e.g. Doxygen) shall be used.

Rationale: Such tools significantly reduce the effort for code documentation. The Doxygen tool extracts documentation from source files: the documentation is written within the code and is therefore easily kept up to date.

7.1.9 Compilation

Guideline 38: [required] For compilation of C source files with GCC, the compiler option –std=c89 shall be used.

Rationale: This option determines the language standard used by the compiler, in this case we want to use the ISO/IEC 9899:1990 standard.

Guideline 39: [required] All compiler warnings must be resolved, even if they are non-fatal. The GCC compiler option -Wall shall be used.

Rationale: The rule of zero warnings is intended to exploit the capability of the compiler to spot possible problems that often go unnoticed by the developers. This rule can be relaxed for compilation of third party libraries.

Guideline 40: [advisory] The code shall not be optimised at the expense of its readability. Optimisations shall be left to the compiler by setting the proper options.

Guideline 41: [advisory] Dynamic libraries shall not be used.

Rationale: TBW

7.1.10 Portability

Guideline 42: [required] Multithreaded programming shall use the POSIX threads library (pthreads).

Rationale: The RTEMS operating system supports the POSIX API.

7.2 Guidelines for C++

Guideline 43: [required] C++ code must be fully compliant with the ISO/IEC 14882:1998 standard (also known as C++98).

Rationale: The RTEMS C Cross Compiler available at the time of writing is based on GCC version 4.4.6. Such version of GCC fully supports the C++98 standard, while it is reported to have only experimental support for the C++11 standard.

Guideline 44: [required] For compilation of C++ source files with GCC, the compiler option –std=c++98 shall be used.

Rationale: This option determines the language standard used by the compiler, in this case we want to use the ISO/IEC 14882:1998 standard.

Guideline 45: [required] Class declarations shall contain the "public:", "protected:" and "private:" keywords exactly once.

Guideline 46: [required] Class member variables must not be declared "public".

Rationale: If the class member variables are hidden behind more widely available accessor functions, then the underlying implementation of the class can be hidden from the user of the class. The implementation may change without affecting the interface itself. This means that other areas of code do not need to change because the internal details of a class have changed.

Guideline 47: [required] A class should always declare a constructor.

Rationale: The "default" constructor, provided by the compiler if no other constructor is explicitly declared, initialises data members to zero. While this may work well for some classes, in general the programmer should always declare a constructor and explicitly initialise the member variables appropriately.

Guideline 48: [required] A class should always declare its destructor.

Rationale: The programmer should always declare a destructor in order to guarantee that memory is reclaimed correctly when an object is destroyed.

Guideline 49: [required] If a class contains any virtual member function then also its destructor must be virtual.

Guideline 50: [required] A class should always declare a copy constructor.

Guideline 51: [required] The layout of ordinary function declarations should be used for member functions, with modifications for constructors with initialisation lists.

Guideline 52: [required] Constructor functions which explicitly initialise any base class or member variable should not rely on a particular order of evaluation.

Guideline 53: [required] Objects should be constructed and initialised immediately if possible rather than be assigned after construction.

Guideline 54: [required] The class should always declare an assignment operator.

Guideline 55: [required] The assignment operator(s) must check for assigning an object to itself.

Guideline 56: [required] The assignment operator(s) must also assign base class member data.

Guideline 57: [required] The assignment operator(s) should return a reference to the object.

Guideline 58: [required] Symmetric operators, with the exception of assignment operator, should be defined as friend functions. All asymmetric operators (i.e. (), [], unary * and unary ->) must be defined as member functions.

Guideline 59: [required] Member functions, which do not alter the state of an object, shall be declared “const”.

Guideline 60: [required] Public member functions must not return non-const references or pointers to member variables of an object.

Rationale: The programmer must ensure that there are no ways of changing member variables without using the intended interface.

Guideline 61: [required] A function may not return a reference to memory, which it has allocated.

Guideline 62: [required] Member functions shall only be declared as “inline” if the need for optimisation has been identified.

Guideline 63: [required] A function shall not be declared as “inline” within the class declaration itself.

Guideline 64: [required] Templates should be encouraged as a convenient way of reusing code

Guideline 65: [required] If templates are used then Auto_ptr pointers should be preferred to normal pointers.

Guideline 66: [required] A function which can issue exceptions (i.e. has a "throw" clause in its declaration) can only be called: 1) from within a "try-catch" construct catching all those exceptions; 2) from within a "try-catch" construct catching some of those exceptions where the other are declared in the "throw" clause of the function containing the "try-catch" construct; 3) from within another function which exports (via the "throw" clause int its declaration) exactly the same exceptions.

