COMP4DRONES

DELIVERABLE

D3.3 – Implementation of Integrated and Modular Architecture for Drones first version

Project Title	COMP4DRONES
Grant Agreement number	826610
Call and topic identifier	H2020-ECSEL-2018
Funding Scheme	Research & Innovation Action (RIA)
Project duration	36 Months [1 October 2019 – 30 September 2022]
Coordinator	Mr. Mauro Gil Cabeza (INDRA)
Website	www.comp4drones.eu

ECSEL Joint Undertaking
Electronic Components and Systems for European Leadership

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 826610

Document fiche		
Authors:	**Partner Name**	**Contributors**
	IMEC-BG	Murali Jayapala, Ljiljana Platisa
	BUT	Svetozar Nosko
	UWB	Ondřej Severa, Lukáš Bláha
	ENAC	Gautier Hattenberger, Fabien Bonneval
	SCALIAN	Raphaël Lallement
	ENSMA	Matheus Ladeira, Emmanuel Grolleau, Yassine Ouhammou, Henri Bauer, Patrick Coirault, Syed Ali Ajwal
	SIEMENS	Olivier Broca
	UNIMORE	Alessandro Capotondi, Gianluca Bellocchi, Andrea Marongiu
	UNISS	Daniel Madroñal, Francesca Palumbo
	UNIVAQ	-
	EDI	Rihard Novickis
	ACORDE	Fernando Herrera
	HIB	Diego Fuentes
	IKERLAN	Iñigo Muguruza
	CEA	Ansgar Radermacher, Mahmoud Hussein, Reda Nouacer Matteo Morelli
	AI	Stefano Delucchi
	UDANET	Maurizio Preziuso
	MODIS	Daniela Parletta
	IFAT	Dominic Pirker, Rainer Matischek
Internal reviewers:	Guillaume Thalmann (ALTRAN), Federico Corradi (IMEC-NL) Reda Nouacer (CEA)	
Work Package:	WP3	
Task:	T3.2	
Nature:	R	
Dissemination:	PU	

Document History			
Version	**Date**	**Contributor(s)**	**Description**
V0.1	03/02/2021	Mahmoud Hussein Reda Nouacer	Initial table of content
V0.2	16/06/2021	Matheus Ladeira Emmanuel Grolleau	Second version of the ToC
V0.3	21/07/2021	Matheus Ladeira Emmanuel Grolleau Richard Novickis	Minor updates to the ToC, added EDI's contribution
V0.4	26/08/2021	Matheus Ladeira Emmanuel Grolleau Richard Novickis	Added introductive material

		Fernando Herrera Mahmoud Hussein	
V0.5	01/09/2021	Gautier Hattenberger Fabien Bonneval	Added ABI bus and comp. WP3-13
V0.6	02/09/2021	Syed Ali Ajwad Patrick Coirault	Added two enabling technologies
V0.7	06/09/2021	Gautier Hattenberger	Added an enabling technology
V0.8	09/09/2021	Ansgar Radermacher	Added presentation of ROS, ROS2 and uORB
V0.9	13/09/2021	Daniel Madroñal Francesca Palumbo Dominic Pirker Rainer Matischek	Added two enabling technologies
V0.10	15/09/2021	Iñigo Muguruza	Added comp. WP3-01
V0.11	17/09/2021	Alessandro Capotondi Fernando Herrera	Added comps. WP3-15_1 and -22
V0.12	22/09/2021	Diego Fuentes Fernando Herrera	Added comps. WP3-04 and WP3-15_2
V0.13	23/09/2021	Fernando Herrera	Added a UML MARTE section
V0.14	27/09/2021	Rihard Novickis Raphaël Lallement	Added comps. WP3-02 and WP3-04
V0.15	28/09/2021	Svetozar Nosko Matheus Ladeira Emmanuel Grolleau	Added comp. WP3-03 Added conclusion
V0.16	29/09/2021	Matheus Ladeira Emmanuel Grolleau	Integrated some minor form corrections, references, index
V0.17	30/09/2021	Emmanuel Grolleau Ondřej Severa Lukáš Bláha Fernando Herrera Ansgar Radermacher Daniela Parletta Fernando Herrera Olivier Broca Stefano Delucchi	Corrections to Section 5, added Enabling technologies
V0.18	1/10/2021	Emmanuel Grolleau Fernando Herrera Reda Nouacer Matheus Ladeira Murali Jayapala Ljiljana Platisa Maurizio Preziuso	Integration of internal review comments and corrections, added components WP3_36_1&2, WP3_19_1&2
V0.19	3/10/2021	Emmanuel Grolleau Reda Nouacer Ansgar Radermacher	Integrated internal reviewer's comments Added ontology details
V0.5	01/09/2021	Gautier Hattenberger Fabien Bonneval	Added ABI bus and comp. WP3-13

Keywords:	Reference Architecture, Implementation, Enabling technologies, Flight Management, Flight Control, Flight Navigation, Mission Management, Payloads, Tools, and Hardware platforms
Abstract (few lines):	This deliverable describes the specifications and the first steps to the **implementation** of COMP4DRONES **reference architecture.** Rather than providing yet another autopilot or yet another meta-model, the concept is to exhibit how existing frameworks are to include specificities regarding autopilot implementations. This document opens to different framework semantics augmentation, targeting a larger adoption. One of the main focuses of the chosen concepts is to allow a fine grain modelling of existing autopilots, and especially off-the-shelf autopilots for (1) easy integration of custom components, and (2) a fine grain representation allowing functional and non-functional properties to be checked.

DISCLAIMER

ACKNOWLEDGEMENT

This document is a deliverable of COMP4DRONES project. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement Nº 826610

Table of Contents

Table of Figures

Table of Tables

Definitions, Acronyms and Abbreviations

Acronym	Title
AADL	Architecture Analysis and Design Language
AAXL	XML formalism of AADL
ABI	AirBorne Interface
ADC	Analog Digital Converter
ADL	Architecture Description Language
AFNOR	Association Française de Normalization (French Standardisation Association)
AHRS	Attitude and Heading Reference Systems
AI	Artificial Inteligence
AIAA	American Institute of Aeronautics and Astronautics
AMCL	Adaptive Monte Carlo Localization
AMP	Asymmetric Multiprocessing
APDU	Application Protocol Data Unit
APEX	Application Executive
APF	Artificial Potential Function
API	Application Programming Interface
APM	ArduPilot Mega
ARC	Air Risk Class
ARCADIA	Architecture Analysis and Design Integrated Approach
ARINC	Aeronautical Radio Incorporated
ARM	Advanced RISC Machine
ARXML	AUTOSAR XML
ASTM	American Society for Testing and Materials
ATSA-DRA	Asynchronous Tag trigger, Slotted Anchor response with Deterministic and Random Allocation
AUTOSAR	Automotive Open System Architecture
BIM	Building Information Modelling
BSP	Board Support Package
C2LINK	Command and Control Link
C4D	COMP4DRONES Project
CAN	Controller Area Network
CCD	Charge Coupled Device
CFA	Color Filter Array
CGRA	Coarse-Grained Reconfigurable HW Accelerators
CMA	Contiguous Memory Allocator
CMOS	Complementary Metal Oxide Semiconductor
CNN	Convolutional Neural Network
COTS	Commercial Off the Shelf
CPU	Central Processing Unit
CSI	Camera Serial Interface
DAA	Detect and Avoid
DAL	Design Assurance Level
DCM	Direction Cosine Matrix
DDR	Double Data Rate
DDS	Data Distribution Service
DMA	Direct Memory Access
DOI	Digital Object Identifier

DOP	Dilution of Precision
DP	Droneport
DSL	Domain Specific Language
DTB	Device Tree Blob
ECSEL	Electronic Components and Systems for European Leadership
ECU	Electronic Control Unit
ELF	Executable Linked Format
eMMC	Embedded Multimedia Card
FAA	Federal Aviation Administration
FIFO	First In, First Out
FP	Fixed Point
FPGA	Field Programmable Gate Array
FPV	First Person View
GCM	Generic Component Model
GCS	Ground Control Station
GLAD	GNSS-based Low-Cost position and Attitude Determination system
GNSS	Global Navigation Satellite System
GPIO	General Purpose Input/Output
GPS	Global Positioning System
GPU	Graphics Processing Unit
GQAM	Generic Quantitative Analysis Modelling
GRC	Ground Risk Class
GTSDB	German Traffic Sign Detection Benchmark
GUI	Graphical User Interface
H2020	Horizons 2020
HCL	HDL Components Library
HDL	Hardware Description Language
HDR	High Dynamic Range
HLS	High Level Synthesis
HRM	Hardware Resource Modelling
HSM	Hardware Security Module
HSoC	Heterogeneous System-on-Chip
HW	Hardware
I/O	Input / Output
I2C	Inter-Integrated Circuit
ID	Identifier
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IMU	Inertial Measurement Unit
INCOSE	International Council on Systems Engineering
INDI	Incremental Nonlinear Dynamic Inversion
INS	Inertial Navigation System
IP	Internet Protocol
IPC	Inter Process Communication
IPS	Indoor Positioning system
ISO	International Organization for Standardization
ISR	Interrupt Service Routine
JARUS	Joint Authorities for Rulemaking on Unmanned Systems
LCM	Least Common Multiple
LDR	Low Dynamic Range

LIDAR	Light Detection and Ranging
LVDS	Low-Voltage Differential Signalling
MAC	Medium Access Control
MARTE	Modeling and Analysis of Real-Time and Embedded systems
MAS	Multi-Agent Systems
MBSE	Model-Based System Engineering
MDA	Model Driven Architecture
MDG	Multi-Dataflow Generator
MISRA	Motor Industry Software Reliability Association
MMC	Multimedia (Card)
MMU	Memory Management Unit
MPSoC	MultiProcessor System on a Chip
MPU	MicroProcessing Unit
MTSD	Mapillary Traffic Sign Dataset
NFP	Non-Functional Properties
OMG	Object Management Group
OS	Operating System
OSEK	Open Systems and their Interfaces for the Electronics in Motor Vehicles
OSRF	Open Source Robotics Foundation
PC	Platform Composer
PFD	Probability of dangerous failure
PID	Proportional Integral Derivative
POI	Point of Interest
POSIX	Portable Operating System Interface
PPRZLINK	Paparazzi Link
PUB-SUB	Publish-Subscribe
PWM	Pulse Width Modulation
QSPI	Queued Serial Peripheral Interface
RAM	Random Access Memory
REQ-REP	Request-Response
RGB	Red Green Blue
ROS	Robot Operating System
RP1, RP2...	Reporting Period 1, 2...
RPN	Region Proposal Network
RT	Real Time
RTL	Register Transfer Level
RTOS	Real Time Operating System
RTPS	Real Time Publish Subscribe
SAE	Society of Automotive Engineers
SAIL	Specific Assurance and Integrity Level
SD	Secure Digital
SE	Secure Element
SGeT	Standardization Group for Embedded Technologies
SIL	Software Insurance Level
SLAM	Simultaneous Localization And Mapping
SMARC	Smart Mobility Architecture
SMP	Symmetric Multiprocessing
SoC	System on a Chip
SORA	Specific Operations Risk Assessment
SPC	Semaphore Precedence Constraints

SPI	Serial Peripheral Interface
SRM	Software Resource Modelling
SSH	Secure Shell
SW	Software
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TMO	tone mapping operator
TRL	Technology Readiness Level
TRNSYS	Transient System Simulation Tool
UART	Universal Asynchronous Receiver Transmitter
UAS	Unmanned Aerial Systems
UAV	Unmanned Aerial Vehicles
UAVCAN	Uncomplicated Application-level Vehicular Computing And Networking
UC	Use Case
UDP	User Datagram Protocol
UML	Unified Modeling Language
uORB	micro Object Request Broker
USB	Universal Serial Bus
UTM	UAV Traffic Management
UTP2	UML Testing Profile 2
UWB	Ultra-wideband
WP1, WP2…	Work Package 1, 2, etc.
XML	Extensible Markup Language
ZCU102	Xilinx Ultrascale+ MPSoC
ZU15EG	Zynq UltraScale+ module

Executive Summary

Drones or Unmanned Aerial Vehicles (UAVs) can perform air operations that manned aircrafts struggle with. Their use brings significant economic savings and environmental benefits whilst reducing the risk to human life. However, current drone embedded architectures are organized in loosely coupled monolithic boards, each composed of processor, memory, and communication resources (e.g., flight control, planning, and vision boards), which in turn does not support the continuous development of drone applications such as the ever-increasing demand on autonomy. Drone embedded platforms must meet this demand, while still providing safety, robustness and a small footprint in physical size and power consumption.

In this deliverable, we provide the first version of **COMP4DRONES reference architecture** implementation that enables the **easy integration** and **customization** of qualified systems embedded in drones. Indeed, D3.2 has delivered the specification (i.e. a vision) of COMP4DRONES reference architecture following a bottom-up approach starting from existing well-known open autopilots (academic or industrial). The deliverable has followed a taxonomical based approach by identifying the building blocks and design patterns for UAV architectures. In this deliverable, model-based design and development to support the instantiation of the reference architecture is introduced. In addition, the design and implementation of a set of enabling technologies (building blocks or generic HW/SW components) are described, complementing the achievements reported in D3.1 and D3.2.

1 Introduction

1.1 Purpose

For drone stakeholders, several problems arise from the fact that autopilots have complex implementations: the added value in drones is usually not in the autopilot itself, but rather in the **customization** of the autopilot by the adjunction of custom software or hardware components. Integrating a new feature (i.e. Hardware or Software component) in an existing off-the-shelf autopilot requires a very high degree of **combined expertise**, a lot of time and effort. This kind of manual developing is error prone inducing a high risk of failure, since if not integrated and checked carefully, a custom component may harm the normal behaviour of the autopilot.

After observing that many languages, frameworks, and tools target generic embedded real-time system, the choice is to focus on the specifics of drone autopilots, in order to leverage existing frameworks and tools for them to be of use in the context of drones.

By using model of off-the-shelf or particular autopilot in frameworks leveraged with their missing semantics, stakeholders will be able to easily integrate their custom components and check functional and non-functional properties.

After justifying the approach, and presenting the related concepts and technologies, the main contributions are developed in Section 5, which presents concepts missing in most Architecture Description Languages (ADL) that should be leveraged to better capture autopilots implementations, and in Section 6 which is presenting the advances in the Enabling Technologies developed by the C4D partners.

1.2 Scope

- Section 2 is a recall of Reference Architecture specifications from previous deliverables;
- Section 3 provides background regarding standards, methodologies, and technologies;
- Section 0 explores the existing architecture implementations in open-source autopilots;
- Section 5 presents the concepts behind this specific reference architecture, as well as its first implementation;
- Section 0 presents the implementation of each generic component in the given DSL;
- Section 7 concludes the document;
- D3.4 (end of the project) will present how this implementation leveraged several frameworks, and will illustrate how some existing off-the-shelf autopilots can be easily customized, and see some of their functional and non-functional properties checked.

1.3 References

[1] ‘Welcome To UML Web Site!’ https://www.uml.org/ (accessed Sep. 29, 2021).

[2] ‘Uml Profile Category - Specifications associated’. https://www.omg.org/spec/category/uml-profile/ (accessed Sep. 29, 2021).

[3] ‘OMG MARTE Web site’. https://www.omg.org/omgmarte/ (accessed Sep. 29, 2021).

[4] A. Koudri, D. Aulagnier, D. Vojtisek, P. Soulard, and C. Moy, ‘Using marte in a co-design methodology’, 2008, Accessed: Sep. 29, 2021. [Online]. Available: https://hal-supelec.archives-ouvertes.fr/hal-00354356.

[5] C. Mraidha, S. Tucci-Piergiovanni, and S. Gerard, ‘Optimum: A MARTE-based Methodology for Schedulability Analysis at Early Design Stages’, *ACM SIGSOFT Softw. Eng. Notes*, vol. 36, no. 1, pp. 1–8, 2011, doi: 10.1145/1921532.1921555.

[6] ‘S3D – Model-Driven Analysis and Design Framework’. https://s3d.unican.es/ (accessed Sep. 29, 2021).

[7] 'HepsyCode'. https://www.hepsycode.com/ (accessed Sep. 29, 2021).

[8] H. Posadas, P. Peñil, A. Nicolás, and E. Villar, 'Automatic synthesis of communication and concurrency for exploring component-based system implementations considering UML channel semantics', *J. Syst. Archit.*, vol. 61, no. 8, pp. 341–360, Sep. 2015, doi: 10.1016/J.SYSARC.2015.07.002.

[9] P. Roques, *Systems Architecture Modeling with the Arcadia Method. A Practical Guide to Capella*, vol. 53, no. 9. iSTE Press, 2018.

[10] A. Burns, 'The Ravenscar Profile', *ACM SIGAda Ada Lett.*, vol. XIX, no. 4, pp. 49–52, Dec. 1999, doi: 10.1145/340396.340450.

[11] F. BUSCHMANN, K. HENNEY, and D. C. SCHMIDT, *Pattern-oriented software architecture, on patterns and pattern languages*. John wiley & sons, 2007.

[12] M. Faugère, T. Bourbeau, R. De Simone, and S. Gérard, 'MARTE: Also an UML profile for modeling AADL applications', in *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems, ICECCS*, 2007, pp. 359–364, doi: 10.1109/ICECCS.2007.29.

[13] L. Yang, K. Cormican, and M. Yu, 'Ontology-based systems engineering: A state-of-the-art review', *Comput. Ind.*, vol. 111, pp. 148–171, Oct. 2019, doi: 10.1016/j.compind.2019.05.003.

[14] T. D. Nguyen, Y. Ouhammou, E. Grolleau, J. Forget, C. Pagetti, and P. Richard, 'Design and analysis of semaphore precedence constraints: A model-based approach for deterministic communications', in *Proceedings of the 2018 Design, Automation and Test in Europe Conference and Exhibition, DATE 2018*, Apr. 2018, vol. 2018-Janua, pp. 231–236, doi: 10.23919/DATE.2018.8342008.

[15] R. Novickis and M. Greitans, 'FPGA Master based on chip communications architecture for Cyclone v SoC running Linux', in *2018 5th International Conference on Control, Decision and Information Technologies, CoDIT 2018*, 2018, pp. 403–408, doi: 10.1109/CoDIT.2018.8394842.

[16] 'Heterogeneous System Architecture Foundation'. https://hsafoundation.com/ (accessed Sep. 29, 2021).

[17] 'The Contiguous Memory Allocator [LWN.net]'. https://lwn.net/Articles/396657/ (accessed Sep. 29, 2021).

[18] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2*. John Wiley & Sons, 2000.

[19] H. Shakhatreh *et al.*, 'Unmanned Aerial Vehicles (UAVs): A Survey on Civil Applications and Key Research Challenges', *IEEE Access*, vol. 7. pp. 48572–48634, 2019, doi: 10.1109/ACCESS.2019.2909530.

[20] 'German Traffic Sign Benchmarks'. https://benchmark.ini.rub.de/gtsdb_about.html (accessed Sep. 29, 2021).

[21] 'Mapillary'. https://www.mapillary.com/dataset/trafficsign (accessed Sep. 29, 2021).

[22] S. Ren, K. He, R. Girshick, and J. Sun, 'Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks', *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 6, pp. 1137–1149, Jun. 2017, doi: 10.1109/TPAMI.2016.2577031.

[23] R. Girshick, 'Fast R-CNN', in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1440–1448, Accessed: Sep. 29, 2021. [Online]. Available: https://github.com/rbgirshick/.

[24] C4DConsortium, 'C4D Deliverable D3.2 of the JU/ECSEL COMP4DRONES project. "Specification of Integrated and Modular Architecture for Drones", July', 2021.

[25] A. Boukara and & A. Naamane, 'Journal of Renewable Energy and Sustainable Development (RESD) A Brief Introduction to Building Information Modeling (BIM) and its interoperability with TRNSYS', *Renew. Energy Sustain. Dev.*, no. June, pp. 126–130, 2015, Accessed: Sep. 29, 2021. [Online]. Available: http://apc.aast.edu.

[26] C4DConsortium, 'Deliverable D1.1 of the JU/ECSEL COMP4DRONES project. "Specification of Industrial Use Cases" Specification of Industrial Use Cases". Version 2.1, March 30th', 2021.

[27] C4DConsortium, 'Deliverable D1.2 of the JU/ECSEL COMP4DRONES project. "System under test requirements". Version 3.3. April 9th', 2021.

[28] C4DConsortium, ‘Deliverable D3.1 of the JU/ECSEL COMP4DRONES project. "Specification of Integrated and Modular Architecture for Drones”. July 31th’, 2020.

[29] ‘Introduction · MAVLink Developer Guide’. https://mavlink.io/en/ (accessed Sep. 29, 2021).

[30] C4DConsortium, ‘Deliverable D5.3 of the JU/ECSEL COMP4DRONES project. "Robust multi-radio communication ”. June, 26th’, 2021.

[31] ‘Ubuntu Manpage: cyclictest - High resolution test program’. https://manpages.ubuntu.com/manpages/cosmic/man8/cyclictest.8.html (accessed Sep. 29, 2021).

[32] ‘Ubuntu Manpage: hackbench - scheduler benchmark/stress test’. https://manpages.ubuntu.com/manpages/bionic/man8/hackbench.8.html (accessed Sep. 29, 2021).

[33] C4DConsortium, ‘Deliverable D6.2 of the JU/ECSEL COMP4DRONES project. "Design Tools”, July’, 2021.

[34] ‘REXYGEN - Programming automation devices without hand coding’. https://www.rexygen.com/ (accessed Sep. 30, 2021).

2 C4D Reference Architecture Specification in Brief

In this section, we describe in brief the C4D reference architecture and its building blocks. More details can be found in the deliverable D3.2.

2.1 Overall Reference Architecture

The overall reference architecture of a drone system is shown in Figure 1. This architecture shows the interactions between the different blocks of the UAS. It is divided into four main groups: flight navigation, flight control, flight management, and mission management. First, the flight navigation includes the drone perception to gather information needed to navigate the drone from one location to another, while avoiding obstacles and preserving the geo-fence using the flight guidance. Second, the flight control executes the guidance commands to fly the drone from one place to another through the drone actuation. It also executes commands coming from the pilot directly. Third, the flight management contains functions for planning the flight trajectory and managing the UAV payload, data, and health. Fourth, the mission management has the mission planning and supervision functionality that are managed by the mission manager. Finally, there are external services that provide information for mission and flight planners to plan a valid mission and trajectory.

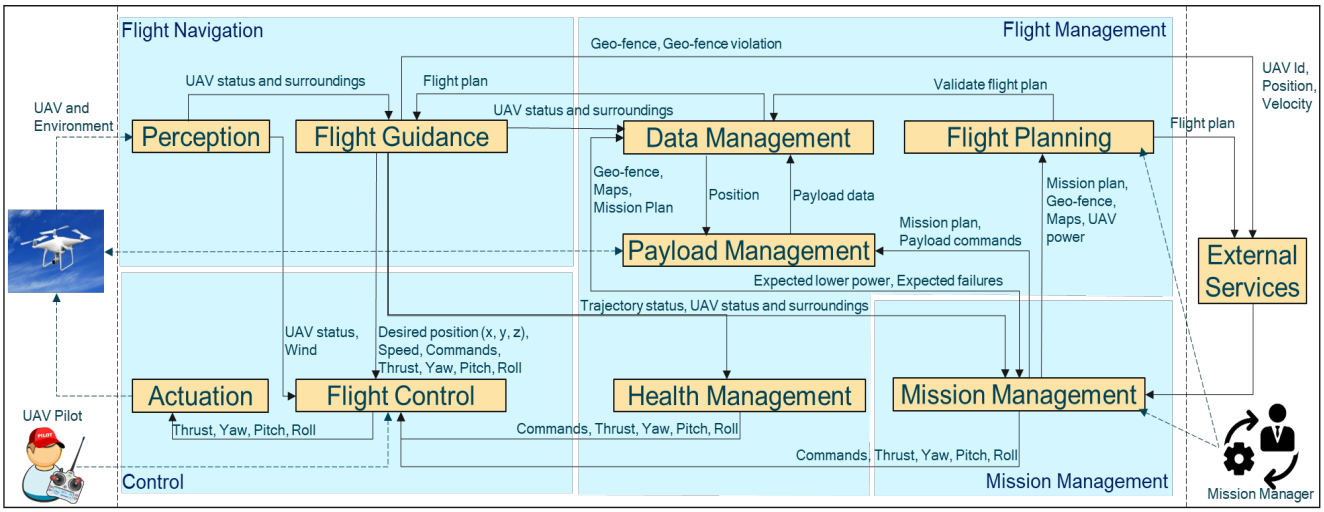


Figure 1: Flat view of the overall drone system architecture

2.2 Building Blocks of the Reference Architecture

A summary of all the blocks of the reference architecture is shown in Figure 2. In the following sub-section, we describe in brief these blocks and more details can be found in the deliverable D3.2.

2.2.1 Flight Planning

The flight planning refers to the determination of the desired, and valid path of travel (the "trajectory") from the vehicle's current location to a designated target, as well as desired changes in velocity, rotation, and acceleration for following that path. It consists of two sub-blocks: trajectory planning and trajectory validation. First, the *trajectory planner* provides a sequence of way-points based on a mission's objectives (e.g., mapping and surveillance), and constraints such as terrain, weather, air space and fuel. Second, the *trajectory validation* is the process of communicating with the external services to validate

the flight trajectory. The external services can process drone plans and checks for compliance with the applicable national regulations.

Figure 2: Overall blocks of a drone system

2.2.2 Flight guidance

The flight guidance refers to the process of monitoring and controlling the movement of a vehicle from one place to another. It consists of four main sub-blocks: trajectory execution, detect and avoid, geofencing, and command executor. First, the *trajectory execution* specifies how the UAV should travel between the way-points. This is an automatic function that considers the specifications of the UAV in order to generate a path consisting of segments of lines and circles that are compatible with the UAV's turn and climb capabilities. Second, *detect and Avoid (DAA)* block allows drones to integrate safely into civilian airspace, avoiding collisions with other aircraft, buildings, power lines, birds and other obstacles. This block observes the environment surrounding the drone, decides whether a collision is imminent, and generates a new flight path in order to avoid collision. It contains two blocks: obstacle detection and obstacle avoidance. Third, the *geofencing* block is responsible for keeping the drone flying without violating the geo-fence area defined by the regulation authorities. It contains two functions: geo-awareness and geo-fence preservation. Finally, the *command executor* is responsible for mode switching and executing the different mission commands. The commands can be: navigation commands, do commands, and condition commands.

2.2.3 Flight Control

Control refers to the manipulation of the forces, by way of steering controls, thrusters, etc., needed to execute commands while maintaining vehicle stability. This block includes flight control, position and velocity control, and attitude and rate control. First, the *flight control* specifies changes in the UAV's thrust, yaw, pitch, and roll to follow the specified path (typically represented as a sequence of line and circle segments) through the position and altitude controls. Second, the *position and velocity* controller take the inputs from the trajectory generation and compute the desired attitude and desired thrust. Finally, the *attitude and rate* controller take the desired attitude (or body rates), and compute the desired command to keep the drone stable, most of time around body axis (roll, pitch, yaw).

2.2.4 Actuation

The actuation is the function that writes data and control signals to an actuator to execute the flight commands. It contains two elements: mixer and actuators drivers. The ***mixer*** takes commands (e.g., turn right) and translates them into individual motor commands, while ensuring that some limits are not

exceeded. The *drivers* are a set of software modules that have the actual interface with the physical actuators to perform the specific actions.

2.2.5 Perception

The perception block includes the necessary functions for providing the information required by the other architecture blocks. It includes: UAV status (i.e., attitude, heading, positioning, and velocity), UAV surrounding, weather, and sensor's drivers. First, an *attitude* block provides attitude and heading information for the drone including roll, pitch and yaw. These measurements can then be used to derive an estimate of the drone attitude. Second, the *drone position* can be estimated using Global Navigation Satellite System (GNSS-based) or, when GNSS are denied using methods such as Simultaneous Localization and Mapping (SLAM). Third, the *velocity estimator* provides the drone speed. The speed can be directly measured using a speed sensor, or estimated using other sensors such as the drone's camera. Fourth, the *UAV surrounding* provides information about the drone environment including static and moving obstacles, and weather information such as wind speed. Finally, *drivers* are software modules that have a direct interface with the physical sensors to acquire the needed data.

2.2.6 Communication

Communication connects sensors, actuators, controllers, gateways and other edge systems. The communication mechanisms take different forms, such as a bus (local to an underlying system platform or remote), or networked architecture for command and control and service data exchange with external systems. Also, Quality of Service (QoS) characteristics such as latency, bandwidth, jitter, reliability, redundancy and resilience should be taken into account. The communication can have four forms. First, a *data link* is the means of connecting the UAV with the external infrastructure for the purpose of transmitting and receiving digital information (i.e., service data communication). Second, *telemetry* is a digital two-way data stream dedicated to ground control, which can both send data about the flight down to a ground control station, and send command up to the autopilot. Third, *internal communication* is used for exchanging data between the different blocks of the UAV or the ground control station. Such communication can be performed using publish-subscribe model or through directed links between the blocks. Fourth, *communication security* is the prevention of unauthorized interceptors from accessing communications medium, while still exchanging data between the UAV system blocks.

2.2.7 Health Management

The health management block is responsible for analysing flight data and for responding to problems in the power supply or faults that can occur in the drone engine, sensors, actuators, etc. It consists of three main blocks: power management, fault management, and diagnostics. First, *power management* continuously monitors important power supply parameters, while dealing with the varying power demands of the many aspects of the UAV's operation. It also optimizes the usage of the power source. Second, *fault management* is in charge of detecting and accommodating failures of the UAV. Faults in UAV are commonly represented as either actuator, plant, or sensor failures. The fault management technique depends on the type of the failure experienced. Finally, *diagnostics* enables the detection of problems' occurrences. It is responsible for real-time monitoring of key performance indicators of a drone, collecting and processing health data with intelligence, so that it can diagnose the real cause of a problem, and then alerting on abnormal conditions and deviations.

2.2.8 Payload Management

The payload management block is responsible for managing the payload added to the UAV to execute certain mission. It consists of three blocks: payload coordinator, payload controller, and data acquisition. First, *payload coordinator* takes the mission plan as an input and keeps track of the UAV position. When a location is reached where data acquisition (e.g., capture an image) or an action needs to be taken by the drone (e.g., drop a package), specific commands are issued to the payload controller or the data acquisition block to perform the drone mission. Second, *payload controller* executes the commands coming from the payload coordinator. Such actions are either performed by sensors (e.g., change a

camera angle), or through actuators (e.g., pick up a package). Finally, *data acquisition* is the process of sampling signals that measure real world physical conditions relevant to the drone mission (e.g., status of a collapsed building), and converting the resulting samples into digital values (e.g., constructing 3D image of the building) that can be later on manipulated by a computer.

2.2.9 Data Management Block

The data management block consists of capabilities for data persistence and storage, data distribution, semantic transformation, etc. These functions can be used in an online streaming mode in which the data are processed as they are received to enable real-time analytics. First, the *prognostics* block serves as a predictive analytics engine of the drone system. It relies on historical data of drone operation and performance, engineering and physics properties of the drone, and modelling information. Second, the *flight logger* keeps track of the UAV status, and records it for latter analysis by the prognostics block. By default, logging is automatically started when UAV is arming, and stopped when disarming. A new log file is created for each drone mission. Third, the *storage* block is responsible for storing all the needed information to execute a mission. Example information needed for executing a flight includes flight plans, geo-fence limitation, UAV model (e.g., UAV power supply). During mission execution, it also stores the generated information about the vehicle health during the flight (e.g., power level, engine temperature, etc.), and information about the drone environment. Finally, the *payload data analytics* encapsulates a set of functions for data modelling, analytics and other advanced data processing, such as rule engines.

2.2.10 Mission Management Block

The mission management block is used for planning and monitoring the execution of a drone mission (i.e., mission planning and mission supervision). First, *mission planning* is the process of producing a detailed flight schedule which concerns the calculation of the route for UAV(s) and the work plan of the payloads (sensors and actuators). A correct mission plan should meet the mission constraints set for the mission objective. Second, *mission supervision* represents a set of functions that the UAS exposes through user interface to enable human interactions with the UAS. It allows UAV operators to communicate with and control a drone and its payloads, either by setting parameters for autonomous operation or by allowing direct control of the UAV, usually from the mission start until landing or the end of the mission.

3 Underlying Concepts

3.1 Related Standards

3.1.1 AUTOSAR

The AUTOSAR standard (www.autosar.org) defines a meta-model that covers various aspects of an automotive system. It comes in two flavours, the classic and adaptive platform. The former is for static and resource constraint deployments, the latter allows for more flexibility. The meta-model includes a description of the application software, state and lifecycle related aspects and the execution resources.

At the application level, a system description is conceptually close to component-based modelling in UML or SysML: a set of component instances. A component (the meta-model uses the term application-sw-component-type) has a set of provided or required ports and one or more internal behaviours corresponding to functions (Runnables).

AUTOSAR has timing extensions, enabling the description of event chains, various timing constraints or the timing properties of the execution platform (see below).

With respect to deployment information, another part of the model describes executables which reference the software components. The executables in turn are referenced by process descriptions. Processes are running on a physical device called ECU (electronic control unit) whose capabilities can be described.

The description of an AUTOSAR model can be serialized in a standardized XML format (ARXML) which is supported by several tools. The execution platform of an AUTOSAR system includes an abstraction layer also called virtual function bus enabling interoperability and basic functions including for instance execution management and communication. The AUTOSAR stack runs on an RTOS, depending on the favour (classic or adaptive) based on OSEK or POSIX, respectively.

3.1.2 DO-178 and DO-254

DO-178 (Software Considerations in Airborne Systems and Equipment Certification) and DO-254 (Design Assurance Guidance for Airborne Electronic Hardware) are guiding standards for the development of software and hardware (respectively) that are used in aeronautical industries.

Amongst the contributions of DO-178, one of the most important ones is the categorisation of applications according to their criticality levels. These are the Design Assurance Levels (DALs), which are defined according to the nature of the consequences of a possible failure in the analysed system: A for Catastrophic (deaths), B for Hazardous (crew has reduced ability to cope and may result in potentially fatal injuries), C for Major (reduction of safety margins, possible injuries), D for Minor (possible discomfort) and E for No effect. Regarding DO-254, which also uses the concept of DALs, it defines processes that the development of complex electronics can follow to guarantee its reliability.

Each DAL is associated with a set of objectives to be achieved for validation. These objectives are cumulative, which means that for achieving DAL-C one needs to also achieve DAL-D in addition to DAL-C specific objectives. These are summarised as follows:

- DAL-E: No objective to be achieved.
- DAL-D: 26 objectives, regarding partitioning integrity, consistency of high-level requirements, traceability until system requirements, executable code conforming to high-level requirements, executable on the target platform, tests to cover high-level requirements, configuration management, etc.
- DAL-C: 62 objectives, i.e., 36 new objectives, including software lifecycle, requirements that are traced, derived and demonstrated, correct algorithms, consistent software architecture (not necessarily verifiable), conformity to standards, etc.

- DAL-B: 69 objectives, i.e., 7 new objectives, including verification regarding target execution, verifiable software architecture, more demanding code coverage.
- DAL-A: 71 objectives, i.e., 2 new objectives, regarding an even more demanding code coverage, including “dead code”.

These safety norms are analogous to other ones found in other domains, such as automobile and trains, according to the table below.

Table 1: Criticality levels in different norms

Domain	Criticality level				
Civil aviation (DO178/254)	DAL-E	DAL-D 10^{-3}/h	DAL-C 10^{-5}/h	DAL-B 10^{-7}/h	DAL-A 10^{-9}/h
Automobile (ISO 26262)	Quality management	ASIL-A	ASIL-B/C	ASIL-D	
General (IEC 61508)	-	SIL-1	SIL-2	SIL-3	SIL-4
Rail (CENELEC 50126/128/129)	-	SIL-1	SIL-2	SIL-3	SIL-4
Safety impact	No impact	Minor	Major	Dangerous	Catastrophic

In order to analyse the safety impact a component has in case it fails, its dependency relations must be clear. Specially if a component with a higher criticality depends on the correct functioning of a lower-criticality component. For example, if the passenger light control is executed in the same hardware as the attitude control of an airplane, it can interfere with the attitude response time and even block its execution, and this would require a normally low-criticality system (passenger light control) to be treated as a high-criticality one (attitude control) so that the safety of the aircraft is guaranteed.

3.1.3 SORA (JARUS)

The Joint Authorities for Rulemaking on Unmanned Systems (JARUS) is a group of experts in aviation safety organisations from around the world. They propose holistic methods for assessing risks of drones and their operations, dividing the operations according to the risks involved: Open (low risk, no requirement needed), Specific (moderate risk, requirement needed for the operation except in standard scenarios with certified operator), or Certified (high risk, needs certification of the drone and the operator).

Their document regarding Specific Operations Risk Assessment (SORA), more adequate for COMP4DRONES’s cases, suggests a methodology for risk assessment to ensure safe Specific-category operations, based on six Specific Assurance and Integrity Levels (SAILs), which are themselves based on a Ground Risk Class (GRC) and an Air Risk Class (ARC).

3.1.4 IEC 61508

IEC 61508 is an international standard published by the International Electrotechnical Commission which describes methods related to the lifecycle of safety-related systems. It is based on the concept that these systems must either execute without failures or fail in predictable ways (fail-safe). The standard defines a safety lifecycle based on engineering best practices to identify and mitigate design errors and omissions, and it also counts on a probabilistic approach to assess the impacts that device failures cause on safety. The standard was later adapted to automotive systems in the form of the ISO 26262 standard, which serves itself as a basis for AUTOSAR.

The document states that an entire device must be considered to have the same criticality (the highest among its parts), unless it can be shown that a component has no direct influence on the safety function, or it cannot interfere with any component of higher criticality. Although there is no explicit mention to the concept of Freedom of Interference of ISO 26262, this notion is present.

Another concept defined in the document is that of a Software Insurance Level (SIL), which can have a value from 1 to 4, according to the following list:

- **SIL 1:** Probability of dangerous failure (PFD) between 10^{-1} and 10^{-2} for low demand components, and between 10^{-5} and 10^{-6} for high- or continuous-demand components.
- **SIL 2:** PFD between 10^{-2} and 10^{-3} for low demand components, and between 10^{-6} and 10^{-7} for high- or continuous-demand components.
- **SIL 3:** PFD between 10^{-3} and 10^{-4} for low demand components, and between 10^{-7} and 10^{-8} for high- or continuous-demand components.
- **SIL 4:** PFD between 10^{-4} and 10^{-5} for low demand components, and between 10^{-8} and 10^{-9} for high- or continuous-demand components.

It also recommends the use of specific programming languages and certain additional measures for languages often used for safety-related code, like C. For example, the Motor Industry Software Reliability Association's (MISRA) guidelines for writing C code is a widely used de-facto standard for writing embedded code in the majority of safety-related industries.

3.1.5 MARTE profile for UML

The Unified Modelling language (UML) [1] is a modelling language intended to provide a standard way to visualize the design of a system. It supports different types of diagrams (e.g., structural, behavioural) that can be interrelated among them, and together provide different perspectives of both a global view and specific parts of a system. It is in much also a general language, providing a graphical syntax for their elements. This can explain, not only its success on the field of software engineering, but also its wide acceptation on other contexts, like system engineering, business models, etc.

Other important aspect for a pragmatic usage of any language is the tooling and the specific methodologies around the language, which delimit its usage, i.e., how to capture the models, and state what they serve for. Nowadays, there is a rich ecosystem of UML modelling tools, e.g., Papyrus, Modelio, Umbrella, Raphsody, or Visio, to mention some. An immediate application of UML diagrams is to document a specification intent and let interdisciplinary teams exchange information. However, many more application possibilities on many different fields are possible. This is dependent on the tooling around, but more importantly on the possibility of smoothly extend and specialize the UML language. This can be easily understood if we only think on one specific UML element, e.g., a UML component. The UML component is a very generic and abstract construct, which can be reused in many contexts (it can be thought as a HW component, as a SW component, as a machine component, as a team component, etc). It is obvious that each tool framework on each context can re-interpret component semantics. This ambiguity and generality enable its wide applicability.

The big risk with this approach is that UML could lose its "unification" potential (and thus its own essence) due to the different interpretations and usages of the same elements given by different methodologies. Fortunately, UML provides a convenient extension mechanism for that: **the profile**. Roughly, the profile can be also understood as a way to create a (graphic) DSL on top of UML. More specifically, A UML profile comprises a specific set of extensions for enabling the build of UML models for particular domains. There are several extension mechanisms (stereotypes, tag definitions, constrains) which are applied to the existing UML modelling elements (components, classes, attributes, etc). A key aspect is that these extension mechanisms refine the standard semantics in a strictly additive manner. This prevents any contradiction with the parent semantics. This is crucial for optimizing the understanding in interdisciplinary tools. Complex cyberphysical or full electronics systems will involve different type of components (e.g., HW, SW, etc), but the common semantics of the component is the same. At the same time, the extension, e.g., a *<<HW_processor>>* or a *<<OS>>* stereotype provides and additional, distinguishing semantics in a specific domain (e.g., electronic system design). This is also key for consistency among models, and tools in the specific domain. In summary, profiles are key for UML as a cooperation enabler, both within a same domain, and across domains. This powerful

extension tool has encouraged the development of many profiles to make their own required extensions. But these proprietary profiles create again an undesirable dispersion of modelling approaches.

A way to prevent such dispersion on the building up a common modelling language able to cover many different domains is the one taken by OMG, which supports also standard profiles, in diverse fields, like for instance, *UTP2* for test, *BPNMProfile* for business processes [2].

The profile for model-driven development of Real Time and Embedded Systems, i.e., the MARTE profile [3], is comprised among those standards. This is a relevant profile for COMP4DRONES modelling methodology. An autopilot, and the navigation components on board the drone in general, are in general real-time embedded systems, with specific performance constraints, with some typical software architecture (aspect addressed for the autopilot in this report), with a specific resource constrained HW platform architecture and with a specific mapping of the SW architecture on the HW architecture. Covering all these aspects (as well as others) is required for a sufficiently expressive modelling, and thus for considering all the required input data for the analysis methodologies relying on these models.

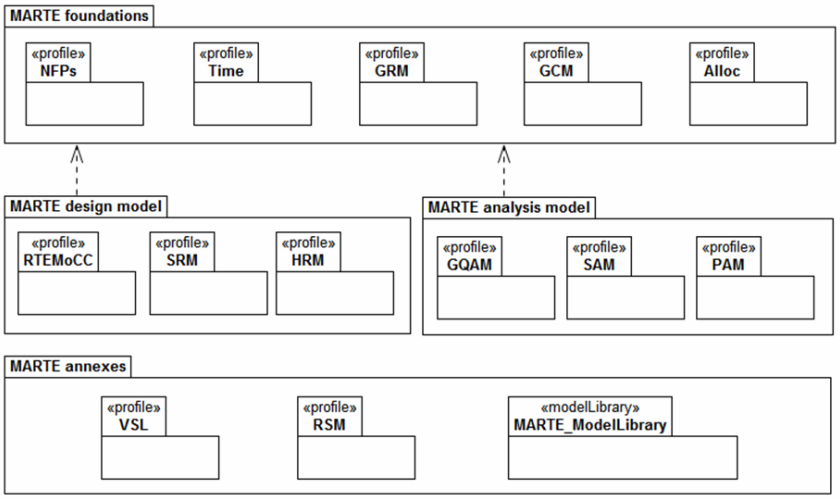


Figure 3: MARTE packages

MARTE profile contributes a number of extensions, structured on a number of packages, as shown in Figure 3. The profile has a “foundation” set of packages which reflects the main two goals of MARTE, i.e., to enable the modelling the main aspects of real-time and embedded systems, and to enable encompassing the model with annotations to enable analysis. In this line, the foundation packages enable time related annotations (Time package) and non-functional properties (NFP package). The foundation packages also support the generic modelling of components (GCM), for high-level functionalities, and resources (GCM), as well as the allocation among them (Alloc package). From this foundation, more specialized extensions are provided. Without entering in full detail, the SRM and HRM packages for software and hardware resource modelling can be taken as an example. The SRM package enables the modelling of specific SW resources, e.g., to capture aspects like active vs passive services or classes. Similarly, the HRM package enables to discriminate UML elements of the same type, e.g., components, as different class of HW resources, like processors, memories, or I/O devices.

Thus, MARTE provides a common, standard way to model both HW and SW aspects of real-time electronic systems and thus more guarantees for interoperability among tools. Moreover, the specification elements supported enable to build models for performance analysis, an important design intent, which extends the usage of MARTE models beyond documentation.

All these advantages have encouraged the adoption of the MARTE standard, by several methodologies (e.g., MoPCom [4], Optimum [5], S3D [6] and Hepsycode [7], the latter two involved in the COMP4DRONES partnership). S3D and Hepsycode have in common a devotion to analysis and related system-level design activities, like design space exploration. In some cases, additional exploitation of

the produced models is proposed. In fact, S3D proposes to use the UML/MARTE model as a centric data resource feeding the design activity in what is called a “single-source” approach. In this approach, the UML/MARTE model does not only serves for documenting, team interaction and analysis, but it is also used also as entry point for an automated implementation process (software synthesis) [8].

Despite the great variety of aspect covered by MARTE extensions, achieving a complete fit for all application domains and design purposes is challenging. Because of this, it is not an isolated fact that those methodologies, despite trying to fully rely on UML and standard profiles and make an optimal exploitation of them, eventually make proposals of adaptations or extensions of MARTE (or other profiles), e.g., S3D and Hepsycode for design space exploration. This is exactly one type of analysis tackled in this report, i.e., which type of modelling elements are required for drone embedded electronics, and more specifically for an autopilot modelling, and with such a basis, if MARTE can cover all of them or some type of extension is required.

3.1.6 AADL

The Architecture Analysis and Design Language (AADL) is used to describe real-time embedded systems. Initially conceived for avionics, it can be used to model and analyse any kind of time-critical software and hardware. AADL can be used as a graphical and as a textual representation (and also as an XML/AAXL representation).

Its main features, represented graphically in Figure 4, consist of:

- A **processor**, which is a physical device capable of executing calculations, such as a microcontroller / computer
- A **device**, a physical component for which the internal structure is unknown, such as a sensor
- The **memory**, responsible for storing data, such as a hard drive or a random-access memory
- A **bus**, which transports data between physical devices, such as a network
- The **data**, an abstract structure containing information
- A **subprogram**, which is a function that can be defined in a programming language
- A **thread**, representing a series of functions
- A **thread group**, capable of representing a certain hierarchy between threads
- A **process**, an isolated execution environment containing at least one thread
- A **system**, containing components that can be independently manipulated

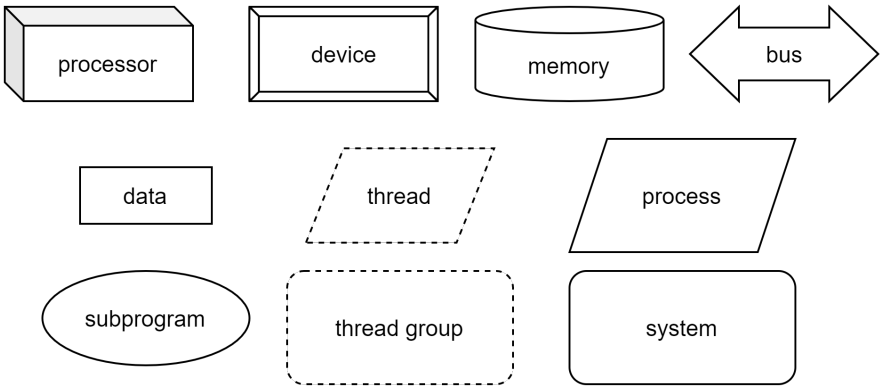


Figure 4: Elements of the AADL language

Using these elements, we can define their interfaces with the use of **ports**, as depicted in Figure 5. They represent input, output, or input-and-output variables, and they can be of one of the three categories:

- **Data**, for transporting information, such as in a whiteboard
- **Event**, a signal to activate a function
- **Event data**, the addition of the previous two, such as the arrival of a message

Figure 5: Ports in a thread, with "data" and "event" as inputs and "event data" as an output

After specifying the components as such, one can create an **implementation** of the components, nesting subcomponents and connecting ports. This will represent the architecture of a given embedded system. Hence, the language allows the creation of a component tree that represents the relation between components and subcomponents. In Figure 6, the possible nesting relations are represented.

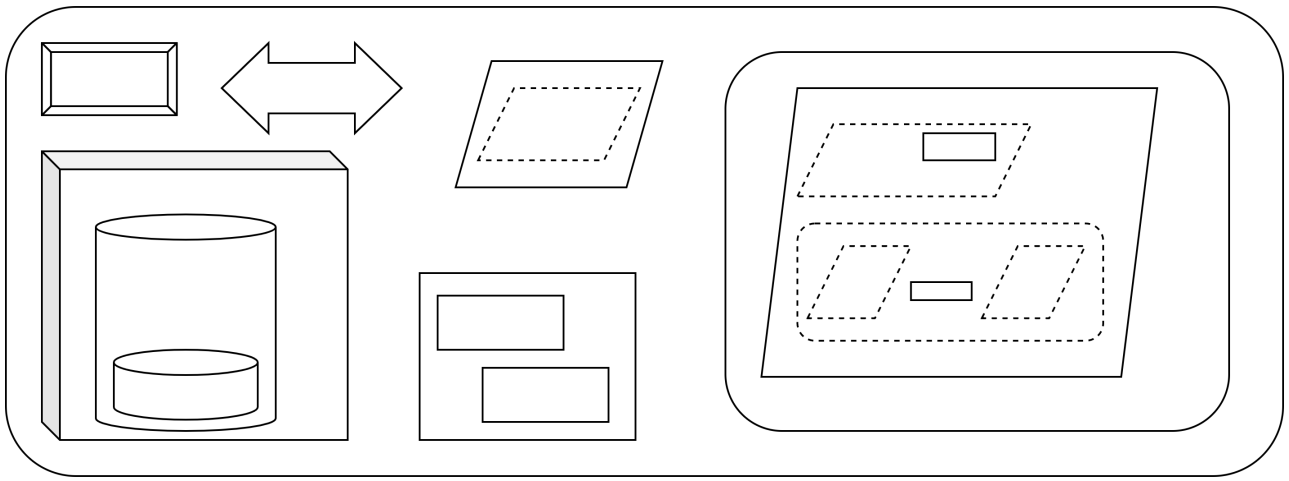


Figure 6: Possible nesting in AADL, starting with the outermost element - a system

3.1.7 Other Standards

There are also other standards regarding drone software, such as: AS6512A, AS6513A, AIR6521, AIR6515, AIR6517, AIR6516, AS6518, AIR6520, AIR6514 and AIR6519, from the Society of Automotive Engineers (SAE), which cover the architecture of the unmanned systems' control segment; F3269-17 and F3201-16 from the American Society for Testing and Materials (ASTM) regarding dependability and behaviour bounds of UAS software, especially those containing complex functions; P1937.1 from the Institute of Electrical And Electronics Engineers (IEEE), about standard interface and performance requirements for payload devices in drones; AC20-148 from the Federal Aviation Administration (FAA) about using reusable components; and G-010-1993 and G-118-2006 from the American Institute of Aeronautics and Astronautics (AIAA) concerning the use of Commercial Off the Shelf (COTS) components and reusable software.