Guideline 67: [required] Every C++ program using exceptions must use the function set_unexpected() to specify which user defined function must be called in case a function throws an exception not listed in its exception specification.

Guideline 68: [required] Every C++ program using exceptions must use the function set_terminate() to specify which user defined function must be called if an handler for an exception cannot be found.

Guideline 69: [required] The programmer should use the new C++ cast operators rather than the traditional C cast.

Guideline 70: [required] The programmer must not override the comma, &&, || and ?: operators.

Guideline 71: [required] Memory allocation using new/delete should be used instead of malloc()/free().

Guideline 72: [required] If the call to new uses [] then the corresponding call to delete must also use [].

Guideline 73: [required] Every C++ program using dynamic memory allocation must use the function set_new_handler() to specify which user defined function must be called in case of memory allocation failure.

Appendix 1: Glossary

Target System

The robotic system that will be used for the final mission. This includes also the RCOS under which the different modules run -OG1 in our case-.

Developer’s Environment

The setup that the designer of perception solutions uses to evaluate different approaches. Once a solution has been designed and implemented, it can be exported to the target system.

CDFF-Core

Part of the CDFF which contains the set of techniques for data fusion implemented in a modular fashion with a Data Fusion Common Interface (DFNCI) to allow high flexibility in configuration as well as in operation as a distributed system on multiple platforms and libraries dedicated to data filtering and outliers removal.

CDFF-Support

Part of the CDFF which Includes orchestration, data product management. This part of the CDFF is integrated in the final Target System.

***CDFF-Dev*[19]**

Not integrated in the target system. Encapsulates all the features of the CDFF which are used only in the designer environment (i.e. won’t be integrated in the target system).

Data Fusion Node (DFN)

Is a software library for data fusion which is defined in the CDFF-Core. These libraries are independent of the CDFF itself but as units might not provide a complete data fusion solution.

[19] Named CDFF-Utils in previous documents

Data fusion processing compound[20] (DFPC)

Libraries connected through a DFNCI by the CDFF. A DFPC is designed to provide a certain data product (e.g. pose estimation, map...). A same DFPC can be implemented with different programming languages and can be executed on multiple RCOS, though it must be adapted to the specificities of these. In Figure 28 a simplified DFPC to perform mapping is presented in the context of OG3. A DFPC is composed CDFF-Core Libraries and CDFF-Support DFPC. The DFPC implementation will differ depending on the target RCOS but the functionality should be the same.

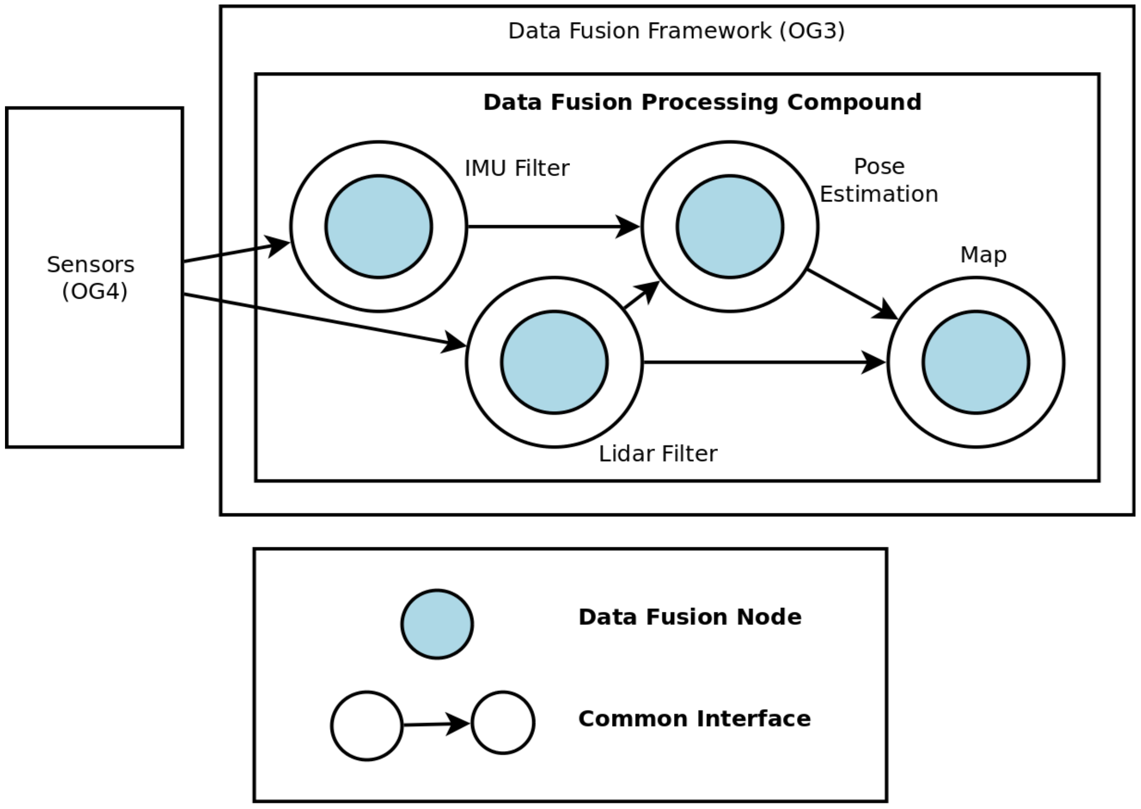


Figure 29: Example of a DFPC. A DFPC receives sensor data as inputs (provided by OG4) and produces Data Fusion Products (delivered to OG2 by the Orchestrator).

[20] Named Data Fusion Processing Chain in previous documents

Data Product Management tool (DPM)

The series of processes that select, structure and store the raw data, intermediary data products and final data products acquired and generated within InFuse.

Operating System (OS)

Software that provides basic computing system functions and an interface to hardware, a Real-Time Operating System (RTOS) implies hard real-time operation.

RCOS or Robotics Middleware

Software framework that facilitates the integration of multiple libraries, supports communications, enables configurations and provides control over them. Furthermore, the RCOS provides control over operating system of the controlled system and to its sensors and actuators.

Appendix 2: InFuse CDFF Product Tree

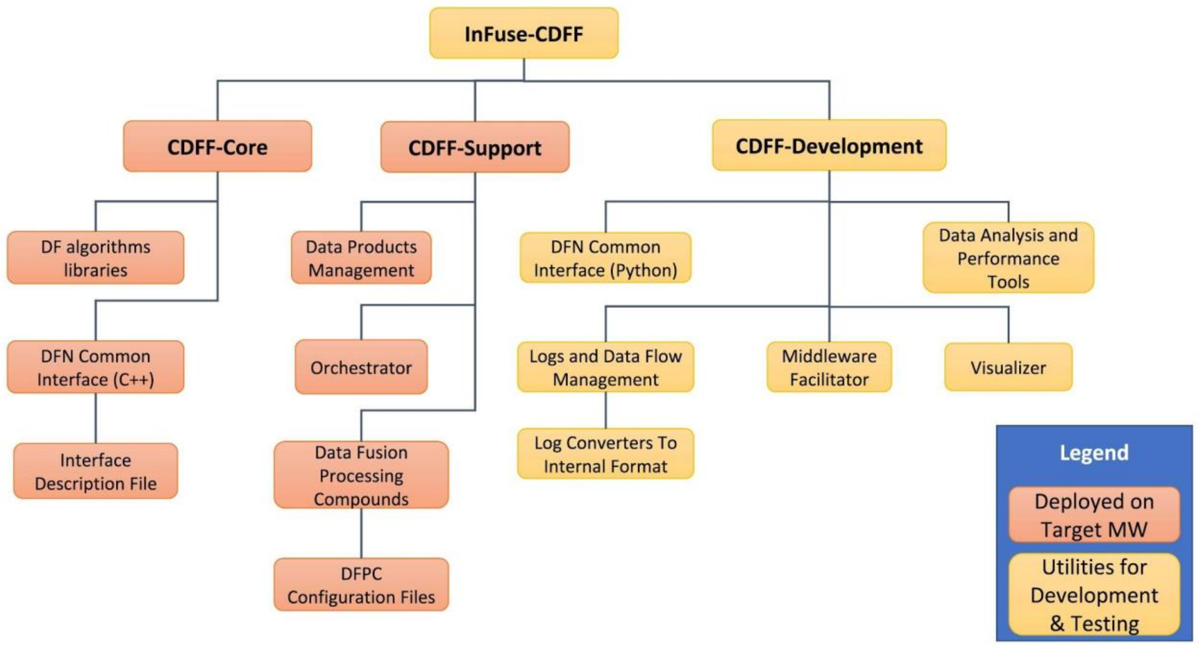

InFuse-CDFF
CDFF-Core
CDFF-Support
CDFF-Development
DF algorithms libraries
DFN Common Interface (C++)
Interface Description File
Data Products Management
Orchestrator
Data Fusion Processing Compounds
DFPC Configuration Files
DFN Common Interface (Python)
Data Analysis and Performance Tools
Logs and Data Flow Management
Middleware Facilitator
Visualizer
Log Converters To Internal Format
Legend
Deployed on Target MW
Utilities for Development & Testing

Figure 30: Product Tree of InFuse Common Data Fusion Framework

Appendix 3: Libraries and Data Types

To implement the different DFNs and support components a list of libraries which the consortium is considering using is provided.