New standards are being published as the technology rapidly advances, hence this research is not extensible and should be constantly reviewed.

3.2 Related methods

3.2.1 MBSE / MDA

According to the International Council on Systems Engineering (INCOSE), "Model-based systems engineering (MBSE) is the formalized application of modelling to support system requirements, design,

analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases."

Model Driven Architecture (MDA), on the other hand, "is an approach to software design, development and implementation spearheaded by the OMG [Object Management Group]. MDA provides guidelines for structuring software specifications that are expressed as models."

Both these approaches are based on the use of modelling techniques for the system according to distinct points of view, e.g., regarding either project requirements, expected functionalities, or real-time properties for verification and validation. Therefore, they can sometimes have an interchangeable meaning.

Models can be defined as a representation of a system that relies on certain key characteristics of the modelled object, resulting in an abstraction of the object. Then, by definition, models are incomplete, and hence flawed, in the sense that no model can represent in a satisfactory way all the characteristics of the object – only the object itself would be able to. Nonetheless, abstraction can allow useful analyses to be made regarding the model, whose results also apply to the object.

Every model needs to follow strict rules to be instantiated, which is the modelling language. This language is itself a model, whose instance is the model of a certain object as discussed before. Therefore, it can also be called a meta-model. The meta-model also must follow specific rules, expressed by a meta-meta-model, which in turn often follows its own rules. This can be viewed in Figure 7.

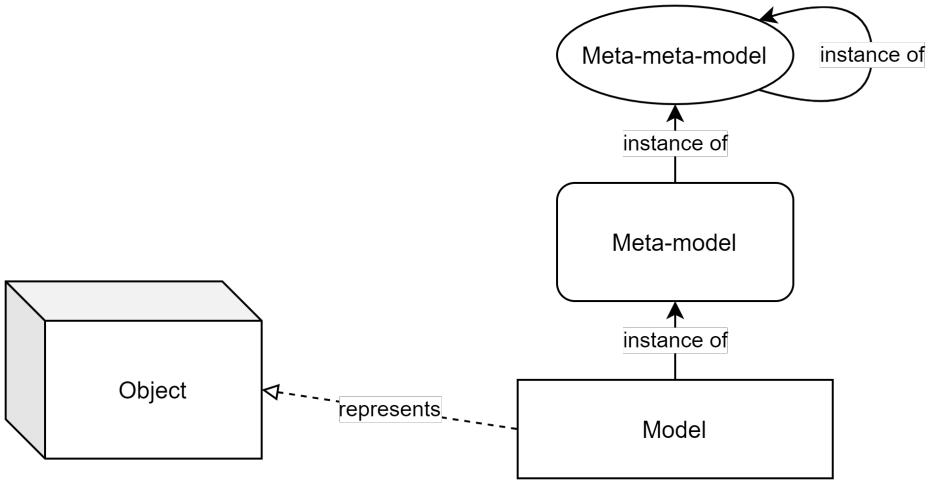


Figure 7: Models and meta-models

Since models allow an automatic analysis of their properties, a model can be automatically transformed in another model according to a different meta-model, if there is an explicit relation between each element of the two meta-models involved. This explicit relation is expressed itself as a transformation model. Knowing that the source code of a program can be viewed as a model of the actual piece of software that it generates after compilation, it becomes also clear that models and model-transformations allow certain software models to have their code automatically generated, which is a very powerful tool for software development.

3.2.2 Arcadia

The Architecture Analysis and Design Integrated Approach (Arcadia) [9], developed by Thales based on languages such as AADL and SysML, is a model-based engineering method for system, hardware and software architectural design. It is also registered as the Z67-140 standard of AFNOR ("Association Française de Normalization" – French Standardisation Association).

The method creates a traceable link between requirements (operational analysis) and the subsequent steps of development (functional needs, non-functional needs, logical architecture), until its most concrete implementation (physical implementation), as can be seen in Figure 8. This methodology is well suited for development approaches based on the V cycle.

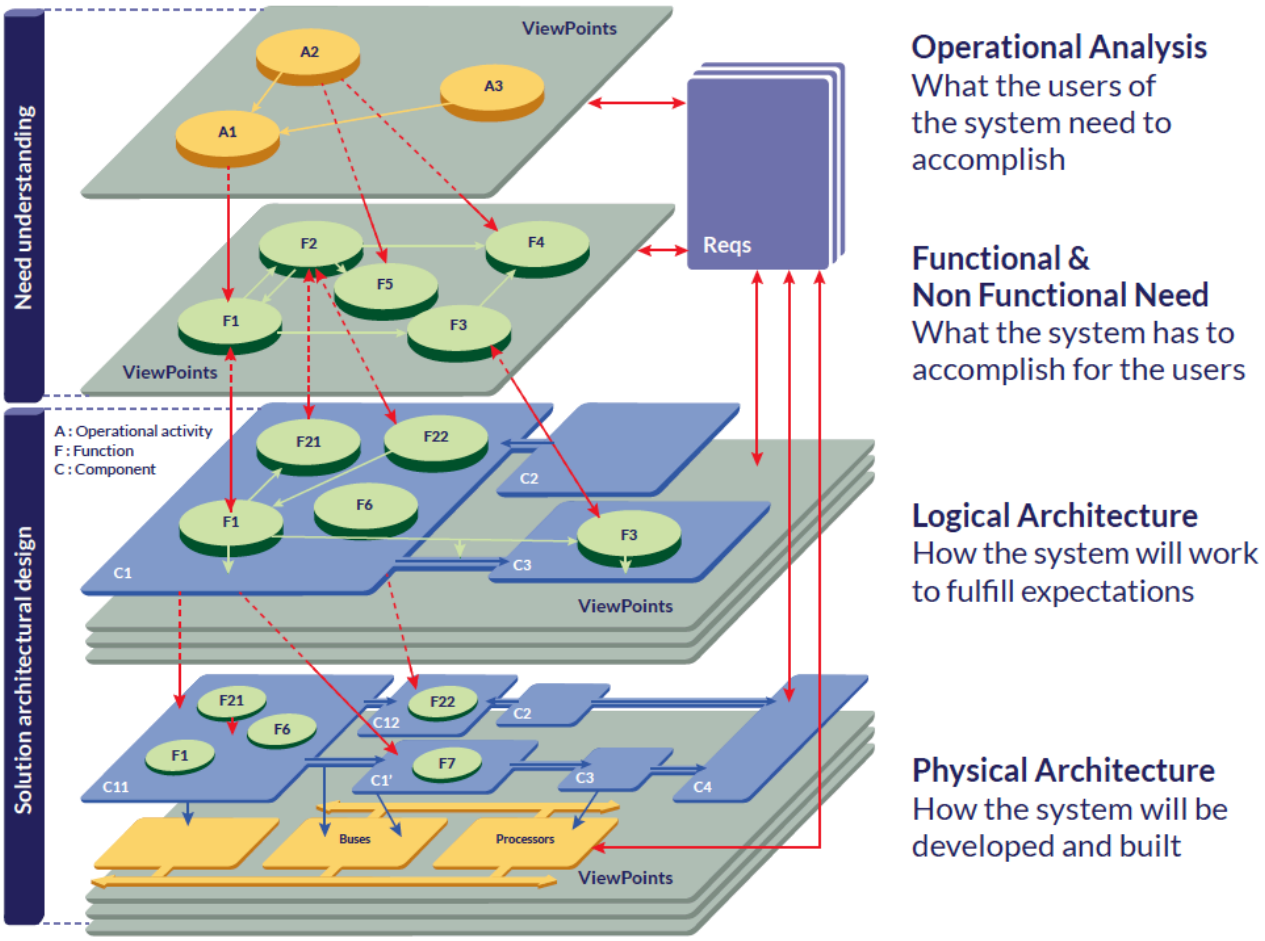


Figure 8: Arcadia Method representation

ARCADIA is supported by the tool Capella, which is an open source MBSE tool built using the Eclipse framework. Capella allows the instantiation of system models from the Operational Analysis to the Physical Architecture, keeping their traceability and also providing some automatic model transformation and generation.

3.2.3 RobMoSys

RobMoSys is a development approach that aims to coordinate the whole community's best and consorted efforts to realize a step-change towards an industry-grade European ecosystem of fully model-driven methods and tools for composition-oriented engineering of robotics software systems. RobMoSys envisions a business ecosystem for robotics software, in which a large number of loosely interconnected participants (robotics-domain experts, component suppliers, system builders, behaviour developers, safety engineers, etc.) depend on each other for their mutual effectiveness and individual success. For this, RobMoSys separates abstraction levels and concerns and manages the interfaces between different roles in an efficient and systematic way by applying model-driven methods and tools in an integrated approach built on top of the current code-centric robotic platforms.

RobMoSys organizes the business ecosystem in 3 tiers (Figure 9). At the uppermost level, Tier 1 structures the ecosystem in general for robotics, independent of sub-domains. It is driven by few representative experts in ecosystems and composition, called the *ecosystem drivers*. It provides all modelling guidelines and structures enable composition. A lower tier conforms to structures defined by a higher. At the middle level, Tier 2 structures the particular robotics sub-domains, such as object recognition, manipulation, SLAM, etc. It is driven by *domain experts* who define domain models in a community effort. Finally, at the lowermost level, Tier 3 uses the domain structures from Tier 2 to fill them with content and create concrete robotics products and services.

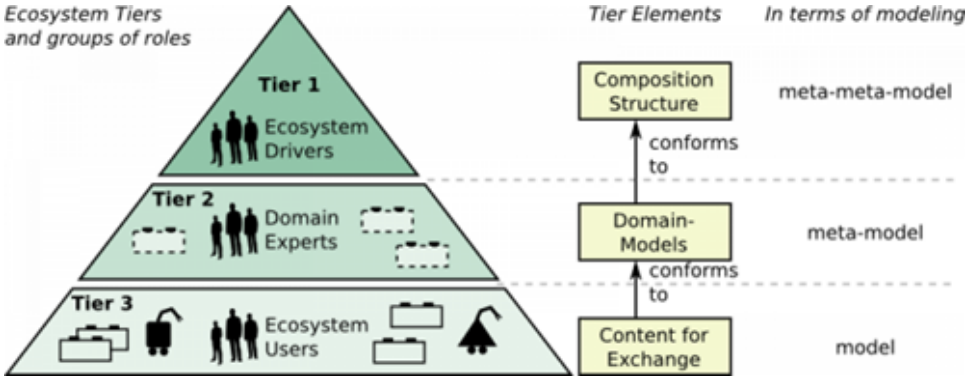


Figure 9: Tiers in a business ecosystem (from the RobMoSys Wiki[1])

RobMoSys uses the concept of *architectural patterns* to identify the *composition structures* that form the ecosystem foundations at Tier 1. Architectural patterns describe best practices and recurring designs that facilitate building robotics software systems by composition and support separation of roles. An architectural pattern combines several *abstraction levels* with a number of *concerns*[2]. The composition structures[3] are the formal models of relationships and interfaces between the different abstraction levels. To define these structures, RobMoSys applies a balanced combination of "freedom *from* choice" and "freedom *of* choice" principles. The composition structures are flexible enough to address specific requirements and to not hinder progress (freedom *of* choice), but at the same time they positively limit the set of available options to ensure composability and separation of roles avoiding fragmentation (freedom *from* choice).

Examples of Tier 1 composition structures are *metamodels* of software components, service definitions, deployment, platform and cause/effect chains, for stepwise management of extra-functional properties; skills, which provide access to the functionalities realized within components and make them accessible to the task level; tasks, which represent what and how a robot is able to do something, independent of the realization. Examples of Tier 2 domain-specific models are service definitions —data structures, communication semantics and additional communication-related properties— for, e.g., robot localization, navigation, etc. Finally, examples of Tier 3 models are software components, e.g., for AMCL localization, Gmapping, etc. providing a localization service; task plots, e.g., to instruct a robot to make coffee; complete composed applications, such as a restaurant butler robot.

3.3 Software Bus

3.3.1 Introduction to software buses

In the scope of the integration of functions to assemble into a whole program, there are several possibilities to implement their data exchange. These possibilities, however, depend completely on the way these functions are implemented concerning some execution aspects, notably the threads and

[1] https://robmosys.eu/wiki/general_principles:ecosystem:start
[2] https://robmosys.eu/wiki/general_principles:separation_of_levels_and_separation_of_concerns
[3] https://robmosys.eu/wiki/modeling:composition-structures:start

processes where they execute. For example, depending on the Operating System (OS), a pair of functions can be executed in the same thread, different threads but same process, different processes, or on totally different platforms, and each of these four possibilities requires the function interfaces to be implemented in a specific way.

The use of software buses can abstract this complexity and allow function interfaces to be implemented in a unique way, regardless of where the function will be executed in relation to the others. The bus implementation will be responsible for managing the data transfer. But this abstraction does not come for free: several performance guarantees are compromised or even removed completely from the system, since the analysis of data communication may become impractical.

3.3.2 Standard and de-facto standard software buses

Many means of communication, protocols and frameworks exist. They are characterized by their scope: intra-process, inter-process hosted by the same OS, inter-OS deployed on the same processor, inter-processor, etc. Nevertheless, software communicating entities are always hosted in a running thread, which is part of a process, defining a memory space. Therefore, what frameworks define as software buses relies, at low level, on a limited set of tools: thread-level local variables, process-level global variables, IPC (Inter Process Communications), specific communication means, routed network protocols such as IP sockets and specific non routed network protocols.

3.3.2.1 Basic communication mechanisms

Within the scope of a single thread:

A thread represents a single thread of execution, therefore, different functions hosted by a thread usually communicate simply through local or global variables, without any need of synchronization mechanisms. Indeed, data produced in a variable by the execution of a function can be read by a function executed later in the same thread. Note that if some global variables are used without synchronization mechanism, race conditions could occur if the same variables were to be used in a parallel entity (for example, an additional thread). As a result, when expressing a communication through unprotected global variables, which is common in autopilot implementations, it is important to express if variables are accessed within a critical section or not.

Within the scope of a single process:

A process represents a memory partition, hosting one or more threads, which are executed concurrently (either in real parallelism on a multicore platform, or are interlaced on the same core). Communicating threads have access to the same global variables, making their communication easy, nevertheless their concurrent execution can generate, without synchronization mechanism, a race condition. For example, given a global variable v initialized at zero, two threads executed in parallel, both incrementing v, can issue two different results: v=1 or v=2, depending on when their executions or pre-emption occur. The result v=1 occurs if both thread reads the (initial) value of v before the other modifies it after incrementing it. This is an example of one of the basic problems of concurrency: resource sharing. Other concurrency problems such as deadlock and starvation, are also to be aware of.

Even in a single threaded process, race conditions can occur because at low level, interrupts can be used to run short programs called Interrupt Service Routines (ISR), that may read or write variables used by threads of the process. In this case, race conditions can occur if no mechanism is used to prevent it.

As a result, basic communication mechanisms at the process level define mechanisms to avoid race conditions, such as semaphores or Hoare monitors. These mechanisms are at the core of blackboard type communication where reader(s) and writer(s) read and write asynchronously in a global variable. These mechanisms are also at the core of loosely synchronous communication mechanisms such as message queues (also called mailboxes), and synchronous communication mechanisms such as rendezvous. Note that this last type of communication mechanism is usually avoided in high integrity

systems since they were shown temporally unsafe [10]. Other mechanisms, such as software events, can also raise race conditions and should also be avoided in high integrity systems.

The communication means offered by operating systems for threads to communicate are dependent on the operating system. In Linux based systems, they are defined in POSIX 1003.1 standard, while they are proprietary in OS such as FreeRTOS, or Microsoft Windows. While they may differ at the margin (possibility to customize buffer size or not, blocking or not blocking a thread sending in a full mailbox, etc.), every operating system offers at least tools to ensure mutual exclusion, allowing blackboard communication, and producer/consumer communication (i.e., the consumer thread is asleep until the producer sends it a value).

Within the scope of a single OS:

Apart from higher abstraction instruments, allowing processes to exchange information and notify each other of events is an inherent function of almost any operating system. These *Inter-Process Communications* (IPC) mechanisms have different appropriateness depending on the use case and combining them may provide the desired properties for the use case. For example, shared memory-based IPC admittedly is the most efficient for high volume data transfer, however, it lacks a proper synchronization and, therefore, it should be coupled with signals (or semaphores). Socket-based (typically TCP/IP) communication is another IPC mechanism with the added benefit of enabling processes communication while executing on different machines. Table 2 shows an indicative latency and throughput comparison of Linux IPC mechanisms using standard benchmarking software[4].

Table 2: Indicative latency and throughput measurements of IPC mechanisms (OS: Ubuntu, Processor: i5-3470, Kernel: 4.15.0, 100 MB[5], 100 tests).

IPC Type	Latency			Throughput (Saturation)		
	Average [us]	Average [Normalized]	Standard Deviation [us]	Average [Mbps]	Average [Normalized]	Standard Deviation [Mbps]
mmap	0,11	1,00	0,24	19 338,70	15,85	12 025,51
shm	0,11	1,06	1,57	18 715,94	15,34	9 273,89
signal	2,87	26,85	6,28	N/A	N/A	N/A
mq	5,71	53,36	13,69	1 220,41	1,00	1 399,95
fifo	7,70	71,93	13,18	3 952,67	3,24	6 586,13
domain	8,36	78,13	9,62	4 697,45	3,85	2 881,17
pipe	8,42	78,70	16,72	9 313,15	7,63	18 474,27
tcp	19,84	185,44	36,69	2 898,24	2,37	2 754,68
zeromq[6]	42,51	397,27	41,85	3 557,66	2,92	304,46

Dealing with software component communication on an operating system level requires more skill, time and awareness of such underlying concepts as multi-threading, memory virtualization and system calls. Furthermore, such distributed communication mechanisms in themselves do not resolve such challenges as location-independence of components, flexible component (re)deployment, integration of legacy code, heterogeneous components [11]. Nevertheless, dealing with the challenge on this level presents an opportunity as the communication can be tailored for the specific architecture and application and can be optimized for higher performance, e.g. zero-copy communication, custom synchronization.

Within the scope of a single partitioned processor:

[4] https://github.com/goldsborough/ipc-bench
[5] Not all mechanisms support a packet size of 100 MB, therefore MQ was tested using 2048 bytes packets, FIFO, PIPE, Domain, TCP using 32768 bytes.
[6] ZeroMQ is not a standard Linux-provided IPC mechanism.

Some specific platforms, such as ARINC 653, allow a safer partitioning than other multi-process OS by statically assigning time and memory to processes, called partitions. In such platforms, a specific inter-partition communication mechanism is defined, such as the Inter-partition communication API in the APEX of ARINC 653.

Within the scope of a multiprocessor system-on-chip:

In the scope of a multiprocessor system, the connected Central Processing Units (CPUs) can be treated unequally using an Asymmetric Multiprocessing (AMP) approach. For example, a certain CPU might be the only one allowed to execute OS code, or to treat I/O operations. This contrasts with the Symmetric Multiprocessing (SMP) approach, in which every processor has full access to every I/O device, shares the same memory and is controlled by the same OS instance, which treats every processor equally. For the symmetric case, a common communication protocol between processors hosted by the same board is OpenMP, a protocol encapsulating rpmsg, an inter-processor communication framework, and remoteproc, a remote procedure call framework. In the case of asymmetric architectures, a common communication protocol between different types of processors is OpenAMP. Both types of communications are highly dependent on the underlying hardware architecture, but a common way of implementing it is by sharing a part of the global memory, which can be accessed by the different cores or core clusters.

Communication between different platforms:

Communication based on the Internet Protocol (IP) can be used for connecting different devices in most networks. The protocol consists of the sender attaching a header to the message being sent, containing the IP address of the sender and the IP address of the destination. This allows a network containing switches to redirect the transmitted messages based on their content. This communication protocol is routable, in opposition to local network protocols.

3.3.2.2 Relevant common software buses

In the context of UAV and UAS, software buses are abstractions provided by frameworks, hiding communication means implementation that were presented in the previous section, and typically offering blackboard and producer/consumer abstractions among software entities that can be executing on different platforms.

ROS and ROS2:

The Robot Operating System (ROS, https://index.ros.org) is an open-source robotics middleware. It has been developed since 2007 at Stanford University and later (until 2013) by the company Willow Garage. Since 2013, development control has been handed to the Open Source Robotics Foundation (OSRF). The term operating system appears in its name, since ROS handles some issues that are close to that of an operating system, e.g. hardware abstraction and threading mechanisms for message handlers. However, it is actually a middleware running on top of a (real-time) operating system.

A ROS system consists of a set of *nodes* that exchange data or provide/require services. The communication of these nodes forms a *graph*. The model is thus well suited for a drone in which sensors and actuators communicate with perception and control algorithms running on potentially distributed hardware.

ROS comes in two flavours, the classical ROS or the newer ROS2 (https://docs.ros.org/en/galactic/). The latter addresses missing real-time capabilities, reduces overhead and bases communication on the RTPS (real-time publish/subscribe) protocol used by the OMG standard DDS (data distribution service, https://www.omg.org/omg-dds-portal/). With the latter, ROS inherits rich configuration options related to the quality of service of communication, ranging from queue sizes to failure handling. This renovation fosters the use of existing (open-source) communication back-ends that are mature and optimized (such as openSplice or eProsima DDS).

ROS supports the exchange of messages based on a publish/subscribe mechanisms: A publisher associates data with a topic and subscribers of this topics receive the data. In addition to message-exchange, ROS supports services (queries in RobMoSys terms), i.e. request-response interactions. ROS standardizes the way in which to specify messages and services. A message structure is a set of attribute definitions, each is formed by a (qualified) type name and the attribute name with optional indicators for arrays and a default value. A service is a combination of a request and a reply.

ROS also ships with a set of libraries (message packages) that define commonly used messages and services for various aspects of a robotic system, e.g. geometry (Point, Quaternion, …), trajectories or common sensors (battery-status, camera, …). The de-facto standardization of these libraries in combination with the use of a standardized middleware fosters interoperability, an important goal also in Comp4Drones. The autopilot PX4 has adopted the ROS way to specify messages (see section 3.7) and added a library of drone-specific messages. It is thus interoperable with ROS2 based components for additional tasks such as perception that are executed on a companion board.

ROS2 supports node configuration and lifecycle management at runtime. Originally introduced in microROS (a ROS variant for resource constrained systems/microcontrollers, https://micro.ros.org/), ROS nodes support application- specific states, called *operational modes* via the system-modes package (https://index.ros.org/p/system_modes/). This enables for instance the use in in a drone mission divided in different steps that might correspond to a specific operational mode.

MAVLink:

MAVLink (http://mavlink.io/) is a relevant standard for drones. It is a lightweight messaging protocol which supports the communication of on-board components, as well as the communication of the drone with the “offboard” components, e.g., ground control station (GCS). This is a key feature for C4D, as a communication standard which enables the interoperability, shall be able at the same time to support the many different types of communication needs that arise from the diversity of components and architectures. MAVlink supports both point-to-point, and public-subscribe (and thus point-multi-point) configurations, which adapt to the different communication needs. For instance, an auxiliar geo-referencing system can use a direct-link configuration to communicate with the flight-control, or a multi-point configuration if such geo referenced data needs to be sent to several parties, e.g., to the flight control, to the geo-fencing, and to the telemetry sub-systems.

MAVlink does not state on the physical levels, not even on the network level. For instance, messages can be sent on a serial USB or on a wireless links with UDP. At the same time, it provides features and mechanisms for considering the underlying technologies and design needs. For instance, it involves a low overhead and supports retransmission, packet drop and corruption detection, which is relevant for the wireless links among drone and GCS, or for the speed and reliability required by the functionalities using the data, e.g., the flight control.

Another interesting feature of MAVlink is that it supports several programming languages and so targets architectures. Therefore, overall MAVLink suits well to the system-level nature of the C4D architecture, to target different types of connections among the functional blocks defined by the architectures.

Aware of these benefits, some C4D partners, like ACORDE are adopting and adapting MAVLink as standard interface of its geo-referencing systems, i.e., on actual implementation. This MAVLink “nature” can be implicitly used, as a way to ensure inter-operability of the integrations, or used, for instance, to label the connections in the COMP4DRONES system-level architectural models, so that interoperability can be assessed at early design stages.