Library name	Description	Link
Open CV	Library for Image Processing.	http://opencv.org/
PCL	Library for 3D pointclouds processing	http://pointclouds.org/
OctoMap	3D maps library	https://octomap.github.io/
ESROCOS Base Types	Types defined in ASN1 which OG1 supports	https://github.com/ESROCOS/types-base
Base Types	Library for types	https://github.com/rock-core/base-types
Slam/Maps	Maps Types	https://github.com/envire/slam-maps
G2O	Graph Optimization Framework	https://github.com/RainerKuemmerle/g2o
URDF	Robot Model Library	https://github.com/ros/robot_model
EnviRe	Libraries for Environment Representation and Visualization	http://envire.github.io/
Vizkit 3D	Visualization Plugins	https://github.com/rock-gui/gui-vizkit_3d_plugins
Eigen 3	Library for linear algebra: matrices, vectors, numerical solvers and related algorithms	http://eigen.tuxfamily.org
LIBSVM	A library for support vector machines	https://www.csie.ntu.edu.tw/~cjlin/libsvm/
Aquila, sigpack, DSPFilters	Libraries for signal processing which will be studied	http://aquila-dsp.org, http://sigpack.sourceforge.net, https://github.com/vinniefalco/DSPFilters

Table 8: Tentative list of libraries and type definitions that will be used in InFuse

Appendix 4: Requirements Traceability Matrix

The requirements were organized in the System Requirements Document (deliverable 3.2) according to a previous categorization of the features. In this appendix the requirements remain organized as previously to facilitate tracking with the previous document.

CDFF Core

User Requirements (UserR)

Req. ID	Compliance (yes / no / partial)	Section where it is addressed	Comments
SR_UserR_A101	Yes	4.1.2.1	
SR_UserR_A102	Yes	4.1.2	
SR_UserR_A103	Yes	4.1.2	
SR_UserR_A104	Yes	4.1.2	
SR_UserR_A105	Yes	4.1.2	
SR_UserR_A106	Yes	4.1.2	
SR_UserR_A107	Yes	4.1.2	
SR_UserR_A108	Yes	4.1.2.4	
SR_UserR_A109	Yes	4.1.2	
SR_UserR_A110	Yes	4.1.2	
SR_UserR_A111	Yes	4.1	
SR_UserR_A112	Yes	4.2	

SR_UserR_A113	Yes	4.1.2	
SR_UserR_A114	Yes		
SR_UserR_A115	Yes	4.2.2	
SR_UserR_A116	Partial. Visualization is only provided on the Developer's Environment	4.3.5	
SR_UserR_A117	Yes	4.1.	

Functional Requirements (FuncR)

Req. ID	**Compliance** **(yes / no / partial)**	**Section where it is addressed**	**Comments**
SR_FuncR_A201	yes, not explicitly addressed	4.1.2	
SR_FuncR_A202	yes, not explicitly addressed	4.1.2	
SR_FuncR_A203	yes, not explicitly addressed	4.1.2	
Req. ID	Compliance (yes / no / partial)	Section where it is addressed	Comments
SR_FuncR_A204	yes, not explicitly addressed	4.1.2	
SR_FuncR_A205	Partial	4.1.2	Question on OG6 / GNC with the orbital

			track => answer from PSA send by PSA on Thursday, 23 March, 2017 7:31:46 PM
SR_FuncR_A206	yes, not explicitly addressed	4.1.2	
SR_FuncR_A207	yes, not explicitly addressed	4.1.2	
SR_FuncR_A208	yes, not explicitly addressed	4.1.2	
SR_FuncR_A209	yes, not explicitly addressed	4.1.2	
SR_FuncR_A210	yes, not explicitly addressed	4.1.2	
SR_FuncR_A211	yes, not explicitly addressed	4.1.2	
SR_FuncR_A212	yes, not explicitly addressed	4.1.2	
SR_FuncR_A213	yes, not explicitly addressed	4.1.2	
SR_FuncR_A214	yes, not explicitly addressed	4.1.2	
SR_FuncR_A215	Yes	5.5.1	
SR_FuncR_A216	Yes	5.5.1	
SR_FuncR_A217	Yes	5.5.1	
SR_FuncR_A218	Yes	5.5.1	

SR_FuncR_A219	Yes	5.5.1	
SR_FuncR_A220	Yes	5.5.1	
SR_FuncR_A221	Yes, not explicitly addressed		
SR_FuncR_A222	Yes	5.5.1	
SR_FuncR_A223	Yes	4.1.2.4	
SR_FuncR_A224	Yes	4.1.2.3	
SR_FuncR_A225	Yes	4.1.2.2	
SR_FuncR_A226	Yes	4.1.2.3	
SR_FuncR_A227	Yes, not explicitly addressed		
SR_FuncR_A228	Partially		The InFuse architecture and functions will be able to support such scenarios, but will be partially demonstrated, only within the InFuse consortium.

Performance Requirements (PerfR)

Req. ID	**Compliance** (yes / no / partial)	**Section where it is addressed**	**Comments**

SR_PerfR_A301	Yes	3.3, 4.2.1., 4.2.2	
SR_PerfR_A302	Yes	3.3, 4.2.1., 4.2.2	
SR_PerfR_A303	Yes	3.3, 4.2.1., 4.2.2	
SR_PerfR_A304	Yes	3.3, 4.2.1., 4.2.2	
SR_PerfR_A305	Yes	3.3, 4.2.1., 4.2.2	
SR_PerfR_A306	Yes	3.3, 4.2.1., 4.2.2	
SR_PerfR_A307	Yes	3.3, 4.2.1., 4.2.2	
SR_PerfR_A308	Yes	3.3, 4.2.1., 4.2.2	

Interface Requirements (IntR)

Req. ID	**Compliance** **(yes / no / partial)**	**Section where it is addressed**	**Comments**
SR_IntR_A403	Yes	4.1.1, (5.3)	
SR_IntR_A404	Yes	4.1.1, (5.5.1)	
SR_IntR_A405	Yes	4.1.1.1	
SR_IntR_A406	Yes	4.2.3	
SR_IntR_A407	Yes	4.2.3	
SR_IntR_A408	Yes	4.2.3	

SR_IntR_A409	Yes	4.2.3	

Resource Requirements (ResR)

Req. ID	Compliance (yes / no / partial)	Section where it is addressed	Comments
SR_ResR_A501	Yes	4.1.1	
SR_ResR_A502	Yes, not explicitly addressed		
SR_ResR_A503	Yes	4.1.2	
SR_ResR_A504	Yes	4.1.2	
SR_ResR_A505	Yes	4.1.2	

Operational Requirements (OpR)

Req. ID	Compliance (yes / no / partial)	Section where it is addressed	Comments
SR_OpR_A601	Yes	4.2.2	

SR_OpR_A602	partial		OG1 related. Inspection utilities should be provided by the RCOS
SR_OpR_A603	Yes	4.2.2	
SR_OpR_A604	partial, via configuration		

Product assurance and safety requirements (ProdR)

Not Applicable.

Configuration and implementation requirements (ConfR)

Req. ID	Compliance (yes / no / partial)	Section where it is addressed	Comments
SR_ConfR_A801	Yes	4.1.1, 5.4	
SR_ConfR_A802	Yes	4.1.1, 5.5	
SR_ConfR_A803	Yes	?	
SR_ConfR_A804	Yes	?	
SR_ConfR_A805	Yes	?	
SR_ConfR_A806	Yes	?	
SR_ConfR_A807	Partial,	5.5	depends on RCOS

Test and Validation (ValR)

Req. ID	Compliance (yes / no / partial)	Section where it is addressed	Comments
SR_ValR_A901	Yes	-	
SR_ValR_A902	Yes	-	
SR_ValR_A903	Yes	In Related Document D4.1	TBD?
SR_ValR_A921	Yes	-	
SR_ValR_A922	Yes	In Related Document D4.1	
SR_ValR_A923	Yes	-	
SR_ValR_A924	Yes	-	TBD
SR_ValR_A951	Yes	In Related Document D4.1	
SR_ValR_A961	Yes	In Document D4.1	

CDFF Dev

User Requirements (UserR)

Req. ID	Compliance (yes / no / partial)	Section where it is addressed	Comments
SR_UserR_B101	Yes	4.3.3	