3.3.2.3 Ad-Hoc software buses

Some specific lightweight software buses allow fast communication between software components. In the context of two autopilots that C4D observed in detail, ABI is a lightweight bus used by Paparazzi, while uORB is used by PX4.

uORB:

uORB is the object request broker used by PX4. It supports lightweight publish/subscribe communication based on a topic. This is quite similar to the publish/subscribe mechanisms in ROS and ROS2: messages have a data structure and are identified via a topic. The way to specify the data structure of a message is identical to ROS. A message compiler produces the required stubs for C++ (currently, only a C++ language binding is supported) which facilitate the creation of a producer and subscriber.

The main difference to ROS is that communication is only local (multi-threaded, but within a single system) and that only publish/subscribe is supported (no services and actions). However, it is possible to interact with ROS by using a bridge for ROS2 (but not ROS1) or DDS, as the data marshalling is the same as used by the RTPS (real-time publish/subscribe) protocol.

PX4 defines a large library of messages for the autopilot which (unfortunately) does not use the ROS standard messages except for primitive types.

ABI (Paparazzi):

The ABI software bus is the internal communication middleware used by the Paparazzi UAV system to exchange messages based on the publish/subscribe paradigm. Its main usage consists in sending sensors data from the low level drivers to the state estimation filters (AHRS, INS), but is also widely used by payload modules to retrieve or send data and commands.

The main ideas are:

- Give an easy way to allow software components to exchange data with minimum delay nor execution overhead
- The components do not need to know each other, they only need the message format
- Each subscriber sets a callback function that will be called when new data is sent

All the possible messages are described in an XML file that specifies the name and ID of the message, as well as the names and types of its fields. All standard C types and usual structures used by the Paparazzi math library are available:

```
<message name="DATA" id="0">
 <field name="a" type="float"/>
 <field name="b" type="struct Int32Vect3"/>
</message>
```

A code generator produces a C code from these definitions, which consists in callback prototypes, binding and sending functions.

Each sender has a unique ID. When binding, a subscriber can filter messages by selecting a specific source, accept all the messages or disable reception.

The implementation is focused on low overhead, so the callbacks are called within the sending functions (in the generated code), therefore this bus is not thread-safe. It is supposed to be used only inside the main autopilot thread and is independent of the underlying RTOS (if any).

The documentation and user guide are available online: https://paparazzi-uav.readthedocs.io/en/latest/developer_guide/abi.html

4 Existing Architecture Implementation

4.1 Drone Autopilots

UAV autopilot systems allow an unmanned aerial vehicle, such as a drone, to perform entire missions autonomously without the need for manual remote control. Operators use ground control stations to set the parameters of the mission and the UAV autopilot directs the drone or other unmanned craft to complete the task.

Among several open-source autopilots researched, four of them stood out: Ardupilot, PX4, Paparazzi and LibrePilot. These were the ones still in development and that had a focus on autonomy. They provide several versions depending on the type of underlying drone structure: rover, helicopter, quadcopter, hexacopter, plane, vertical take-off plane, etc. They are depicted in Figure 10, together with inactive projects (i.e., with no updates since February 2017) and software focused on first person view (FPV). Inactive projects are in black and white, while active ones are coloured in green. Also, when a project forks from another, it appears connected to the origin at its right.

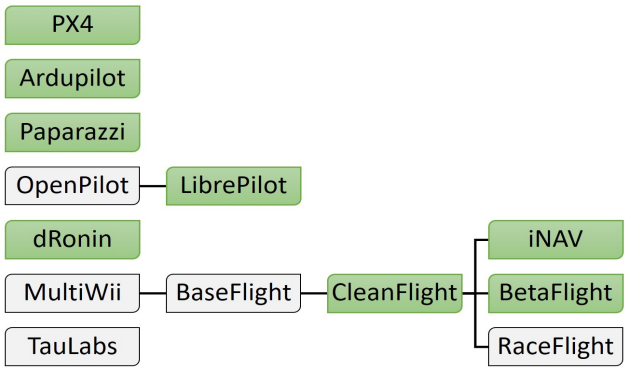


Figure 10: Autopilot projects evaluated during research

4.2 Ardupilot

From a task execution point of view, Ardupilot is based on a main loop that distinguishes critical functions from non-critical ones. The first piece of code to be executed, called the fast loop, regards the critical functions (critical motion control). Later, the time left for the non-critical tasks is distributed by an application-level scheduler. The scheduler executes the other tasks in a best-effort manner, according to priorities statically defined by the coders. This structure is represented in Figure 11.

In Pixhawk boards, the main loop is executed at 400 Hz, and at 100 Hz in APM 2.x boards. These values are defined statically. For each board it is deployed to, Ardupilot has a different structure for its tasks and a different set of priorities, in order to consider the specific implementation characteristics of each hardware. Nevertheless, the distinction in priority levels between critical tasks and complementary ones remains the same.

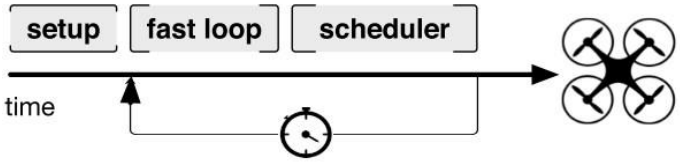


Figure 11: Ardupilot's control loop

From a functionality point of view, the code has different modes, each one containing a slightly different structure. Yet, every mode has three main stages:

Sensor data interpretation: At first, sensor data is filtered and translated to meaningful values from an engineer point of view (i.e., position, velocity and accelerations – linear and angular – and remote commands).

Controllers: The values interpreted from the sensors are compared to reference values or used as new references in control loops.

Actuator translation: The control outputs are translated to individual actuator (motor) commands.

An autonomous mode will have a simple attitude controller, that receives the targeted attitude (row, pitch and yaw), compares to the actual attitude, and then commands a certain change in the inclination of the vehicle, but also some control loops happening before, as seen on Figure 12: at first, the measured position is taken into account for updating the trajectory in the navigation control loop; then, this trajectory will determine a target position, which will be compared to the measured position in a position control loop in order to determine a target attitude; finally, once it is determined, the target attitude is used as an input for the same attitude control loop that exists in the manual mode.

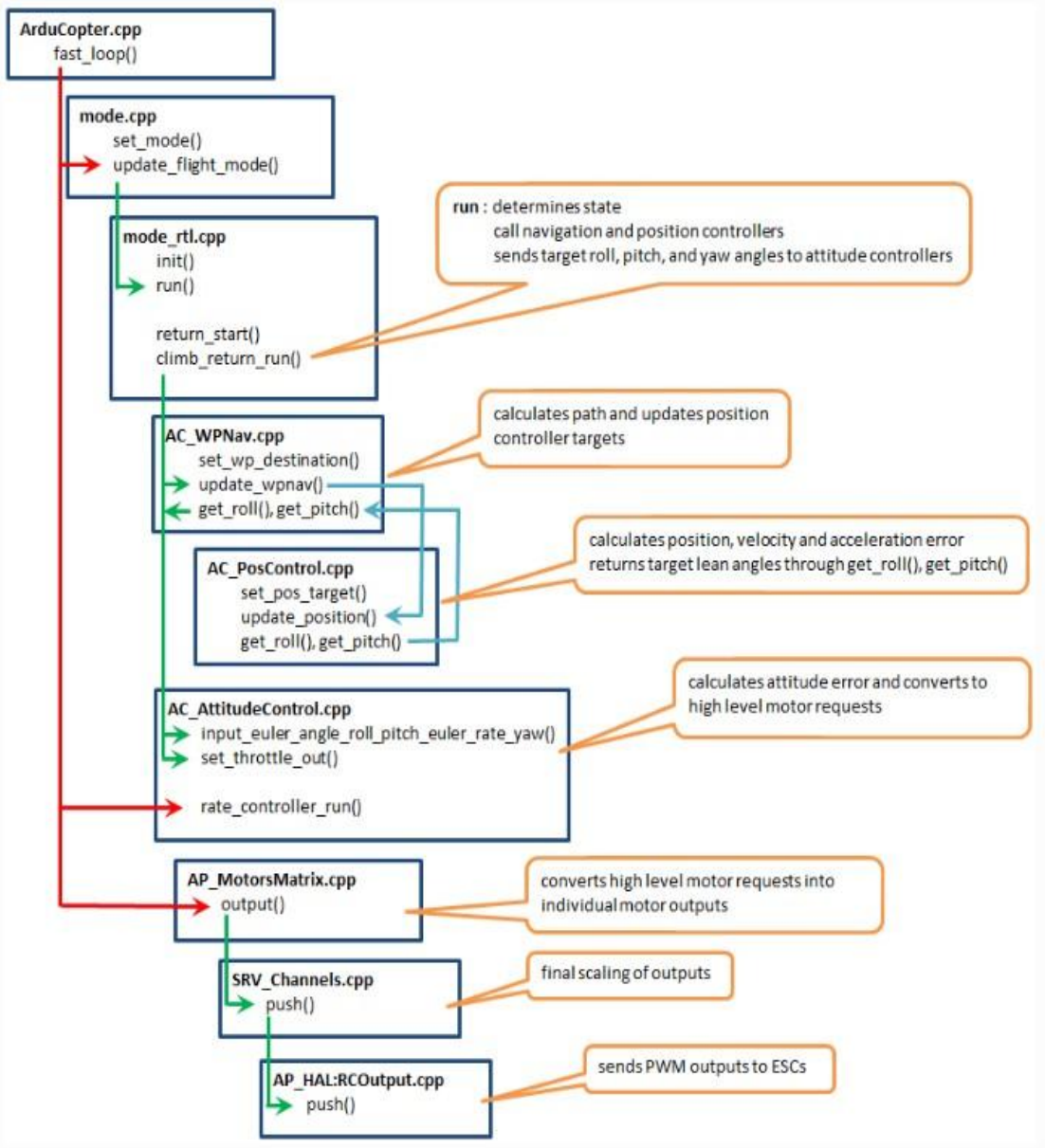


Figure 12: Ardupilot's autonomous control sequence

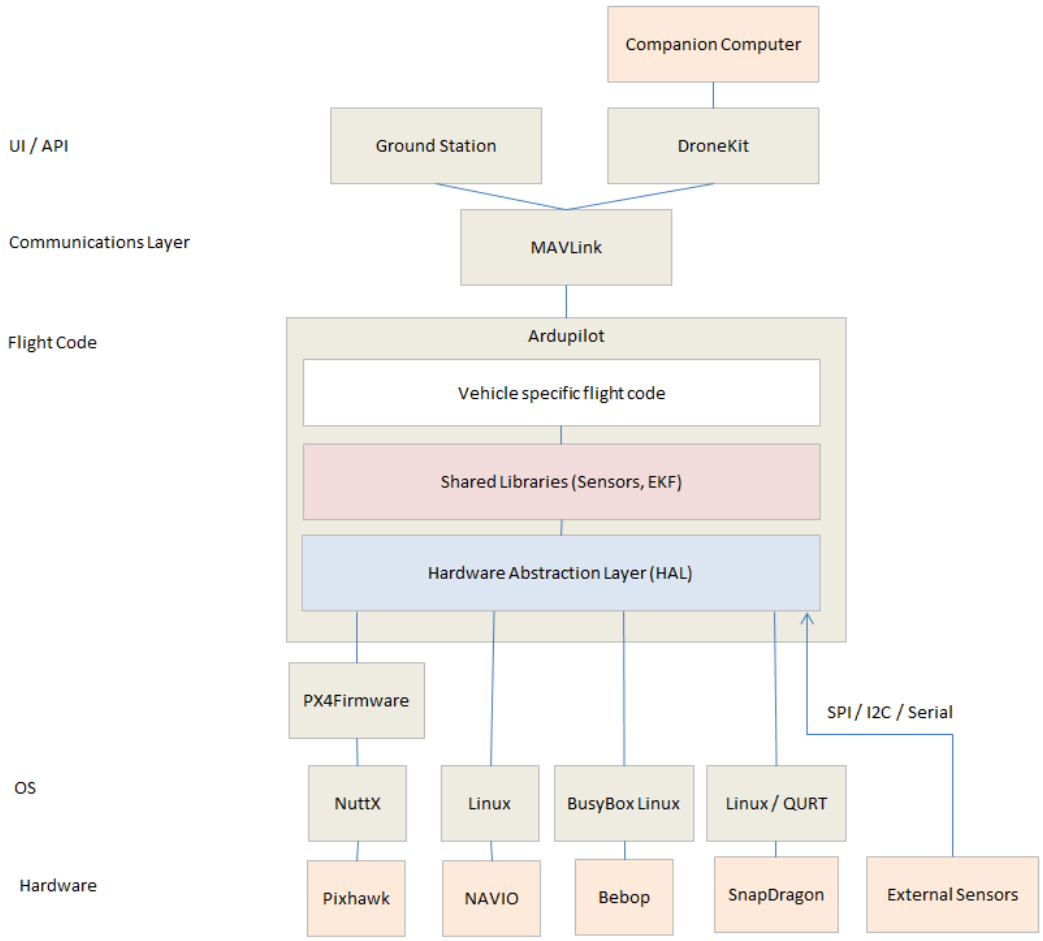


Figure 13: Ardupilot's modules

4.3 PX4

PX4 is composed of several modules as seen in Figure 14, with roughly each module representing a different parallel task. This makes the system's operations and communication fully parallelized, and its components can consume data from anywhere in a thread-safe fashion. This data is made available in the form of messages, which are transmitted between modules using the uORB middleware. The drivers and operating system are modelled after the POSIX interface standards, and the onboard networking is following the UAVCAN standard proposal. PX4 also offers a FastRTPS Bridge to enable the exchange of uORB messages between PX4 components and offboard Fast RTPS applications, including those built using the ROS2/ROS frameworks. All of this gives PX4 a great modularity, which means that, for developing a new module, only the topics it must interact with will matter to the developer. On the other hand, from a functionality point of view, it resembles that of Ardupilot. A diagram representing its behaviour is presented in Figure 15. Also, the development team considered the time constraints involved in each control loop, as seen in Figure 16.

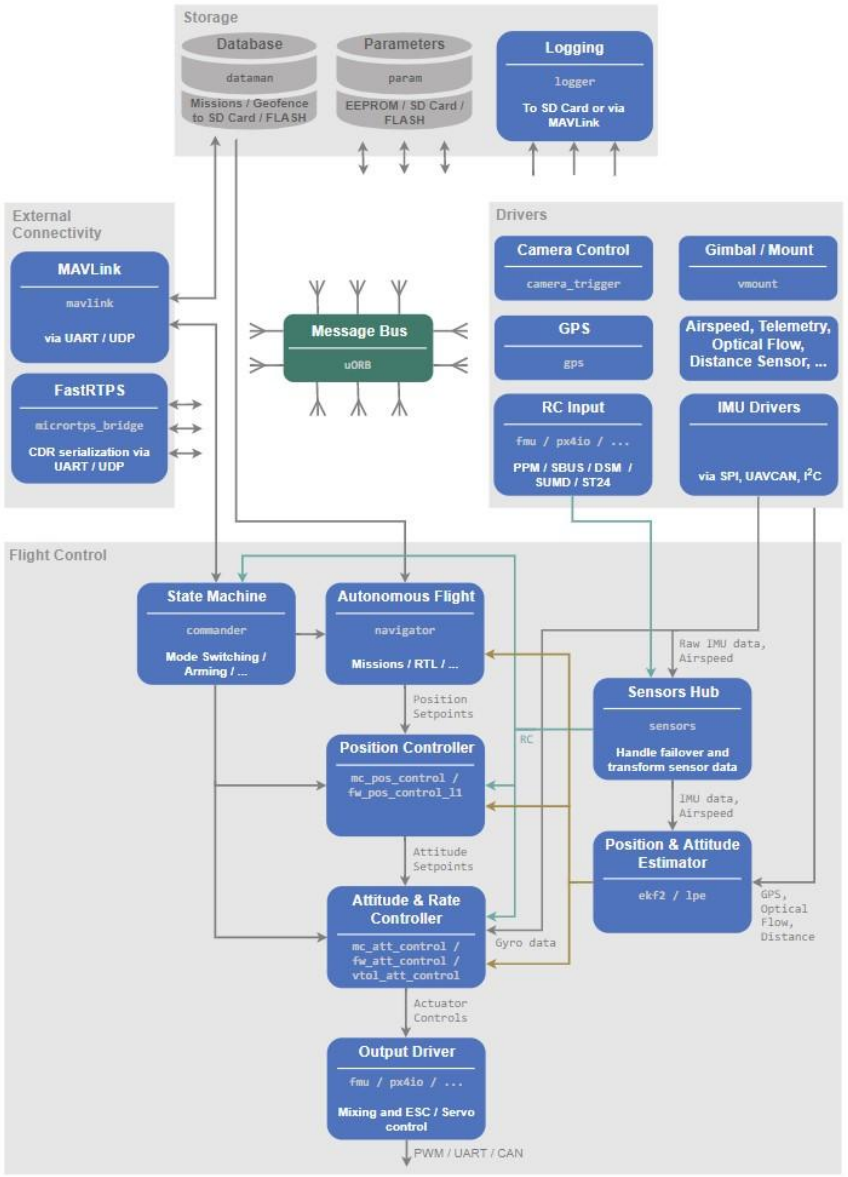


Figure 14: PX4 modules

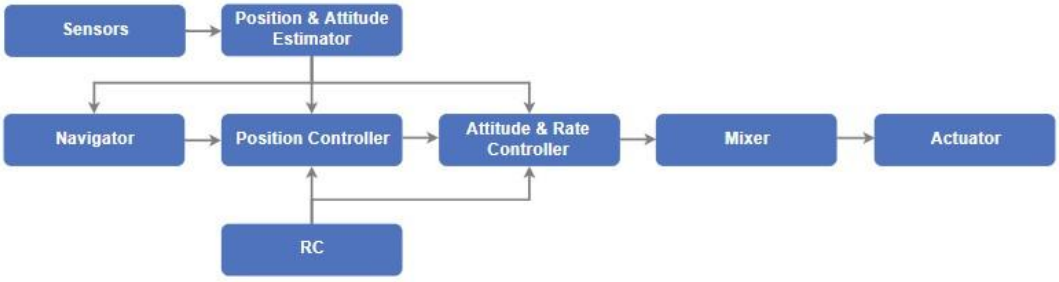


Figure 15: PX4 functionality structure

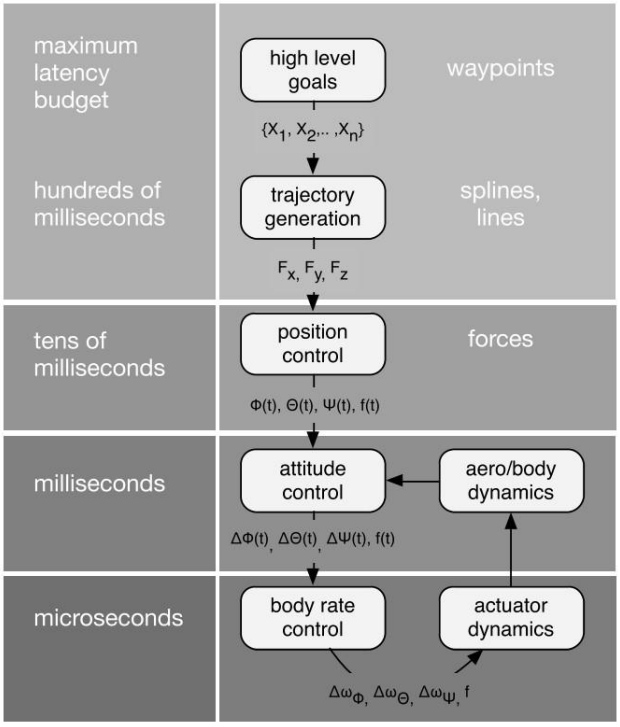


Figure 16: PX4 time constraints

4.4 LibrePilot

It is implemented using the FreeRTOS embedded real time operating system and a hardware abstraction layer named “PiOS” on top of the FreeRTOS base, as well as a custom scheduler for delayed callback functions with real-time scheduling guarantees.

The control loops, on the other hand, are different from the preceding ones. In Manual Control mode, the flight controls directly steer the actuators in an open loop. It is the pilot actuating the control transmitter that closes the loop using hand eye coordination or relying on aerodynamic flight stability. In Stabilized Control mode, the command from the receiver is fed into one of the two loops: in Rate mode, a single PID control loop is used for control; in Attitude mode, two cascaded PID loops are used where the outer loop controls the angular velocity setpoint based on attitude error, and the inner loop controls the actuator based on angular velocity errors. Finally, in control modes that involve the Autopilot, the craft is flown by the Flight Controller based on trajectory instructions.

The code is made of modules that can be callback functions or specific threads. The modules can only communicate using UAVObjects, a type of data previously defined. This gives the code a lot of modularity, but the modules are not self-contained – one still must know the code of the interacting modules to develop a function to interact with them.

4.5 Paparazzi

The autopilots proposed have a list of built-in modes, covering most of the usual needs. A certain control stack is called for each mode, and although it is possible to choose the control loop being used at the moment. However, an experimental mode allows the implementation of a custom autopilot state machine. It is then possible to change or extend the number of modes, or even run several parallel control loops. See http://wiki.paparazziuav.org/wiki/Autopilot_generation for more details.

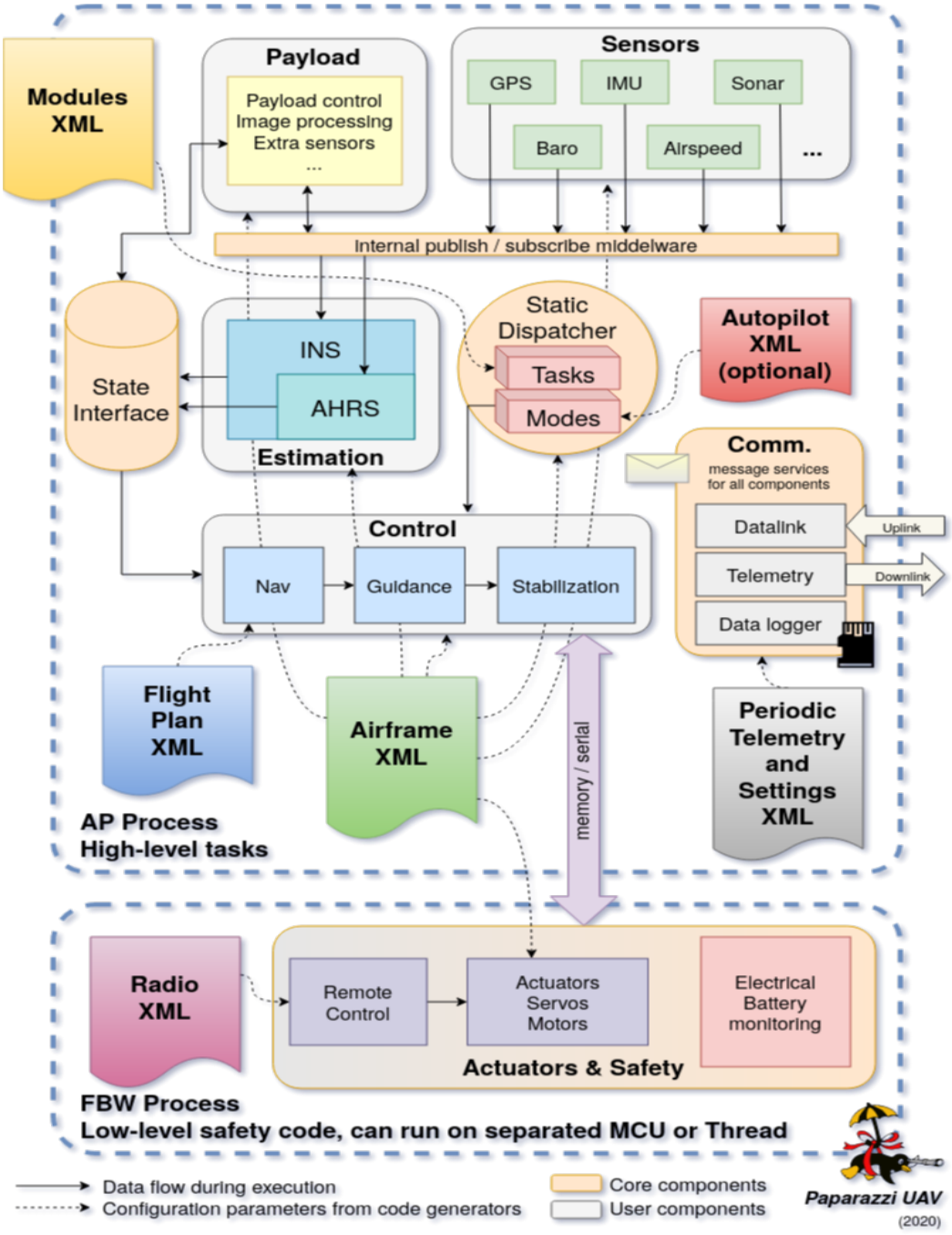


Figure 17: Paparazzi on-board architecture

These components are separated between critical (such as AHRS, IMU, INS and GPS) and non-critical, and the act of calling them is done in the same way as Ardupilot, with statically defined periods and in a sequential order. Although, efforts are being made to transition to a real-time operating system (RTOS) based solution. The latest hardware is based on ChibiOS, an efficient and light RTOS for microcontrollers. The main part of the autopilot is handled in a single thread, but most of the low-level drivers have their own threads, with either high priority (communication with internal sensors) or low priority (SD logging) compared to the autopilot task. It is also possible to create threads for certain heavy tasks related to payload management or parameter estimation in dedicated modules. This is ensuring a proper timing for the core autopilot control stack.

In addition, the data between the sensors and the estimation filters are managed by a publisher/subscriber mechanism, where the estimated state is published in a blackboard type interface for the control and navigation loops. Conversion between the different attitude or position representations (euler, DCM, quat, lat/lon, ecef, …) are automatically handled by this state interface.

The control loops, just like PX4, resemble that of ArduPilot, with a core attitude control loop, and the possibility of other control loops to execute on top of this one, with slower rates. However, the navigation layer is based on a specific flight plan language (based on XML files generating C code) allowing complex mission description (http://wiki.paparazziuav.org/wiki/Flight_Plans). It is also possible to use a dynamic task based approach (http://wiki.paparazziuav.org/wiki/Mission). Both approaches are based on built-in flight patterns like “go to point”, “fly segment” or “around circle”, but can be extended with custom navigation patterns.

Paparazzi provides a simplified way to add modules without having to know the functioning of other modules, by using standardized XML file descriptions. The developer still needs to know how functions are called in other modules so that it can interact with them. It is also possible to create interaction between modules by using the internal publish/subscribe software bus.

5 Conceptual Modelling of the Reference Architecture

5.1 Rationale

When thinking of an implementation of an Integrated and Modular Architecture for Drones, two different approaches could be followed. An approach that could come to mind is to focus on implementing a new autopilot from scratch. We believe that this approach is not sustainable and would not catch the interest of the domain stakeholders. The reasons for the non-sustainability of the approach are multiple:

- Existing off-the-shelf autopilots, such as Paparrazi, PX4 or Ardupilot for example have been developed, and tested, hundreds of thousands of hours thanks to hundreds of contributors and users. Developing from scratch a new autopilot would require such a huge effort and create a useless concurrence with existing autopilots.
- New embeddable systems on chips, sensors, actuators, operating system versions, that could be used to run an autopilot are released on a weekly basis, and porting an existing autopilot to new platforms, and new sensors or actuators, also requires a large community of contributors to make a software product living and scalable.

The risk of obsolescence of such an approach is therefore too high, and the chances for adoption too low, for this approach to be chosen.

Most C4D stakeholders are interested in integrating as effortlessly and safely as possible their specific components into several existing autopilots. By effortlessly, we mean that when designing a new custom component, that we call Enabling technology in C4D, (new sensor, new functionality, new hardware component, etc.), a stakeholder should not have to develop a complete expertise of the target autopilot. The complete code, often written in C/C++, should not have to be mastered, to be able to integrate a custom component. This, for several reasons: safety problems, cost in time and resources, problems with future evolutions of the autopilots and versioning. When integrating a custom component, the stakeholder should be able to quickly identify by what means the component can be integrated. He should be able to see how it can gather its inputs, how its outputs can impact the behaviour of the autopilot, and these questions are highly related to software buses. During this process, the possible occurrences of race conditions should also be clearly visible. By safely, we mean that the process should be highly supported by analysis tools, able to help in detecting any potential functional or non-functional flaw, by checking functional and non-functional properties.

A graphic approach is often chosen in systems engineering to reduce time and efforts spent, and more and more analysis tools require models of systems as input expressed with Model-Based System Engineering (MBSE). C4D, in WP6, is also gathering and developing a lot of analysis tools, most of them taking models conforming to a meta-model as input.

Therefore, the implementation that we propose is based on a meta-model, “implementing” the specification that was obtained in C4D D3.2 deliverable, able to represent existing autopilots in such a way that integrating a new custom software or hardware component should require low effort, while also allowing analysis regarding performance, security, energy, software safety, etc. This approach is by far less effort consuming than developing a new autopilot while reducing the risks of non-adoption.

The meta-model shall therefore be able to represent any existing autopilot (usually coded in C/C++), while also showing the link between the code and the building blocks defined in C4D D3.2.

The addressed concepts, the underlying rationale, the goals of the effort, should be kept in mind, in order to choose the useful modelling artefacts, and to avoid useless constructions that would just participate in obfuscating the model. Therefore, the main question that we should address is “what is an implementation?”

When considering the specification of an autopilot as building blocks, such as in Figure 1, these building blocks could be considered as functions, decomposed in sub-functions such as in Figure 2. If we had infinite processing power, infinite energy to power it, infinite bus and networks bandwidth, infinite memory, then every such function could be executed continuously, and that would be it, the autopilot would be the continuous execution of these functions, executed instantaneously, following a data-flow semantics. Of course, in real life, resources are limited in size and speed, and functions, or information passing on a bus or network, take time and space. Moreover, resources are limited and often shared, and thus arbitrated, among entities. We claim that the main role of an implementation, compared to the specification, is to assign a rhythm to functions, that can no longer be executed continuously, such that they can be allocated to entities which take part in a software/hardware architecture (threads, interrupt service routines, processes, partitions, OS, CPU, etc.) such as typically described and modelled with an Architecture Description Language (ADL), such as AADL or UML MARTE. It is then not a surprise to realize that most of the useful concepts that we could find in an autopilot implementation model are present in several ADLs.

Nevertheless, while studying the software/hardware architecture of autopilots, which often rely on a monolithic core, we could find some constructs that are not present in aforementioned ADLs, and that should be represented. Therefore, this section relies on AADL and UML MARTE for referencing common ADL concepts and discusses in detail the semantics of specificities that are characteristic of autopilot software/hardware architecture. The choice of AADL and MARTE is consistent with the ideas of implementations that C4D partners foresee. In order to favour adoption and future use of the C4D implementation concepts, the current C4D implementation targets several future implementations within existing frameworks as extensions. The possible foreseen extensions of frameworks are shown in Figure 18.

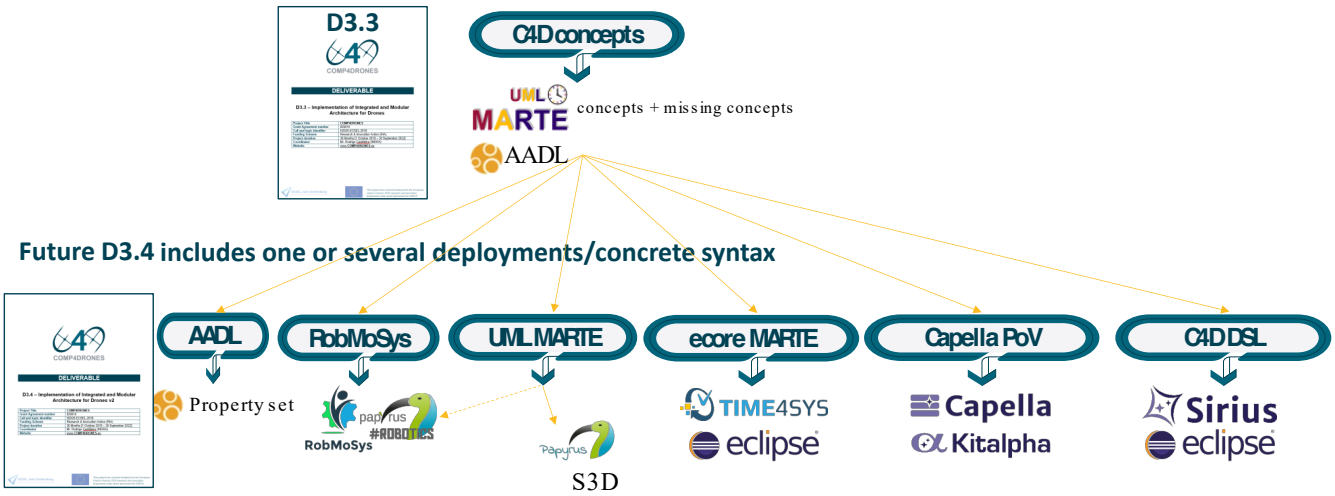


Figure 18 : Possible implementations in frameworks of the C4D proposed implementation concepts

The current document stresses out particularities of autopilot implementations, which may be absent from either AADL or UML MARTE. During the RP3 of the project, these missing concepts will be introduced in several frameworks in order to extend their semantics such that precise autopilot implementations can be expressed. Possible and interesting implementation could be provided within:

- A specific AADL extension, using, for example, property sets.
- An extension of Papyrus for Robotics, a tool developed within the RobMoSys ecosystem, which is using UML MARTE concepts as base concepts for the hardware/software architecture.
- S3D, which is a framework based on an UML MARTE extension.
- Time4Sys, a partial eCore based implementation of UML MARTE, focusing on performance analysis.

- A Capella Physical Point of View offering AADL concepts augmented with specific C4D concepts.
- A specific DSL.

The main advantage of targeting several frameworks is that each of them comes with either a specific analysis or architecture exploration tools, or even with an ecosystem, including other points of views or tools, allowing different functional and non-functional properties to be addressed.

As a result, the goal of this chapter is to describe the necessary or interesting concepts which are useful to describe an autopilot implementation, using as a basis an intersection of well-known concepts in existing ADLs such as AADL and UML MARTE. This will allow an easier integration in the target frameworks/languages which are, for the majority, either MARTE based or AADL based. Note that a rich mapping between both AADL and UML MARTE is presented in Annex A of the 2nd version of the UML MARTE profile as defined by the OMG. Another discussion about comparative semantics between AADL and UML MARTE can be found in [12].

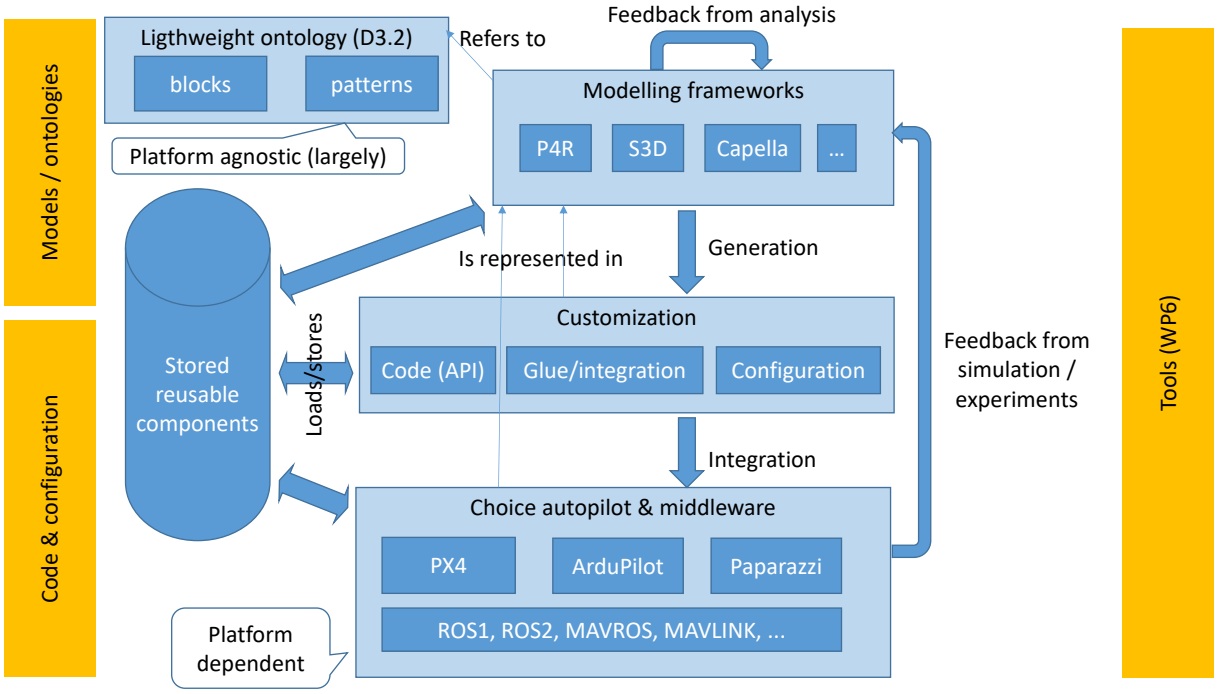


Figure 19: General view of C4D workflow

On Figure 19, a view of the C4D workflow is represented. As introduced in the deliverable D3.2, building blocks and patterns allow to define a **lightweight ontology**. This ontology was already useful during the discussions with the partners since its definition allowed faster mutual comprehension. As an example, enabling technologies providers (see Section 6) were able, by themselves, to identify in what building block their component was to be inserted. It must be noted that the choice of working with an ontology allows for mutual understanding between the stakeholders at every level: from the understanding of a specification or an implementation to clearly formulate and decompose requirements.

What we could observe when trying to model existing autopilot with existing embedded systems related modelling languages is that some semantics are lacking or could benefit from a better readability. One of the most representative characteristics of autopilots is its very monolithic implementations. As a result, we considered existing **modelling frameworks** and propose in this section to enrich their semantics for a better representation of autopilots. Of course, we are not starting from scratch, since a lot of the **tools** (collected in WP6) are clearly based on MBSE and require input models conforming to the meta models

implemented within modelling frameworks. There is for now a gap between the world of autopilots coders, who work in code (C, C++ mainly), and the world of MBSE tools. The major goal of this section is to show what modelling artefacts would help to capture the behaviour of autopilots code, in order to fill the gap between the code and the models, and for existing autopilots (PX4, Paparazzi, etc.) to be abstracted to a model to be represented in a modelling framework. For this purpose, a fine grain representation may be necessary for some types of actions, such as **customization** through the addition of a custom component or analysis. The modelling effort will be important but can be capitalized since the obtained model can be stored for future use and transformed using model transformation to target other frameworks.

Several interesting operations can be done once an autopilot is represented in a framework, and, depending on the framework, several functional and non-functional properties can be addressed. The autopilot, along with custom components, can beneficiate from **feedback from analysis**, in order to either validate its functional or non-functional behaviour or simply to point out potential flaws (e.g., violated performance constraints). This part, based on analysis tools, will allow to toughen existing autopilots, and to address some of the objectives defined, e.g., in the DO-178C standard. This feedback can also be given through architecture exploration or helping tools. Several choices are handcrafted today in autopilots configurations, that would greatly benefit from the help of automated tools. Finally, simulation and experiments will also be used to get **feedback** and take corrective actions.

Another point that can be addressed by tools is the identification of the practical way a custom component can be integrated to an autopilot. This part is nowadays a very high hurdle for drone manufacturers, and this part is always a risk, even on an a priori safe autopilot. Without tools helps, side-effects can occur, that can jeopardize the whole system. Some tools developed in the last phase of C4D will therefore address this problem of generation of glue code for custom components to interact safely with off-the-shelf autopilots. Since autopilots are usually configured specifically for different platforms, this tedious phase also needs to be tool assisted. In any of these cases, autopilots providers will also beneficiate greatly from realizing how well, or not, their autopilot allows fast and safe customization, and take corrective actions in future versions.

A code and model repository would help, at every level, to capitalize on a mutual effort. At this stage, one can wonder if defining a pivot language could be helpful, but the problem of such an approach is that a pivot language is either a “least common multiple (LCM)” or a “greatest common divisor (GCD)” of the possible uses of the language. If it is a GCD, semantics required in one language are lacking, and it is lost during model transformation between a source language and the pivot language. If it is an LCM, it is very hard to define without inconsistency, and the potential users are driven back by the complexity. Moreover, since model transformation between UML MARTE and AADL is clearly defined already in UML MARTE standard, we believe that model transformation allows a model of an autopilot to be defined and transformed from a framework to another if the extra semantics inherent to autopilots is clearly defined. This is therefore the object of this section.

5.2 Concepts

5.2.1 Notion of ontology

The C4D D3.2 deliverable (see Section 2.2 for an overview) defined “building blocks” for autopilots. These blocks can be characterized by a lightweight ontology of UAS and shed a useful light on components of an implementation. The architecture description language is refined with information from this ontology. For instance, with respect to threading aspects, actual autopilot implementations mix within the same threads, for the sake of performance, several functions participating in different “building blocks”. This tends to complexify the implementation compared to the specification. It is therefore important to be able to identify, for each modelling artefact, which building block it belongs to. A lightweight ontology is defining a hierarchy of concepts connected by “general” associations and not by strict formal definitions. The implementation model shall therefore integrate concepts related to

lightweight ontology definition as well as ontological annotations. In general, the additional information improves the quality of analysis and code generation mechanisms.

A systematic state of the art on ontology-based systems engineering is given in [13]. In the context of a dataflow, a part of the lightweight ontology can be seen as a dataflow graph, interfaces can be represented as abstract concepts, but could be refined in long term use. For example, for building blocks communicating a position, the position could have several natures (absolute, relative to a point or another, a mix of both, expressed in a coordinate system or another, etc.).

In case of C4D, it is important to consider existing ontology standards for the robotics domain to foster adoption and reduce the effort (i.e. the same argument as choosing a standard autopilot instead of developing or own). The Core Ontology for Robotics and Automation (CORA) is an ontology developed in the context of the IEEE Ontologies for Robotics and Automation Working Group and CORA ensures that consistency is maintained. The main concepts shown in left part of Figure 20 focus on conceptualization of subdomains such as robot, robotic system ad robot part into a larger domain.

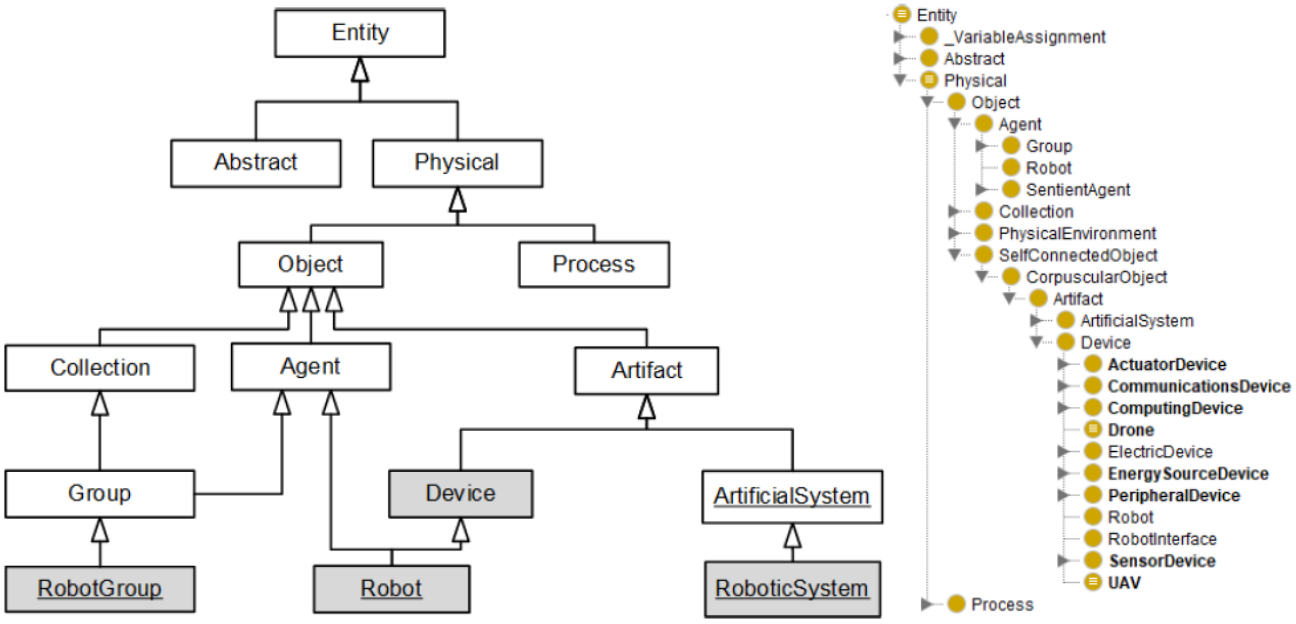


Figure 20: (a) Main CORA abstraction concepts (underlined), (b) integrated drone parts ontology

In order to obtain a drone-specific ontology (or “dronetology”), we propose to capture properties of the physical components defined with it. The ontology is extended with 6 types of devices: ActuatorDevice, Communications Device, ComputingDevice, EnergySourceDevice, SensorDevice and PeripheralDevice. In the right part of Figure 20, an example of the classification in a taxonomy of the drone parts integrated with CORA is shown. This ontology enables the tailored modeling of drone aspects.

The use of an ontology is tool agnostic but requires tool specific extensions. In case of Papyrus for Robotics, the elements are currently mapped from a domain specific ontology to elements of the UML meta-model (available in form of an ontology) and create a corresponding UML profile (future version will have a more direct ontology support). After that, a system can apply the profile created from the ontology in addition to the robotic model already present. Compared to the existing system model, additional information can be used to extend/refine the existing code generation or provide additional analysis capabilities. These capabilities are based either on the UML model applying the profiles or on an ontology that can be generated from the UML system model

Table 3: Ontological concept

Concept	Ontological annotations
Recommendation	Advisory – Additional information, link with standard ontologies
AADL related concept(s)	Structural and behavioural aspects
UML MARTE related concept(s)	Structural and behavioural aspects

5.2.2 Layered decomposition

According to the provided information from the C4D partners, it becomes clear that distinct components often fit in different abstraction layers of a drone system: some components regard pieces of hardware, such as multi-spectral cameras, while others abstractly describe control algorithms that could be implemented in different coding languages. Hence, we need to distinguish between different abstraction layers, which requires to firstly define these layers, represented in Figure 21.

Figure 21 : Layered representation of the drone SW model

5.2.3 Concepts requiring specific adjustments

5.2.3.1 Functions

A function is a programmatically implementation of a mathematical application, typically injective (determinist), it can be seen as an entity able to transform and produce output data from some input data. A function is usually represented at least by a name.

In the most abstract layer, one can think of functions that are executed by the drone. A single function can be implemented in different programming languages and using different algorithms.

Table 4: Function concept

Concept	Function
Recommendation	Required
AADL related concept(s)	Subprogram
UML MARTE related concept(s)	Step and/or UML operation

Note that in UML MARTE, Step is defined in the GQAM (Generic Quantitative Analysis Modelling) package. It is used as part of a sequence in a scenario and requires a host (SchedulableResource) association.

Inputs and outputs:

A function is characterized by a set of typed inputs and typed outputs. It is not a surprise since the modelling artefacts can represent software sub-programs, which are characterized by their signature (i.e., in and out typed variables). We can note that while some languages, such as C and most programming languages, represent differently the return value of a function, compared to other output variables, it is not necessary to make such a distinction in a model, the choice between the return value and the output parameters being purely linked to low level choices.

Table 5: Function inputs and outputs concept

Concept	Function inputs and outputs
Recommendation	Required
AADL related concept(s)	Subprogram parameters
UML MARTE related concept(s)	UML parameters

Communication between functions within the same thread:

As presented in Section 3.3, communications between functions are very diverse. Within the same thread, they usually can take three different forms: communication through local variables (i.e., thread-level), communication through global variables (i.e., process-level) or using a software bus which can abstract the communication means. In order to avoid race conditions, it is important for any implementation using anything else than local variables to express if variables are protected against race conditions (semaphore, Hoare monitor) or not. Since a lot of autopilot implementations are monolithic and are programmed to minimize CPU and memory footprints, unprotected global variables are very often used. It is a way for them to avoid synchronization mechanism overhead (acquire and release semaphore), and to minimize memory footprint since subprograms accessing local variables require data to be duplicated within several local threads' stacks. As a result, adding any parallel access to these unprotected global variables can raise race conditions, while dozens of these unprotected variables are used in an implementation. Communication between functions within different threads or processes or platforms is addressed within the corresponding layer description.

Table 6: Function inputs and outputs concept

Concept	Function inputs and outputs
Recommendation	Required
AADL related concept(s)	Parameter Connections
UML MARTE related concept(s)	Object flow on UML activity diagram
C4D specific notes	Often implemented through unprotected global variables in a monolithic implementation

Hierarchical decomposition:

In several design methods for embedded systems (e.g., Arcadia), functions can be decomposed hierarchically in sub-functions, themselves being decomposed in sub-functions, etc. It is very useful when designing a system, since it corresponds to the classic divide and conquer strategy used when programming, nevertheless, it is not mandatory for a language addressing a method where an existing system is represented.

Table 7: Function hierarchy concept

Concept	Hierarchical decomposition of functions
Recommendation	Advisory
AADL related concept(s)	N/A
UML MARTE related concept(s)	N/A or using the concepts of components and sub-components

Function logics:

Depending on the needs addressed by the target language, the language may or may not allow to express the function logic. Several languages allow a function logic to be expressed in a specific formalism such as UML state machines, or UML sequence diagram, other specific languages or even directly programming language code (like it is possible in AADL or in S3D). Some formalisms are adequate to express some logics, for example an UML state machine is very convenient to express how a function behaves depending on a discrete set of states, while it is less convenient to express the behaviour of a function responsible for the drone's stability, which can be based on a PID or INDI controller for example. Function logics are interesting to be expressed in the goal of checking functional properties.

Table 8: Function logic concept

Concept	Function logics
Recommendation	Advisory
AADL related concept(s)	State machines and Modes
UML MARTE related concept(s)	UML State machines, UML activity diagram, other behavioural UML diagrams

5.2.3.2 Threads

A function must be executed in a thread of execution. A thread can be an OS object that executes a certain algorithm and maintains its context data. A thread of execution can also be triggered by an interrupt (Interrupt Service Routine - ISR) or even be executed by a hardware dedicated circuit. Depending on the underlying execution platform, multiple threads can be executed in parallel, either in the same memory space (called a process) or in different memory spaces. This implementation has considerable impact on the performance of the system impacting its real-time properties (delays, response times, etc.), and therefore must be considered.

Each thread is a sequential execution that has its local memory space (note that in the case of an ISR, the memory space can be shared with the operating system).