SR_UserR_B102	Yes	4.3.3	
SR_UserR_B103	Yes	4.3.5	
SR_UserR_B104	Yes	4.3.5	
SR_UserR_B105	Partial,	-	OG1 related. Part of the RCOS, depends on the target platform
SR_UserR_B106	Yes	4.1.2	
SR_UserR_B107	Yes	4	

Functional Requirements (FuncR)

Req. ID	**Compliance** **(yes / no / partial)**	**Section where it is addressed**	**Comments**
SR_FuncR_B201	Yes	4.3.3	
SR_FuncR_B202	Yes	4.3	
SR_FuncR_B203	Yes	4.3.5	
SR_FuncR_B204	Yes	4.3.4	The performance analysis can be checked better on the target system with all the frameworks integrated
SR_FuncR_B205	Yes	5.5	
SR_FuncR_B206	Yes	4.2 and 4.3	

SR_FuncR_B207	Yes	4.2.2 and 5.4.1	
SR_FuncR_B208	Yes	4.1.2	
SR_FuncR_B209	Yes	4.3.5	The visualizations are only possible in the Developers Environment. On the target system they should be supported by the RCOS.
SR_FuncR_B210	Yes	4.3.2	
SR_FuncR_B211	Yes	4.2.2	

Performance Requirements (PerfR)

NA.

Interface Requirements (IntR)

Req. ID	**Compliance (yes / no / partial)**	**Section where it is addressed**	**Comments**
SR_IntR_B401	Yes	5.2	

Resource Requirements (ResR)

Req. ID	**Compliance (yes / no / partial)**	**Section where it is addressed**	**Comments**

SR_ResR_B501	Yes	4.1, 5.3	
SR_ResR_B502	Yes, but depends on DFPC	4.2	
SR_ResR_B503	Yes, not explicitly addressed	-	

Operational Requirements (OpR)

NA.

Product assurance and safety requirements (ProdR)

NA.

Configuration and implementation requirements (ConfR)

Req. ID	**Compliance** **(yes / no / partial)**	**Section where it is addressed**	**Comments**
SR_ConfR_B801	Yes	4.1.1, 4.2.1	

Appendix 5: List of Sensors available in the InFuse Consortium

This is a preliminary list of sensors available within the consortium, that should serve for internal testing purpose.

Sensors	SpaceApps	DLR	DFKI	MAG	USTRAT	CNRS-LAAS
Stereo camera	Point Grey Bumblebee XB3	Guppy Pro F-125 B + Schneider/ Kreuznach Cinegon 1.8/4.8		[various] + baseline-variable stereo bench	Zed camera	Stereoscopic bench (2 Marlin Firewire cameras, Schneider Cinegon 4.05 mm lenses)
RGB camera	uEye XS, Point Grey Bumblebee XB3	Guppy Pro F-125 C + Schneider/ Kreuznach Cinegon 1.8/4.8		[various]		
Structured light camera	Kinect v2 / Asus Xtion	LRU rear cam + proj		Kinect v2 / Asus Xtion / …		
Thermal camera		LRU ScienceCa m				
ToF camera						
2D lidar	Hokuyo URG-04, SICK LMS 151			RPLidar Laser Scanner	Hokuyo scanning laser range finder	
3D lidar	SICK LMS 151 (with FLIR PTU D46_17)					Velodyne HDL-32; Sick LMD-MRS 40001S01 (4 planes, produces

						reflectance information)
IMU (MEMS)	XSens MTI-28A53G35	XSens MTi-10		XSens Mit / Sparkfun Razor9DOF IMU	MPU-9150	XSens MTi-111
IMU (fiber optics)						Fiber optic gyro (single axis): KVH 5000
Rover wheel odometry	Proprietary [Husky]	LRU				Odometry: wheel encoders
Robot arm encoders	Directed Perception D46-17	[PTU]			Optical encoder, Bourns 5V dc 256 Pulse Optical Encoder, and 64 pulse.	PTU: Directed Perception PTU-D46 model
Force/torque sensors	Invenscience arm 2	LRU			Force sensors, FlexiForce A301	
Sun tracker	Fusion Selection DS-50 S6 (solar tracker for a 2-axis controller) - not a real sun tracker					
Star tracker						
Radar						

Ultrasonic sensor						
Magnetometer						
Hyper/multi spectral camera	Ximea Linescan LS100-NIR	LRU ScienceCam		[various] + baseline-variable stereo bench	Zed camera	Ximea Linescan LS100-NIR