Table 9: Thread concept

Concept	Threads
Recommendation	Required
AADL related concept(s)	Thread
UML MARTE related concept(s)	swSchedulableResource, SaScenario, WorkloadEvent, etc.

Process or system level thread scheduling:

In a single process OS, threads are scheduled at the system level, while in a multi-process OS, such as Linux, conforming to the POSIX 1003.1 standard for the profile PSE 53 or 54, threads can be scheduled hierarchically within their owning process, or be scheduled directly at the system level. In most cases, a thread is characterized by a static priority, and the scheduler is based on these priorities. When there is one core, it is simply the ready thread with the highest priority which is elected for execution. Some more dynamic schedulers can also exist in some OSs, such as Earliest Deadline First, which is giving the highest priority to the most urgent ready thread. On a multicore system with *n* cores, it is more complex. If the scheduler is global, then the *n* highest priority threads are executed on a core, which theoretically allows to use every core at the same time, but also generates a lot of overheads due to thread migrations between cores. If the scheduler is partitioned, each thread is assigned a core, and each core is executed like in the single core case. In some hybrid cases (like it is the case in Linux), some balancing functions may re-assign threads to different cores in case of an unbalanced core assignment. The concepts allowing to represent thread scheduling are already present within other ADLs, and autopilots do not differ from other embedded systems at this level.

Table 10: Scheduling

Concept	Scheduling
Recommendation	Required
AADL related concept(s)	DispatchProtocol, etc.
UML MARTE related concept(s)	computingResource, scheduler

Inter-thread synchronization and communication:

Most operating systems supporting multi-threading offer several types of inter-thread communication means. One of the most powerful set of communication means is provided by the POSIX 1003.1 pthread standard, and includes tools to program communication following the following paradigms:

- Blackboard (asynchronous) communication: using a global variable, protected by mutual exclusion semaphores, that can use priority inheritance protocols to avoid priority inversion.
- Message queue (loosely synchronous) communication: a message queue is used to force a consumer thread to wait for, and be triggered by the data left by a producer in the queue (producer/consumer paradigm).
- Other synchronization mechanisms such as: barriers, events, or rendezvous which are synchronous and may make the performance analysis impossible [10].

Table 11: Inter-thread synchronization and communication

Concept	Threads
Recommendation	Required
AADL related concept(s)	Data port, event port, event data port and port connections
UML MARTE related concept(s)	SwCommunicationResource, SwMutualExclusionResource, FlowPort, MsgPort

Thread-level static scheduler:

A specificity that C4D could observe in off-the-shelf autopilots is that within its monolithic implementation, a central thread in the autopilot possesses its own internal scheduler. This thread acts like a non-preemptive cyclic executive and executes its internal functions according to (1) a static scheduling table, (2) the internal state of the autopilot. One should note that the process of generating the static schedule is tedious, and usually hand-crafted or automatically generated following rules- of-thumbs - based algorithms when customizing the autopilot. To the best of our knowledge, while non-preemptive cyclic executives used to be widely used in the 1990's, most MBSE methods addressing Cyber-Physical Systems do not provide any means to specify a static scheduler at the thread level. The most important functions related to perception, flight guidance, flight control and actuation, as well as health management, are executed within such a non-preemptive cyclic executive hosted by a thread. Therefore, we believe that this specificity must be addressed by the model, since these functions represent the core of the autopilot.

Another observation made in off-the-shelf autopilots showed that two implementation versions of the non-preemptive cyclic executive exist:

- On bare- metal microcontrollers, the main thread is executed as fast as possible, within a non-timed loop. As a result, from an iteration to the next, the time spent is variable and depends on the functions executed in the iteration. It is therefore not possible to tell how many iterations are required to insure for example, that a function will be executed every 10 milliseconds. In this kind of setup, the periodicity of the functions executed within the cyclic executive is insured by comparing the actual clock with the value of the clock that must be reached for 10 milliseconds to have elapsed since the previous execution of the function. If the actual clock is greater than the expected clock to release "periodically" the function, then the function is executed in the current iteration. Of course, without prior analysis, no one can tell from how much the actual clock can be late compared to the target clock for running the function.
- On microprocessors and microcontrollers hosting an operating system which is running the autopilot, the cyclic executive itself is implemented as a periodic thread (usually simply the main

thread), with a period of the order of 1 millisecond. In this case, an integer can be used to count the elapsed periods modulo the period of the function, such as a function with a period of 10 milliseconds will be executed every 10 iterations of the cyclic executive (which is, in this case, more of a periodic executive). Like for the case of bare metal microcontrollers, no one can tell, unless an analysis is made, what is the maximum lateness that the release of a function can suffers.

In both cases, some events can occur when some ISR was triggered and raised a flag (e.g., if some sensor data arrived). A function is executed within the iteration if the event occurred, adding some difficulty to characterize the execution of an iteration. An iteration can, therefore, depending on function periods, execute some functions, and also depending on some (typically external) events execute some other functions.

This kind of implementation does not conform to usual real-time systems standards, nevertheless, it allows a very small memory footprint (there is no copy of variables), and a very small processor utilization (no overhead due to thread synchronization, copy of variables, scheduling, etc.). Moreover, the monolithic aspect of this implementation allows a system that can change its mode (waypoint navigation, return to base, landing, etc.) at any time to make sure that every function within its core has a consistent behaviour with the current mode. Indeed, in a multitask implementation of such behaviour, a mode could be changed by a thread in the system and be perceived later by another thread. In this mono-threaded implementation, if the current mode is computed at the end or at the beginning of each cyclic executive iteration, then every subsequent function works in the same mode, and consistency problems (e.g., a function behaving assuming a mode, while another function assumes another mode) cannot occur.

When two functions communicate while being executed in the same thread, the implementation of the functional exchange can either use a local variable (which is always thread-safe[7]), or a global variable. In Paparazzi, most functions communicate through global variables. Communicating through global variables is not thread safe unless synchronization mechanisms (such as semaphores or monitors) are used.

Extending an existing autopilot by adding new threads can generate race conditions if accessing global variables used in another thread, therefore the implementation model shall clearly represent how internal functions communicate.

Figure 22 represents, in an AADL-inspired representation of such a pattern: a thread of period one millisecond hosts two functions, F_1, of period 3 milliseconds, and F_2 of period 5 milliseconds. They communicate through an unprotected global variable.

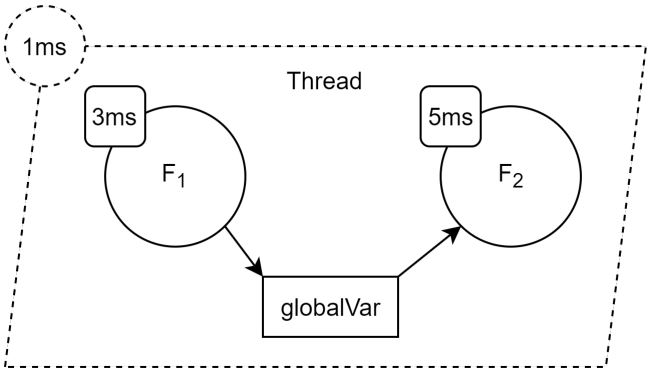


Figure 22: Functions scheduled by a periodic thread

[7] An operation, e.g., a data access, is thread-safe if its execution cannot generate a race condition.

Figure 23 represents two threads: on the right-hand side, a cyclic thread schedules two internal functions. F_5 is a function that is to be executed when the flag is set by the left-hand thread, while F_4 is a function to be executed every 5 ms. The symbol "B" in a circle means that the task is a background task, i.e., a cyclic task. The symbol "S:20ms" represents a sporadic activation of the left-hand thread, which will be triggered by the interrupt service routine (or can be directly the ISR itself) when some event occurs (e.g., a frame arrives on a network, or a byte on a serial bus, etc.).

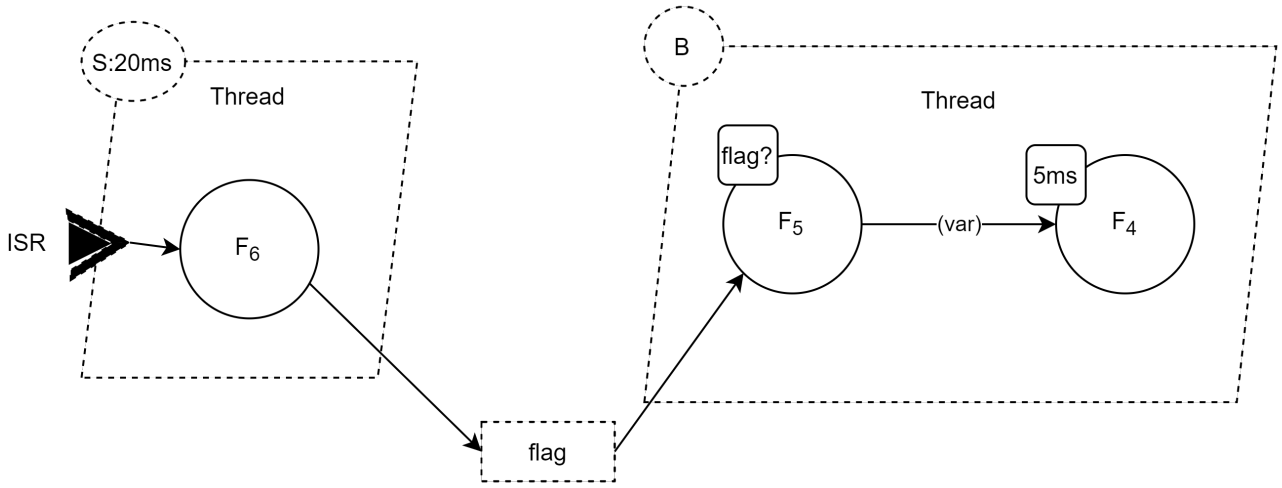


Figure 23: Functions scheduled by a cyclic thread

Table 12: Thread scheduler

Concept	Thread-level static scheduler
Recommendation	Advisory
AADL related concept(s)	N/A
UML MARTE related concept(s)	swSchedulableResource with its properties isStaticSchedulingFeature set to true, isPreemptable set to false, and timeSliceElements representing the static schedule. Nevertheless, the conditional functions (such as F_5 in Figure 23) cannot be represented.

Multiperiodic precedence constraints

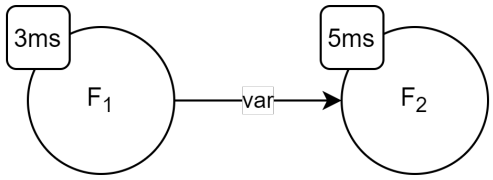


Figure 24: Representation of two communicating functions

A precedence constraint between functions represents the need for a function to be executed before the other. When functions are executed at the same rhythm in the same thread, the sequential execution is enough to ensure the precedence constraint. Nevertheless, it is not that trivial when functions have different periods: in this case, we talk about multiperiodic precedence constraints. Most ADLs propose the notion of threads, but to the best of our knowledge, few allow (1) a fine grain representation of function calls within a thread, and (2) multiperiodic communication constraints, as expressed between two functions in Figure 24.

In Figure 24, the two functions F_1 and F_2 are hosted by a thread, using an internal scheduler to execute them periodically at their respective period. Over the Least Common Multiple (LCM) of the periods of

the functions, which is 15 ms, F_1 will be released at times 0, 3, 6, 9, 12 milliseconds, while F_2 will be released at times 0, 5, and 10 ms. We will call each release of a function a job. It may be important to control which of the jobs of F_1 precede which jobs of F_2. In the literature, different multiperiodic precedence constraints have been proposed, their role is to specify which jobs of the preceding task precedes which jobs of the preceded task. The precedence model called Semaphore Precedence Constraints (SPC) which is proposed as an AADL extension in [14] is the most flexible way to express multiperiodic precedence constraints while using a polynomial size (in fact, only one integer) representation.

A simple way to understand a SPC is to consider the Petri net pattern represented on Figure 25. The transitions (black flat rectangles), from left to right, represent the execution of the functions F_1 and F_2 of Figure 24. We do not represent the part of the Petri net before and after each of the transitions making the functions able to execute at their respective periods (respectively 3 and 5 ms). F_1 sending a message is represented by the production by the left-hand side transition, of 3 tokens in the central place. When the right-hand side transition can fire, it can only do so if the central place holds at least 5 tokens, which are consumed when this transition fires (i.e., when F_2 consumes some data produced by F_1). During each LCM of the periods of the communicating tasks, there are as many produced tokens as consumed tokens. H here, there are 3 tokens generated per execution, and over a LCM of the periods there are 15/3=5 executions, giving, leading to 3*5=15 produced tokens, and 5 times 15/5=15 consumed tokens. The role of the initial marking is to tell which job precedes which. With a null initial marking, for each LCM of the periods, their first job of F_2 can be executed only after two executions of F_1, because F_1 produces 3 tokens, while F_2 consumes 5. Then, 1 token being left in the central place, two more executions of F_1 are necessary to allow one execution of F_2. Finally, since after this execution, 2 tokens are left, the 3rd and last execution of F_2 within the LCM=15 ms considered period has to wait for the 5th and last execution of F_1, leaving the system in the same state as before the LCM of the periods (i.e., empty central place). If we wanted a different pattern, we could adjust the initial marking of the central place. As an example, if the initial marking is 2 tokens, then $F_{1,1}$ precedes $F_{2,1}$ (for convenience we here use the second index to represent, within an LCM of the period, the order in which the function is executed), then $F_{1,3}$ precedes $F_{2,2}$ and $F_{1,5}$ precedes $F_{2,3}$. The initial marking can also be negative, representing for example a necessary initialization phase during which F_2 has to wait for the data produced by F_1. SPC patterns can also include cycles, as long as they do not create cyclic dependencies between jobs.

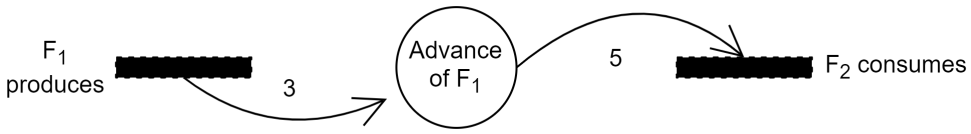


Figure 25: Precedence pattern of SPC represented by a Petri net

The main advantages of the SPC compared to other generalized precedence pattern is therefore the simplistic notation (a SPC can be represented only by one integer, which is the value of the initial marking of the central place), and the fact that some analysis tools have been provided in the real-time scheduling domain.

Table 13: Multiperiodic precedence constraints

Concept	Thread-level static scheduler
Recommendation	Advisory
AADL related concept(s)	N/A
UML MARTE related concept(s)	N/A

5.2.4 Classic ADL concepts

This section regroups concepts which are present in most ADLs, they are grouped in this section for the sake of completeness, but no specific details are given.

5.2.4.1 Processes

A process is a memory partition, able to hold at least one thread (main) and other threads. The threads within a partition are usually able to share global variables. Some OSs differentiate processes from threads, while others consider that each thread executes in a single process. This will determine what kinds of communication between threads will be possible, and in what cases threads are isolated from others, also having a considerable impact on the analysis of real-time aspects. Process isolation is mandatory for ensuring the central concept of freedom from interference, defined in ISO 26262 for example. Since any function executing in any thread executing within a single process can read and write anywhere in the process memory, any function can interfere out of control with any other function of the same process. As a result, freedom from interference between two functions requires at least that these two functions be part of two different processes. Most OSs offerimplement processes usually by using virtual memory addressing, which is requires a specific hardware help provided by a Memory Management Unit (MMU). Some OSs (e.g. VxWorks) offer static memory isolation, with or without hardware control at the hardware level.

Table 14: Processes

Concept	Processes
Recommendation	Required
AADL related concept(s)	Process and every related concept
UML MARTE related concept(s)	MemoryPartition and every related concept

The fact that most process decomposition is based on virtual memory implies that a variable in a process memory space can simply not be addressed by another process. As a result, inter-process communication requires specific use of functions provided by the OS, such as IPC or sockets for example (see Section 3.3.2.1).

5.2.4.2 Partitions

A partition is a static memory allocation as well as time slot allocation, able to host an operating system. It offers the hosted system a communication API as well as an API to access to platform peripherals, such as the network interface. The most common partitioning systems used in avionics conform to the ARINC 653 standard. Some autopilots rely on ARINC 653 systems, making them able to strongly insure freedom from interference.

Table 15: Partitions

Concept	Partitions
Recommendation	Advisory
AADL related concept(s)	ARINC 653 property set
UML MARTE related concept(s)	ARINC 653 library

5.2.4.3 OS

The Operating System (OS) is responsible for providing application interfaces to the hardware pieces of the physical device, such as the processor itself, memory slots, I/O ports, etc.

Each OS has its own architecture, defining if whether multiple partitions, processes and threads are allowed or not. In the negative case, we can consider that the OS has a single partition, a single process and a single thread. In system modelling, OSs are often left abstract and represented by their scheduling policy (among partitions, processes or threads), and services, drivers and protocol stacks.

5.2.4.4 Platform

The platform is the hardware in which the whole mentioned software executes. It will determine how fast calculations can execute, how much memory is available, which I/O ports can be accessed, the energy consumption, built-in sensors, and OS compatibility – not every OS can run in every platform. A platform can also include specific devices that are worth representing, such as memories, sensors, actuators,

and I/O devices such as network buses. Embedded drone hardware is not different from other embedded platforms, and classic ADLs models should be able to model this platform directly.

Table 16: Hardware platform

Concept	Platform
Recommendation	Required
AADL related concept(s)	Processor, Memory, Bus, Device
UML MARTE related concept(s)	hwProcessor, hwMemory, hwBus, hwDevice

5.2.4.5 Allocation

Like in any AADL, an autopilot implementation shall allow allocations to be represented: functions to threads, threads to processes, processes to partitions or OSs, OSs to processor and memory or to partitions.

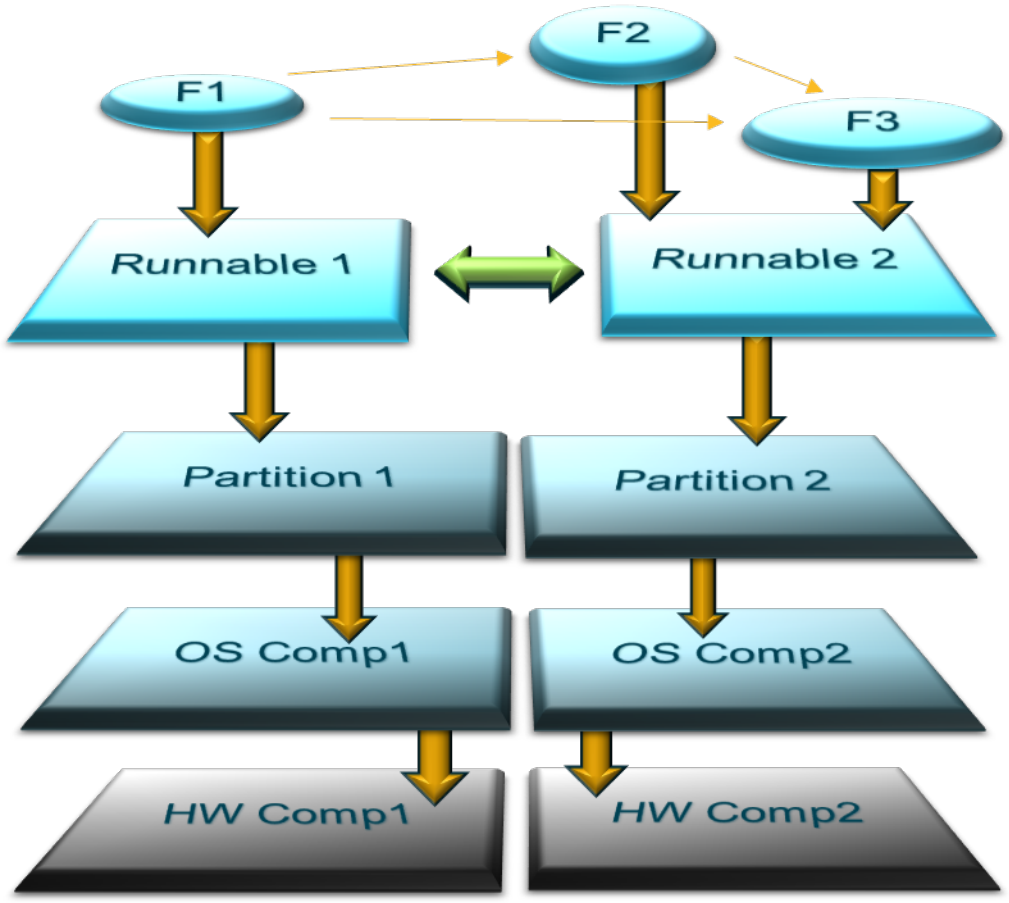


Figure 26: Complete view of a system model

5.2.4.6 Sensors and actuators

In most autopilots, sensors use a short list of I/O interfaces. Depending on the interface, only some specific programming patterns can be used. This section regroups the most common interfaces and the corresponding programming patterns.

Programming patterns

When reading a sensor, programming patterns can be grouped in three categories:

1. Time based (polling): this is the simplest programming pattern: a reading function is called cyclically or periodically. Note that if the reading function is slow, this will slow down the thread in charge of the polling.
2. Event based (interrupt based): this pattern requires an interrupt to be triggered by the I/O device when some reading is available. This is the most reactive model, when the interrupt is used to trigger the reading function, either directly in an interrupt service routine (ISR) or in a thread triggered by an ISR.
3. Transaction: this pattern is often used with master-slave buses (I2C, SPI, etc.) where the master that requests a reading, at the beginning of a transaction, and waits for the addressed device to answer within the same transaction. The main drawback of this method is that the read may be slow (dozens up to hundreds of microseconds for an I2C bus).

General Purpose Inputs Outputs (GPIO)

GPIO are two states pins, able to allow conversion between a binary value and two possible electrical levels. A configurable GPIO pin can be configured as input, converting an external signal into a binary value, or as output, converting a binary value into an electric signal.

Some GPIO pins configured as inputs can trigger an interrupt when the electric signal changes from low to high or from high to low level. They can be called counters, or pullup pins.

For drones, typical uses of such I/O are alimentation relays or LEDs for outputs, and switches for inputs. Reading a GPIO input pin can either be time based or event based in the case where the input can trigger an interrupt.

Analog-Digital Converter (ADC)

Most sensors deliver natively an analogue signal: the electric tension (typically in the range [0-5V] or [0-10V]) represents the measured physical quantity. An ADC is the device in charge of converting an output analogue signal to a numerical value, expressed as a fixed-point or a floating-point value. This is therefore an input, and usually, it is not able to trigger an interrupt. As a result, the only available programming pattern is time based. Examples of analogue sensors are magnetometers, gyroscopes, accelerometers.

Asynchronous serial I/O (UART)

Despite its age, the RS-232 protocol is still very common on drones and in embedded systems in general. It is mainly due to its simplicity: for a unidirectional communication between peers, since on an embedded system the ground is often common, only one wire is necessary. On a platform with an UART controller, the arrival of a byte triggers an interrupt, as a result, possible programming patterns are event-based and time-based. If the communication is time-based, since throughput are often slow, reading a byte can take up from 173 µs at “fast” speed up to 1 ms at “slow” speed. Examples of asynchronous serial I/O sensors are GNSS sensors, and wireless MoDems.

Synchronous master-slave buses (I2C, SPI, etc.)

In these synchronous buses, one or several masters share a bus with one or several slave devices. Only a master can start a transaction: it sends the address of the target device, followed by an acknowledgement bit. On synchronous buses, a bit duration on the bus is such that if sent at a bus extremity, enough time is given for it to reach the furthest extremity and come back before the next bit is sent. The bus also has a recessive state and a dominant state, such that if a recessive bit is sent, it can be overridden by the dominant bit with the same bit duration. It allows a node to send a recessive acknowledgment bit and wait for the addressed node to override it with a dominant bit, acknowledging the reception of its request. Then the interrogated device can send what the master requested. The synchronous nature of buses forbids the throughput to be high (for example, on I2C, bits are sent at some hundreds of hertz). These buses can be programmed in request-response pattern, the problem

being that a transaction issuing the reading of a 32 bits integer from a device takes hundreds of microseconds.

Networks (Ethernet, CAN, WiFi, etc.)

Assuming there is the adequate network controller, these controllers typically trigger an interrupt when a frame arrives. As a result, the typical programming pattern is event based. Note that most OSs provide drivers which are event-based, store the received frames in a buffer, allowing the application to easily be time based: it just polls the buffer.

Pulse Width Modulation (PWM)

PWM communication are common especially with model aircrafts actuators. A PWM signal is a sequence of periodic electric signals where the information is encoded in the ratio high (or low) level duration of the signal divided by the PWM period. Generating a PWM requires a PWM output, linking a programmable clock to a GPIO output, while reading a PWM requires a pull-up GPIO input to call an ISR able to count the delay between rising and falling edges and store it in a variable or a queue. The classic programming pattern for reading PWM inputs is therefore time based: the variable or the queue where the last input(s) value is (or are) stored is polled.

5.2.5 Communications through software buses

Software buses, whether standards, and de facto-standards, or ad-hoc ones, are very common in autopilots (see Section 3.3.2). They provide a software abstraction allowing functions to communicate within a scope. The scope is of prime importance to understand how a function input, or output, can be impacted through a software bus. For example, some software buses, such as the Paparazzi ABI, are strictly limited to single thread communication. These buses provide therefore an abstraction allowing two functions hosted by the same thread to communicate. Other software buses, such as ROS and ROS2, allow functions communication, even if they are hosted in different processes, which can themselves be hosted on different CPUs. Of course, it is always possible to represent the full route taken by the software bus. For a multi-platform software bus, a possible route for the transported data could be function to thread, thread to process, process to OS, OS to platform bus, bus to target platform, target platform to target process, target process to target multi-partition OS, multi-partition OS to hosted OS, hosted OS to target process, target process to target thread, and target thread to target function. This would be a very heavy modelling effort just to represent, for example, that a ROS topic is updated by a function, and that another function is reading it. Moreover, it would make the model way less readable than if for example, we were just representing in the model the fact that a function updates a data with the topic name, and that another function reads the same data asynchronously, on another platform. Functionally speaking, it is more readable and easier to model, even if there is some apparent semantics loss on the exact path that is used by the data, impacting, for example, performance analysis. This loss is in fact, only apparent, since given a software bus, knowing the function updating the topic and the function reading the topic, it would be possible from the model to rebuild the full route taken by the data.

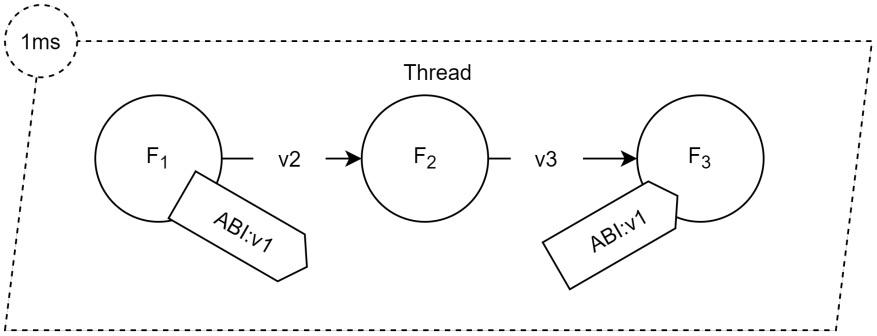


Figure 27: Functions communicating using a software bus within the same thread

Figure 27 represents in an AADL-inspired syntax how an abstract software bus could be represented to model a communication within the same thread, like ABI is able to handle. It does not save much modelling effort, nevertheless, the example given Figure 28 represents a communication through a software bus crossing different platforms (hardware platforms are not represented on the figure). On this figure, Thread 1 could be a part of the core autopilot, while Thread 2 could be a ROS node.

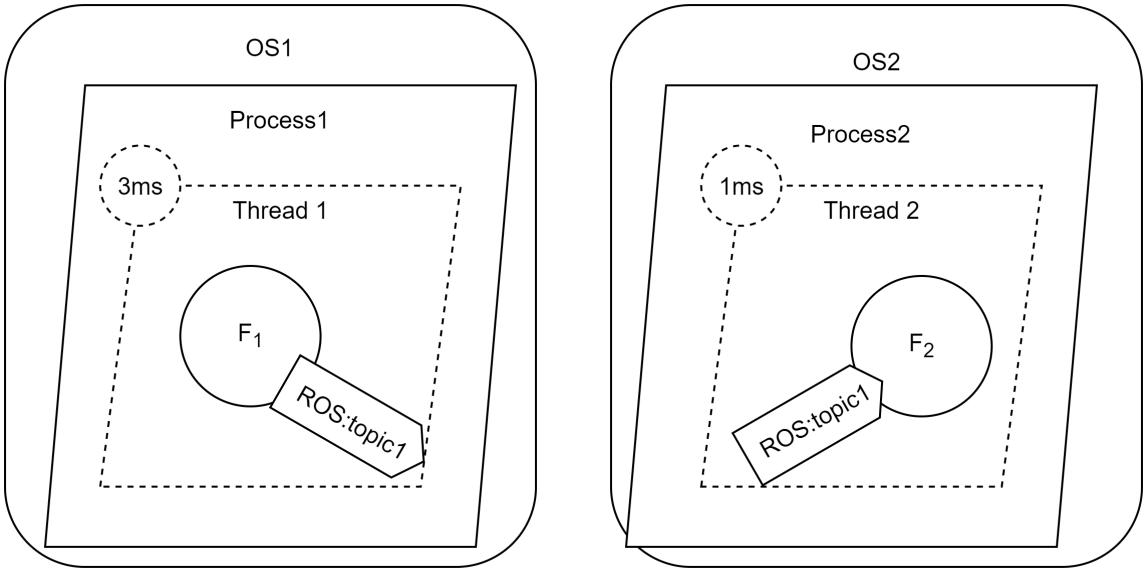


Figure 28: Functions communicating using a software bus across different platforms

Table 17: Software buses

Concept	Software bus
Recommendation	Advisory
AADL related concept(s)	N/A
UML MARTE related concept(s)	N/A

5.2.6 Summary

In order to summarize in a single location the specificities, which could either improve (Advisory concepts), or be required (Required concept) to model autopilot implementations within an ADL are regrouped in Table 18.

Table 18: Summary of Drone embedded architecture related concepts

Concept	Ontological annotations
Recommendation	Advisory – allows better readability
AADL related concept(s)	Subprogram
UML MARTE related concept(s)	Step and/or UML operation
Concept	**Function**
Recommendation	Required
AADL related concept(s)	Subprogram
UML MARTE related concept(s)	Step and/or UML operation
Concept	**Function inputs and outputs**
Recommendation	Required
AADL related concept(s)	Subprogram parameters
UML MARTE related concept(s)	UML parameters
Concept	**Function inputs and outputs**
Recommendation	Required

AADL related concept(s)	Parameter Connections
UML MARTE related concept(s)	Object flow on UML activity diagram
C4D specific notes	Often implemented through unprotected global variables in a monolithic implementation
Concept	**Hierarchical decomposition of functions**
Recommendation	Advisory
AADL related concept(s)	N/A
UML MARTE related concept(s)	N/A or using the concepts of components and sub-components
Concept	**Function logics**
Recommendation	Advisory
AADL related concept(s)	State machines and Modes
UML MARTE related concept(s)	UML State machines, UML activity diagram, other behavioural UML diagrams
Concept	**Threads**
Recommendation	Required
AADL related concept(s)	Thread
UML MARTE related concept(s)	swSchedulableResource, SaScenario, WorkloadEvent, etc.
Concept	**Scheduling**
Recommendation	Required
AADL related concept(s)	DispatchProtocol, etc.
UML MARTE related concept(s)	computingResource, scheduler
Concept	**Threads**
Recommendation	Required
AADL related concept(s)	Data port, event port, event data port and port connections
UML MARTE related concept(s)	SwCommunicationResource, SwMutualExclusionResource, FlowPort, MsgPort
Recommendation	Advisory
AADL related concept(s)	N/A
UML MARTE related concept(s)	swSchedulableResource with its properties isStaticSchedulingFeature set to true, isPreemptable set to fasle, and timeSliceElements representing the static schedule. Nevertheless, the conditional functions (such as F_5 in Figure 23) cannot be represented.
Concept	**Thread-level static scheduler**
Recommendation	Advisory
AADL related concept(s)	N/A
UML MARTE related concept(s)	N/A
Concept	**Processes**
Recommendation	Required
AADL related concept(s)	Process and every related concept
UML MARTE related concept(s)	MemoryPartition and every related concept
Concept	**Partitions**
Recommendation	Advisory
AADL related concept(s)	ARINC 653 property set
UML MARTE related concept(s)	ARINC 653 library
Concept	**Platform**
Recommendation	Required
AADL related concept(s)	Processor, Memory, Bus, Device

UML MARTE related concept(s)	hwProcessor, hwMemory, hwBus, hwDevice
Concept	**Software bus**
Recommendation	Advisory
AADL related concept(s)	N/A
UML MARTE related concept(s)	N/A

6 Implementation of Enabling Technologies

This section presents the advances in the implementation of Enabling Technologies. The C4D repository can be found at the address:

https://comp4drones.github.io/Component_repository/wp3_components_list/

6.1 WP3-01 Drone Pre-certified MPSoC based module

Levels	Functional
Require	Communication service to obtain data from the neighbours
Provide	Hardware blueprint for implementing different architectural blocks, such as, obstacle detection or obstacle avoidance.
Input	Sensor data or obstacle distance data
Output	Required obstacle distance or the new trajectory for avoiding the present obstacle.
C4D building block	Obstacle Avoidance, Obstacle Detection.
TRL	4

6.1.1 Detailed Description

Nowadays, commercial drones rely heavily in the use of microcontrollers to execute the autopilot that controls it. This is the case of the most used autopilots, such as, PX4, Paparazzi or Ardupilot, which has been analysed in the scope of this project.

In recent years, these drones had added several features that makes them more autonomous, not needing an external input to offer more secure and reliable flights; and are capable of doing more and more tasks, such as in-flight object detection, sensor data gathering and processing, or SLAM algorithm execution.

This has been enabled by additional computing capabilities delivered by companion computers packed in within drones.

Year by year, this extra computing power is increasing in capability, power performance or factory form, allowing to implement more complex behaviours in drones, that were not possible not that long ago.

Following this trend, the use of FPGA in edge devices hasn't been that common in contrast to GPUs or ARM-based processors. One of the reasons behind is because of its programming complexity, they require a deep knowledge of how they work and expertise to program them. But state-of-the-art AI techniques and data processing algorithms are nowadays commonly implemented in such systems, extending the usage of this devices.

In the project scope, the Drone pre-certified MPSoC based-module brings a modular hardware blueprint to enable and ease the use of modern heterogeneous-computing architectures to the drone architecture.

The device contains a Zynq-UltraScale+ SoC that provides of a flexible computing architecture, which contains a quad-core ARM processor capable of running Linux, dual-core Cortex-R5F optimized for real-time and safety-critical applications and a FPGA for parallel algorithm execution and data processing- to implement demanding computing and communication applications. This SoC provides the means of creating tailored computational architecture for the target applications, considering safety or real-time aspects.

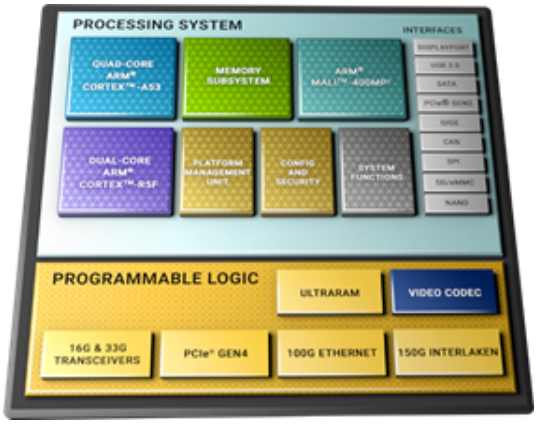


Figure 29: Xilinx Zynq UltraScale+ architecture block diagram

In addition, the device has been designed following modular approach, mimicking the reference architecture, to enable the reuse of the hardware in different use-cases and building blocks.

Design and Implementation

To bring hardware modularity for such a complex piece of hardware, the system has been designed following the SMARC (Smart Mobility Architecture) standard. This standard was created by SGeT (Standardization Group for embedded Technologies), a non-profit organisation.

This standard defines a set of requirements for Computer Modules, which are leveraged in size, mechanical and electrical characteristics, connection pin out and properties or capabilities. Thanks to this, self-contained and defined embedded Computer Modules has been created that are part of a new standardized ecosystem, allowing the use of several vendor Computer Modules in the same manner.

Figure 30: Congatec NXP i.MX8 SMARC 2 module

As the Computer Modules provides encapsules all the computing capabilities, but lack of interfaces to interact with other devices, the use of Carrier Boards is necessary. These Carrier Boards provides of the feeding voltages, communication PHYs or/and connector to interoperate with other devices.

The Drone Pre-certified MPSoC based module is being designed following the SMARC standard, fitting a Zynq UltraScale+ SoC and adding additional electronic devices, such as RAM and eMMC memory chips, to be capable of running complex software and Oss, such as Linux.

The Carrier Board is being designed having the drone hardware requirements in mind. The design provide means of connecting sensors to the hardware. For that purpose, Ethernet, USB, or CSI ports has been added. These communication means are commonly used to connect sensors like Lidars or cameras that enables the execution of object detection and classification, obstacle avoidance or other tasks, such as data gathering. In addition, it provides of I2C and SPI bus connection, which are widely

used to attach low throughput sensors or microSD slot for data recording. It also provides 12v feeding port, to feed attached sensors.

In addition, a safety assessment has been done to ease the compliance of regulatory requirements. As outcome to this assessment, additional hardware elements have been added to mitigate the identified risk sources.

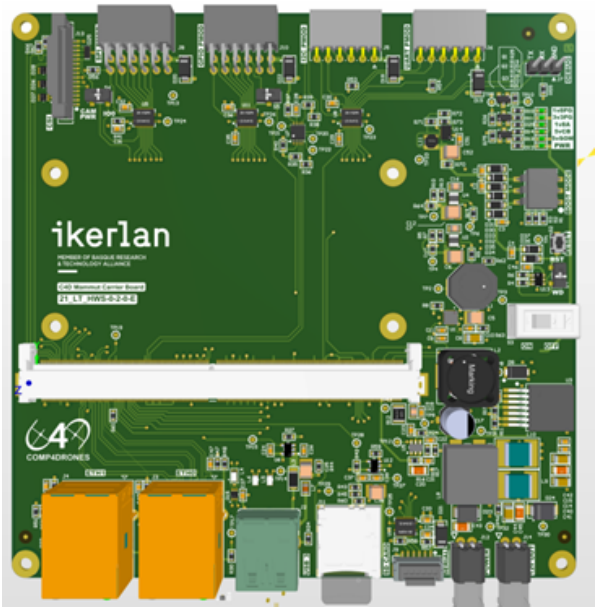


Figure 31: A preview of the Ikerlan's Mammut Carrier Board

All this hardware provides the basis for implementing desired building blocks, thanks to the powerful computing capabilities and flexible hardware setup.

The next section provides insight of how this developed design eases the integration of architectural building blocks.

Contributions

To understand better the impact of the developing hardware, Figure 32 abstracts how a C4D building block would be implemented in the Drone Pre-certified MPSoC based module, in this case, an Obstacle Detection block.

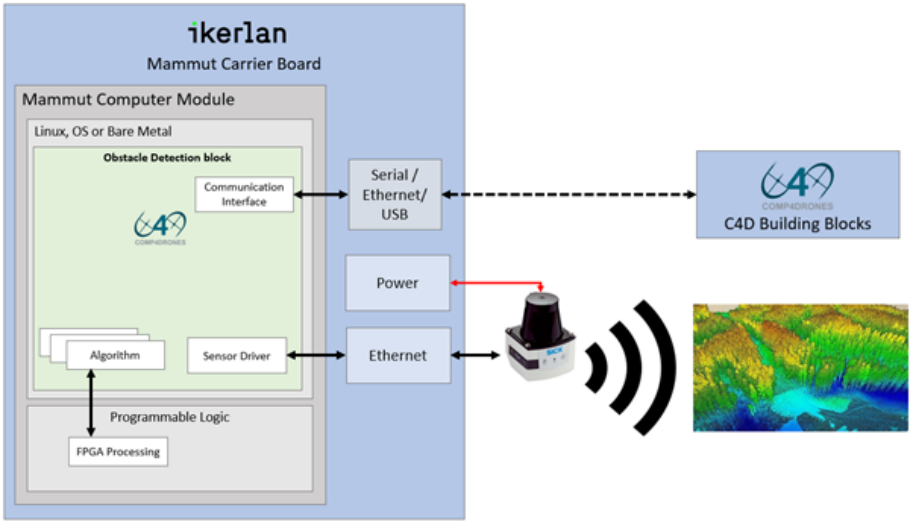


Figure 32: Building block implementation conceptual diagram

As we can see, the developed hardware can contain all the hardware and software elements that would supports the building block. Thanks to its physical interfaces, an obstacle sensing device can be plugged, for example, a LiDAR. The Mammut Computer Module would be able to fetch the data thanks to the drivers and execute the specific algorithms. These algorithms could benefit of the programmable logic unit, which offers parallel computing and high frequency throughput. Once that is ready, it would send that data, in the specified data bus by the C4D architecture to the rest of C4D building blocks.

This setup can be replaced, adapting the required functionality, and deploying new C4D building block.

6.2 WP3-02 Modular SoC-based embedded reference architecture

Levels	System
Require	Specification of the architecture, D3.2
Provide:	Heterogeneous (FPGA + MPU) platform for carrying out computationally demanding algorithms on a drone
Input	HW accelerators and SW implementation of the core drone building blocks, i.e. for perception, actuation, flight planning, guidance and control, health and payload management, communication, data management and mission management.
Output	Energy efficient medium (SoM with a carrier board) and supporting software infrastructure for deploying multitude of autonomous flight control architectures.
C4D building block:	N/A, i.e. reference architecture
TRL:	5

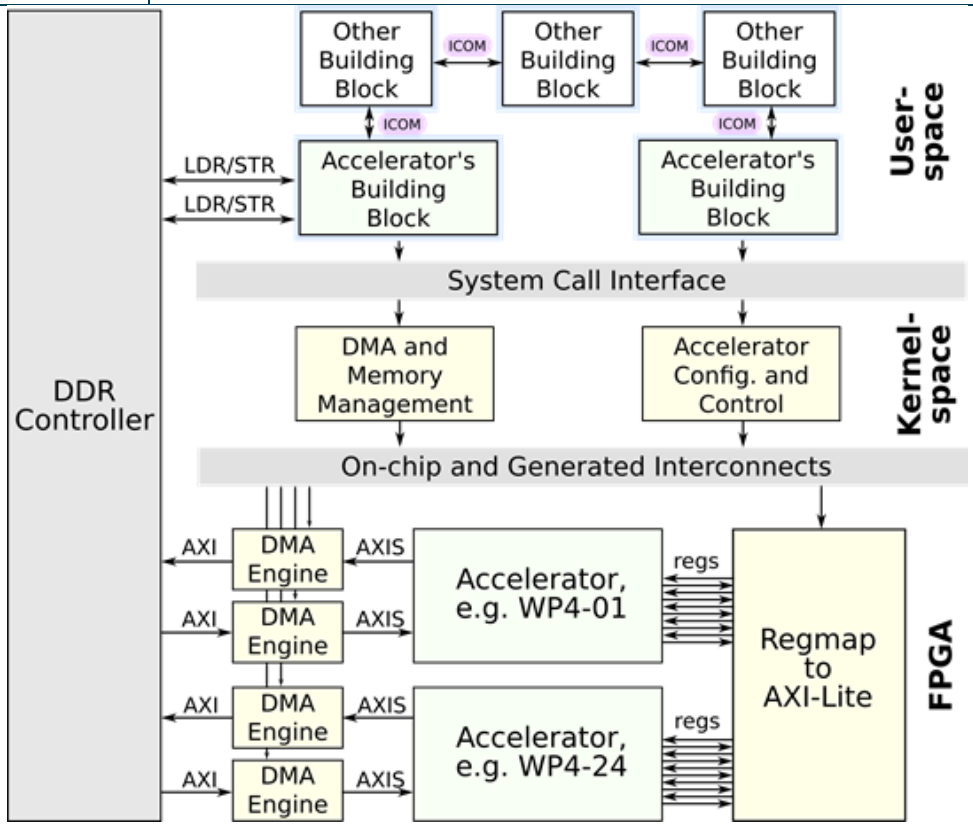


Figure 33: Internal system architecture of the Modular SoC-based embedded reference architecture

6.2.1 Detailed Description

Core enablers for reducing energy consumption and enabling mobile computing are *System-on-Chip* (SoC) and their extension *Heterogeneous System-on-Chip* (HSoC) technologies. The currently available embedded flight controller platforms mostly use the sequential processing paradigm, which is a limiting factor for the drone's onboard computational capacity. Our objective is to improve the SoA drone capabilities by using modern HSoCs, which reduce energy consumption and accelerate computationally intensive algorithms, e.g. localization based on visual SLAM. We focus on employing HSoCs that combine *Field Programmable Gate Arrays* (FPGA) and processor computing paradigms within a single chip. Hence MPU can be used for providing a high-level control and for processing non-localized (requiring intense scatter-gather data access) algorithms, while FPGA is dedicated to processing pipelines and tasks requiring parallel and/or determined execution.

In the C4D project, EDI designs and contributes a Modular SoC-based embedded reference architecture that includes the following hardware and software components:

- **MPSoC-based autonomous flight controller hardware**, including a Zynq UltraScale+ ZU15EG *System on Module* (SoM) from Trenz Electronics with a custom carrier board tailored for use with midsize drones.
- **FPGA vendor-agnostic IP cores** for standardized communication with the accelerators and their run-time configuration.
- **Linux modules** for *Direct Memory Access* (DMA) engine control, virtual memory management, accessing accelerator configuration and providing user-space API.
- **System architecture frameworks** for implementing component-based software architectures and ensuring appropriate inter-communication mechanism, i.e. inside process, between systems, zero data copy.

6.2.2 Contribution and Improvements

EDI is designing a specialized carrier board PCB for the Ultrascale+ SoMs from Trenz, shown in Figure 34. The design involves exposing standard connectivity such as SPI, I2C, UART, QSPI for MMC as well as high-speed buses – USB2.0 and USB3.0. The reference platform is intended for but not limited to mid-sized drones. Further iterations should bring the size of the computing platform even drones of smaller indoor-type.

Figure 34: Core component of the reference platform – Trenz Ultrascale+ SoM

During the project, EDI plans to utilize the platform to deploy and validate the developed localization component - Hardware-accelerated Optical flow and SLAM (WP4-01) – and, hopefully, accelerators developed by other partners. The FPGA portion of the design utilizes standardized interfaces and vendor-agnostic DMA engines, which enable the unification of the driver modules (for high-performance HSoCs supplied by Xilinx or Intel).

Another improvement is the system architecture implementation tools – *compage*[8] (for component-based software management) and *icom*[9] (for inter-component communication), both of which were originally developed under the PRYSTINE project (G.A. 783190) for high-performance controllers in autonomous cars, and during the C4D project, functionalities are extended for supporting drones.

6.2.3 Design and Implementation

As the PCB design has not yet been finalized (at the writing of this document, the components are selected, the circuit is done but the efforts are directed towards the actual placement & routing and spatial requirement considerations), the other software and hardware (FPGA) related work is being done using prototyping platforms, i.e. Xilinx ZCU102 Ultrascale+ MPSoC. The on-chip system architecture, shown in Figure 33, represents the hardware and software portions of the MPSoC package, i.e. FPGA, kernel-space and user-space.

At the hardware level, the selected communication interface for data exchange is based on the *Direct Memory Access* (DMA) engines, as it offloads MPU and is best suited for high data throughput [15]. The DMA engines utilize the standard AXI4 interface, which ensures compatibility with a collection of high-bandwidth cores. Although Figure 33 follows a shared memory model [16], the FPGA may also utilize dedicated FPGA DDR memory, as long as quick access to the data is not required by the software. DMA engine serves as a translator from AXI4 memory mapped to AXIS streaming interface, which is suited for pipelined data-driven circuits, i.e. accelerators. The configuration of the accelerators is achieved by exposing I/Os of the internal registers. A block “Regmap to AXI-Lite” translates access into a memory-mapped protocol, which further is used to interface with the MPU portion of the system. The handoff information to the software is passed in a standardized way (especially for ARM-based SoCs) – by utilizing *Device Tree Blob* (DTB) format.

The autonomous flight controller runs a Linux operating system which assumes the incorporation of hardware abstraction mechanisms, most notably memory virtualization. These complexities are handled by Linux kernel modules that provide a user-space interface for DMA control and accelerator configuration. Notably, the memory virtualization mechanism results in a scattered physical layout of the memory. One option is to use scatter-gather memory access capabilities of the DMA engine. Nevertheless, this involves the creation of transaction descriptor tables (often a linked list) and additional memory access requests from another bus master apart from the MPU. We utilize kernel’s *Contiguous Memory Allocator* [17] (CMA) feature, which marks the physical memory region and manages its run-time swaps whenever a new allocation takes place. The user space is supplied with a *malloc*-like interface, which handles memory management and contiguous page mapping.

The user-space application follows a blackboard programming pattern [18], which is well-suited for research work. We provide two low-footprint open-source frameworks for the implementation of system architectures, the aforementioned *compage* and *icom*. *Compage* stores component configuration information in a separate *Executable Linked Format* (ELF) file segment, which provides a modular addition of new components (threads or processes). It also provides an INI-based file for configuring and replicating the software components. *Icom,* on the other hand, provides inter-component communication. It supports multiple communication paradigms (PUB-SUB, REQ-REP, PUSH-PULL) and can use the most appropriate communication mechanism (sockets, FIFO, message queues, etc.) depending on the communicating component location (distinct machines, same machine or same memory space).

[8] https://gitlab.com/rihards.novickis/compage
[9] https://gitlab.com/rihards.novickis/icom

6.3 WP3-03 Sensor information algorithms

Levels	Functional
Require	Payload data (drone images)
Provide:	Tone mapped HDR video as Axi4 Video Stream
Input	RGB images captured by drone (Payload data)
Output	RGB images processed by component (HDR tone mapped images)
C4D building block:	(Video) Data Analytics
TRL:	4

6.3.1 Detailed Description

HDR is part of image processing that focuses on capturing, processing, and displaying images with a High Dynamic Range. Its primary goal is to achieve reproduction of the captured scene on digital devices. HDR (HDR) for single images is deeply studied topic (although it is still quite opened). HDR can be obtained using specialized HDR sensors, or using standard sensors by acquisition of image sequence with different exposure times. These images can be captured simultaneously e.g. through beam splitter on several CCD/CMOS sensors or more often sequentially. HDR can be obtained using specialized HDR sensors, or using standard sensors by acquisition of image sequence with different exposure times. These images can be captured simultaneously e.g. through beam splitter on several CCD/CMOS or more often sequentially (merging/fusion of exposures). With the advent of HDR video capturing new problems arose. Application of TMO without careful consideration of temporal coherence between consecutive frames may lead to adverse effects. Video tone mapping methods needs to carefully consider temporal coherence to preserve this temporal character, for example during fast luminance changes.

In the **C4D** project BUT is improving HDR video tone mapping FPGA IP core. The design is based on programmable hardware tightly connected to an "embedded" processor (FPGA SoC Xilinx Zynq, but may also be implemented on other platforms with FPGA). It covers all functionality: reading data from a camera sensor, merging multiple images with alternating exposures into HDR images/HDR video and applying HDR tone mapping. The system can be extended with other functions (software, hardware or FPGA IP core) such as HDR video compression, image pre-processing, exposure control, and the “ghost-free” function removes possible artifacts caused by the movement of objects in the programmable hardware. This block provides acquired data in HDR or tone mapped format and can be extended with other data analytics tools/algorithms (e.g., detectors).

In the **C4D** reference architecture context, this block supports data acquisition for further processing either in the FPGA or in the following systems. Sensor information algorithms is part of payload management – data acquisition block, and component provides inputs to the data management block (i.e., payload data analytics).

6.3.2 Contribution and Improvements

BUT is implementing and improving sensor data processing algorithms which include software and firmware for FPGA. This involves video processing algorithms (for example HDR algorithms). HDR multi-exposure fusion algorithm to be implemented in the drone, possibly also implementing tone mapping and/or ghost removal in order to "feed" further image and video processing subsystems in the drone by image information with high dynamic range.

BUT increased performance of the algorithms, which reduce latency and increase throughput (currently IP core can process up to 200 mega pixels per second). Robustness of the controller with respect to environmental disturbances and increased resiliency. This improvement will be based on increased robustness of the video processing with respect to HDR while keeping the processing means and extent of video processing "unchanged" thanks to the tone mapping that virtually brings the "same image format" as in usual processing.

6.3.3 Design and Implementation

Component is divided into four main blocks (each block can be use independently):

1. Sensor data acquisition
2. Buffering
3. HDR Merging and deghosting
4. HDR Tone Mapping

Sensor data acquisition

Architecture is based on Xilinx Zynq platform which is connected to Python 2000 CMOS sensor using LVDS (Low-voltage differential signaling) interface. The CMOS output consists of a raw CFA (Color Filter Array) image data with a Bayer filter mosaic.

Buffering

The raw image is stored to DDR memory using DMA and double buffering to avoid overwriting of the data. For DDR write one DMA is used. For reading image data 3 DMAs are used.

HDR Merging and deghosting

The HDR merge block reads three image streams simultaneously through the DMAs. First, it applies inverse camera response function to obtain image with linear response and merge HDR image. The merging algorithm performs per-pixel processing and requires a relatively small number of per-pixel operations. Some of its functionality is computationally demanding (e.g. division and Gauss function calculation), however, it can be optimised and/or tabulated. The Gaussian function used for ghosting suppression can be convenient because the pixel values are discrete and only a finite combination of pixel values is possible.

HDR Tone Mapping

HDR pipeline is implemented in FPGA and pipelined at 200MHz while processing one pixel per clock. Input of Tone mapping block is 18bit CFA pixel in 10.8 fixed-point (FP) representation (10 integer and 4 fractional bits) and output is RGB pixel in <0,1> interval. Algorithm is based on Durand and Dorsey tone mapping operator. Durand operator is originally two pass algorithms, because it requires extreme values of the base layer. Implementation of multi-pass image processing algorithm in FPGA is problematic because of limited memory size. Typically, there is not enough space to store whole image directly in FPGA. In our case we need to compute only minimum and maximum value (or percentiles) of the base layer and we selected approach where minimum and maximum value is used from previous frame.

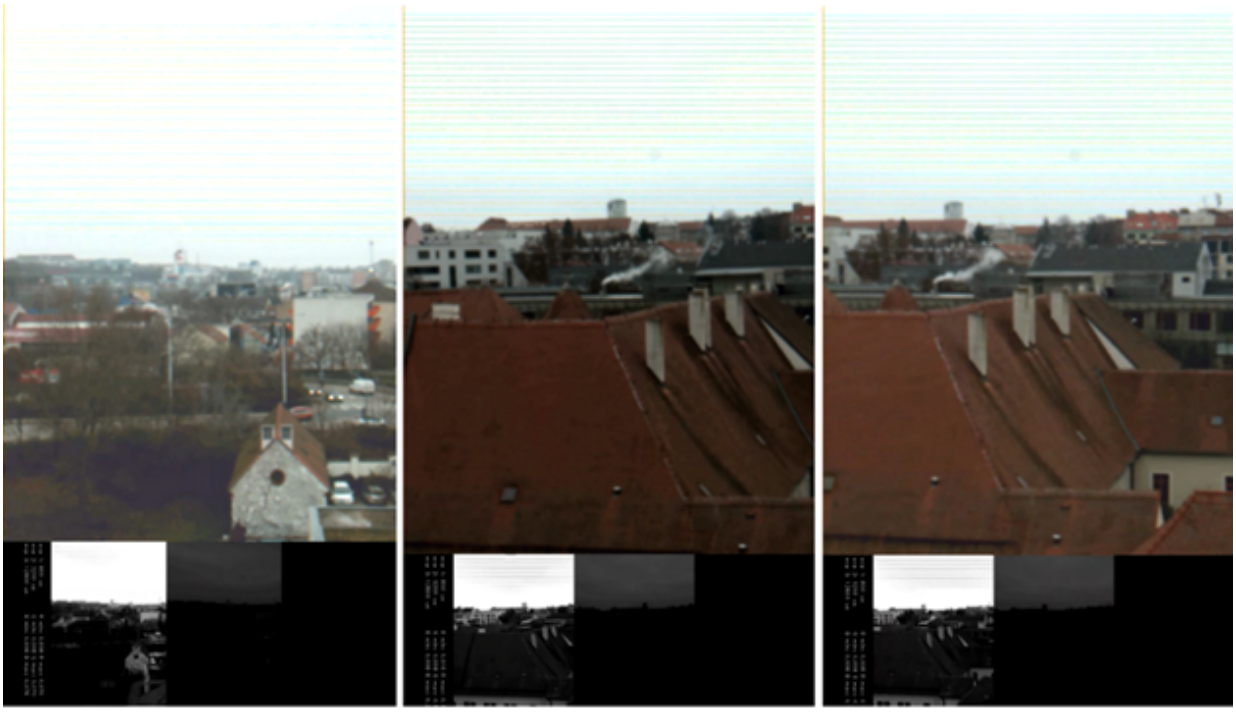


Figure 35: Example of HDR processing during test flight. Top images contain tone mapped images, bottom images show LDR images used for merging and tone mapping

6.4 WP3-04 Computer Vision Components for drones

Levels	Functional
Require	Payload data (drone images)
Provide:	DL-based SW to detect and classify objects (road elements, traffic signs) from images captured by drones
Input	RGB images captured by drone (Payload data)
Output	Inventory of road elements with their position in the terrain
C4D building block:	(Video) Data Analytics
TRL:	5

6.4.1 Detailed Description

Machine learning and deep learning are an arising approach in dealing with large amount of data gained from drones [19]. For infrastructure planning and design, typical data acquired through drones are images. For construction monitoring, either real time videos or 3D models are needed. Focusing on object recognition and detection in aerial images captured by drones, a major challenge with the integration of artificial intelligence and machine learning with autonomous drones' operations is that these tasks are not executable in real-time or near-real-time due to the complexities of these tasks and their computational costs. In the last few years, deep convolutional neural networks have shown to be a reliable approach for image object detection and classification due to their relatively high accuracy and speed. Furthermore, a CNN algorithm enables drones to convert object information from the immediate environment into abstract information that can be interpreted by machines without human interference. Thereby, the main advantage of CNN algorithms is that they can detect and classify objects while being computationally less expensive and superior in performance when compared with other machine-learning methods.

In the C4D project scope, the *computer vision component for drones* brings a deep learning-based software which uses a convolutional neural network (CNN) algorithm to detect, and classify objects from raw data in such a way that it brings to the project capabilities of interpreting surroundings and detecting scenarios from data captured with drones.

6.4.2 Contribution and Improvements

Concretely, the proposed application scenario for the *computer vision component for drones* is the inspection, assessment and maintenance of civil infrastructures and construction elements, in which many inventories for damage and defects detection are carried out. Such inspections and the corresponding inventories are performed through human visual observations, being a tedious and time-consuming work prone to human errors, and therefore, an expensive work, so accelerating such inspections is a current challenge in the construction industry.

Then, the *Computer vision component for drones* is a post-processing computer vision system based on previously trained CNN algorithms which enables the auto-detection and geo-referencing of different objects from RGB images captured by the drones' on-board camera. Particularly, this computer vision system intends to improve the digitalization of the state of the constructive process of a Civil Infrastructure by auto-detecting and geo-referencing different road elements which can be found in any civil infrastructure or construction elements. Thereby, one of the main challenges in the construction industry like the road element inventory realization is solved.

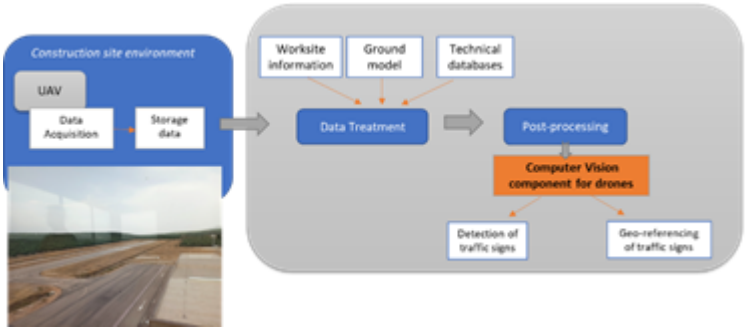


Figure 36: Computer Vision Component in the application scenario

Considering the C4D Reference Architecture Context, the *computer vision component for drones* supports the *(Video) Data Analytics* building block in the *Data Management* block by performing an offline data analysis over the RGB images (that is, payload data) captured by the UAV in order to provide an auto-detection and geo-referencing of different objects. Although this *computer vision component for drones* will provide to the Business domain an inventory of the road elements detected and their corresponding position in the terrain to serve as input for the creation of the BIM model, this *computer vision component for drones* could be extended for other business functions following the generic Building Block defined for the Payload Data Analytics.

6.4.3 Design and Implementation

The *computer vision component for drones* is being developed following the typical steps for a deep-learning-based SW:

1. Dataset creation
2. Election of the most accurate object detection model.
3. Training of the object detection model using previously created dataset.
4. Testing of the object detection model already trained.

So, it is possible to state that the *computer vision component for drones* is built upon two main axes: the **dataset** and the **object detection model**.

For the **dataset creation**, several available datasets have been contemplated:

- **German Traffic Sign Detection Benchmark (GTSDB)** [20]**,** which is a single-image detection dataset based on images taken in Germany. It contains around 900 images, 600 for training and 300 for test. The traffic sign classes in this dataset are shown below:

Figure 37: Traffic sign classes in GTSDB

- Images from **drone flight videos**. This dataset is built with the images captured during the 1st data acquisition campaign performed in UC2 – Construction in Jaén (Spain). An example is shown below:

Figure 38: Example of images from drone flight videos

- Dataset built from the videos generated with **AirSim** (**drone simulation environment)**.

Figure 39: Example of images from drone simulated flight videos

- **Mapillary Traffic Sign Dataset (MTSD)** [21], which it is a dataset with street-level images around the world. These images are annotated with bounding boxes and traffic sign classes. There are 100000 high-resolution images, 52000 of them fully annotated. Also, there are over 300 traffic sign classes, as it is shown below.

Figure 40: Overview of all traffic sign classes in MTSD

Related to the **object detection model**, **Faster R-CNN** [22] is the model used to date, and it is an object detection system proposed by Ren et al. in 2015. Its architecture is shown in Figure 41 and it is formed by two components: Region Proposal Network and Fast Region-Convolutional Neural Network.

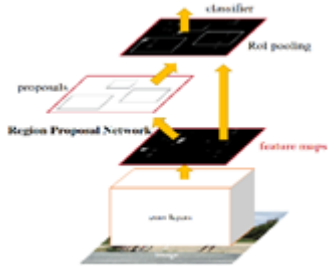


Figure 41: Faster R-CNN model general architecture

Concretely, **Region Proposal Network (RPN)** is a deep convolutional network that proposes image regions where an object might be found. RPN component acts as an attention mechanism that accelerates region proposal phase, and in our use case, it creates new regions (bounding boxes) where a traffic signal exists. Complementarily, the **Fast Region-Convolutional Neural Network (Fast R-CNN)** [23] takes the regions proposed by RPN and classifies them efficiently. In our use case, this component classifies traffic signal regions in 'Mandatory', 'Danger', 'Forbidden'.

Below, it is shown an implementation conceptual diagram for the *computer vision component for drones.*

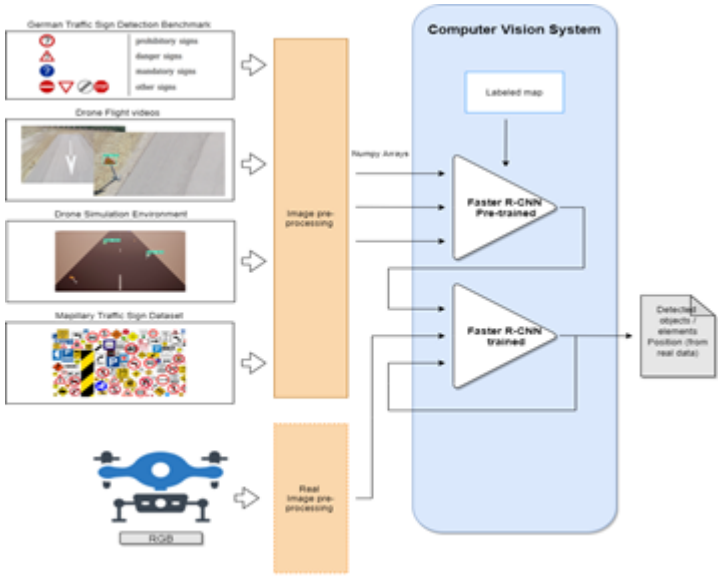


Figure 42: Implementation conceptual diagram for computer vision component

Based on Figure 42, the data flow of *computer vision component for drones* is detailed as follows:

1. Setup of all prerequisites: software libraries are installed and loaded, and dataset and pretrained model are downloaded.
2. The pretrained Faster R-CNN model is loaded into memory using TensorFlow Object Detection API [24]
3. The label map is loaded into memory. This object maps the three category indices to category names for the translation of the model's predictions.
4. The pretrained model performs detection over test set images.
    1. Each image is prepared with Pillow library.
    2. Images are transformed into Numpy arrays.
    3. Images are preprocessed for the model to make predictions on them.
    4. Detection takes place.
5. Predictions are visualized with TensorFlow Object Detection API and Matplotlib's Pyplot module.

Below, it is possible to see traffic sign detections on images from real Drone Flight Videos by the Faster R-CNN model implemented in the *computer vision component for drones.*

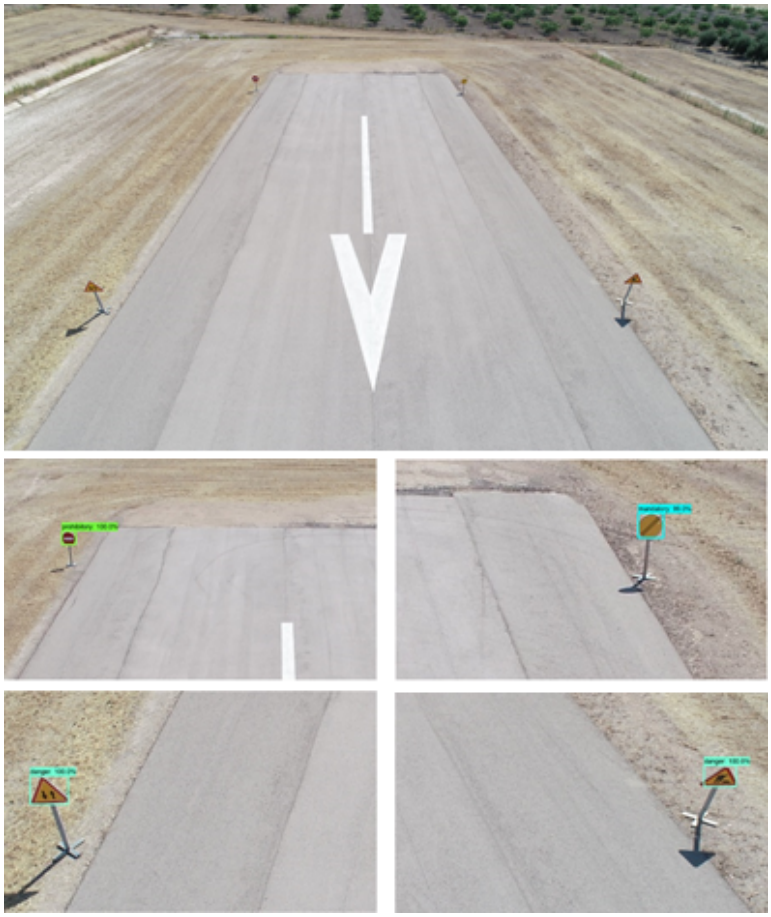


Figure 43: Detections performed by Faster R-CNN on a Drone Flight image. (top) The model fails to perform any detection over the whole image. (bottom) After chopping the image, the model correctly detects and identifies the four traffic signs.

6.5 WP3-10 Generic API and component for trusted communication

Levels	Function
Require	HSM-chip
Provide:	C-based API-lib for accessing HSM hardware-functionality
Input	API-call + generic byte-array (e.g. data to be signed)
Output	Low-level I2C commands, return data to API
C4D building block	Communication
TRL	5

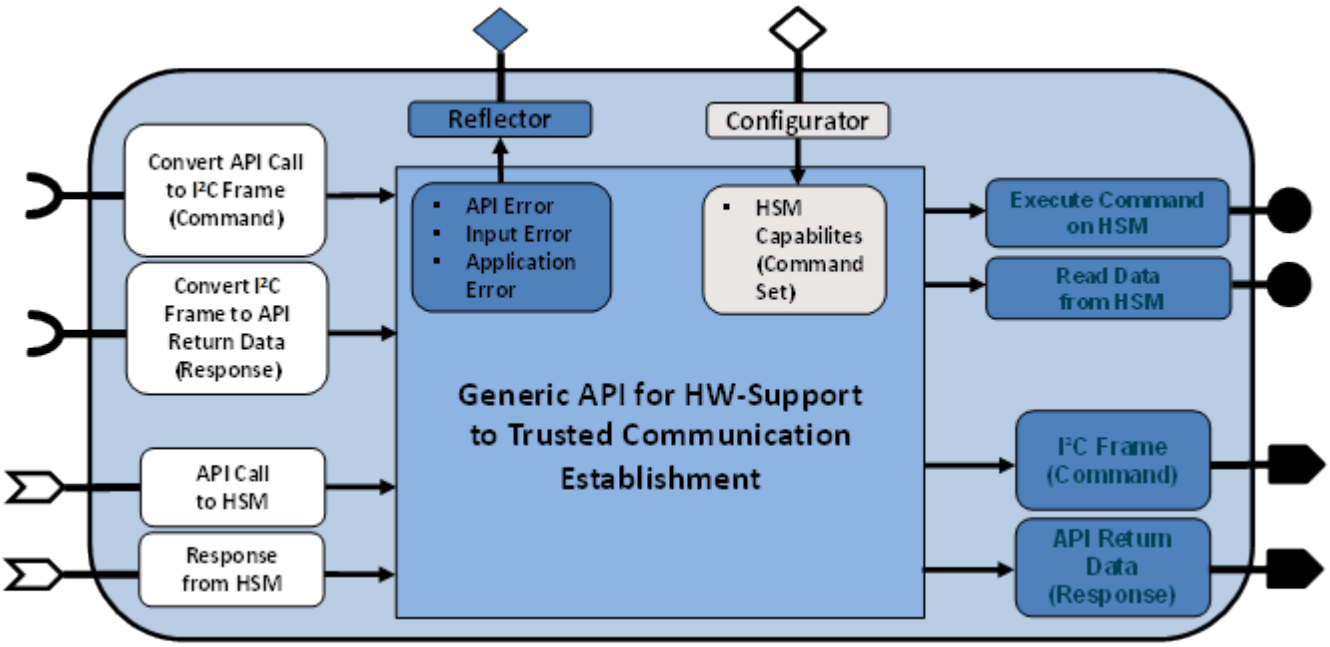


Figure 44: Building block diagram for WP3-10

6.5.1 Scope and contribution

One – small yet important – piece of a jigsaw of an overall drone architecture is protecting the data to be transmitted via various wireless communication channels, and as a consequence thereof the required remote authentication. Protecting the communication link of a drone to any other end point (e.g. other drone, base station, and infrastructure) is important. Otherwise, the communication partner cannot verify if the communication link is established to the partner intended to. Further, in is required to maintain integrity as well as confidentiality. Therefore, in many embedded systems, a protected communication link is established via TLS between the communication partners. However, in today's embedded systems, all security-critical TLS mechanisms are typically purely executed in software in the microcontroller alongside the embedded OS and various applications. This conventional system design also makes the pure software-based TLS prone to software and side-channel attacks. Therefore, to minimize the attack vector, IFAT is developing an enhanced security concept, in which the TLS handshake is supported with a Hardware Security Module (HSM, alternatively also denoted as Secure Element (SE)) in order to perform certain security critical operations and store confidential key material.

While in the course of WP5 IFAT is working on the higher-level implementation of the required security-specific algorithms and libraries, in the course of WP3 IFAT is working different aspects of defining a standardized lower-level API for the easy and modular integration of such a hardware security component into any modular drone architecture. The main purpose of this (low-level) API is to make the required functionalities of an integrated HSM accessible to various drone software-frameworks, which typically are executed on a general purpose microcontroller in C programming language (such as ROS for example). Furthermore, the API is designed in a generic way, in the sense that the same API shall be usable for more than just one hardware security device (since today most new generation/releases of HSM devices have slightly different command sets). In this way, within WP3 IFAT provides a concept to integrate hardware security components into drones with an API to be usable for future drone system integrators for various security relevant tasks. This concept will be described in more details in the paragraphs below.

6.5.2 Design and Implementation

To access the functionality of an HSM, a command library needs to be designed and implemented. An important requirement in this regard is modularity. This is important to prevent code duplication in future

developments and easy replacement of HSMs. Based on that design decision, various different use cases can be addressed due to the high reusability. Therefore, the API for the command library used to establish a trusted communication channel shall be design in a generic way. Additional requirements would be a small memory footprint and multi-threading support. First, many embedded systems have limited resources, and second, the driver should not use any blocking functions, because this would be disastrous in multi-threaded environments. The latter requires synchronization between the API function.

Architecture Design:

As depicted in Figure 45 the multi-threaded architecture is split into two main parts, the transport driver and the command library. The transport driver is communicating with the HSM, whereas the command library exposes the functionality to the higher level user application.

Splitting the architecture in transport driver and command library allows being independent of the hardware, since the provided features of the hardware are different, and the command set can easily be extended. Executing a specific API function results in a serialized command which is added into the command queue by the command library. The transport driver processes the command and relays it via the I2C driver to the HSM, where it is executed and passed back via the transport driver to the command library and subsequently to the user application.

Transport Driver:

The implemented driver (SW-Library) reflects similar layers, which are depicted in yellow in Figure 45. The physical layer mainly consists of a set of registers to communicate with the two surrounding modules, the I²C driver and the data link layer. The data link layer is providing reliability to the communication channel by adding checksums and sequence numbers. Network and transport layer are combined in one module for simplification, since they share the same header structure. The main task of the transport layer is the packet fragmentation. The command queue worker is the direct interface to the command library and handles the state machine to communicate with the HSM. The chip control is out of scope for this deliverable, but it handles the power management of the HSM.

Implemented generic HSM command library:

When the user application calls an API function, the command library serializes with the APDU-builder, into the standardized “APDU”[10] format and submits it to the command queue. The command format is different for various devices, but devices from one product family mostly share the same format. This shows again the importance of splitting the architecture into the lower level transport driver and the command library to offer a generic solution. As depicted in Figure 45, the command library offers a basic set of functions which is used by the user application to establish a trusted communication channel.

This generic library-abstraction a) helps to ease the integration effort for drone system integrators to integrate a HSM component into their drones, and b) provides a more generic API (as compared to state-of-the-art) in the way that the API is not limited to just one specific version of HSM chip-variant. In the case of IFAT, this API is planned to support different Infineon hardware-security chip family derivates and also upcoming versions.

[10] APDU see: https://de.wikipedia.org/wiki/Application_Protocol_Data_Unit

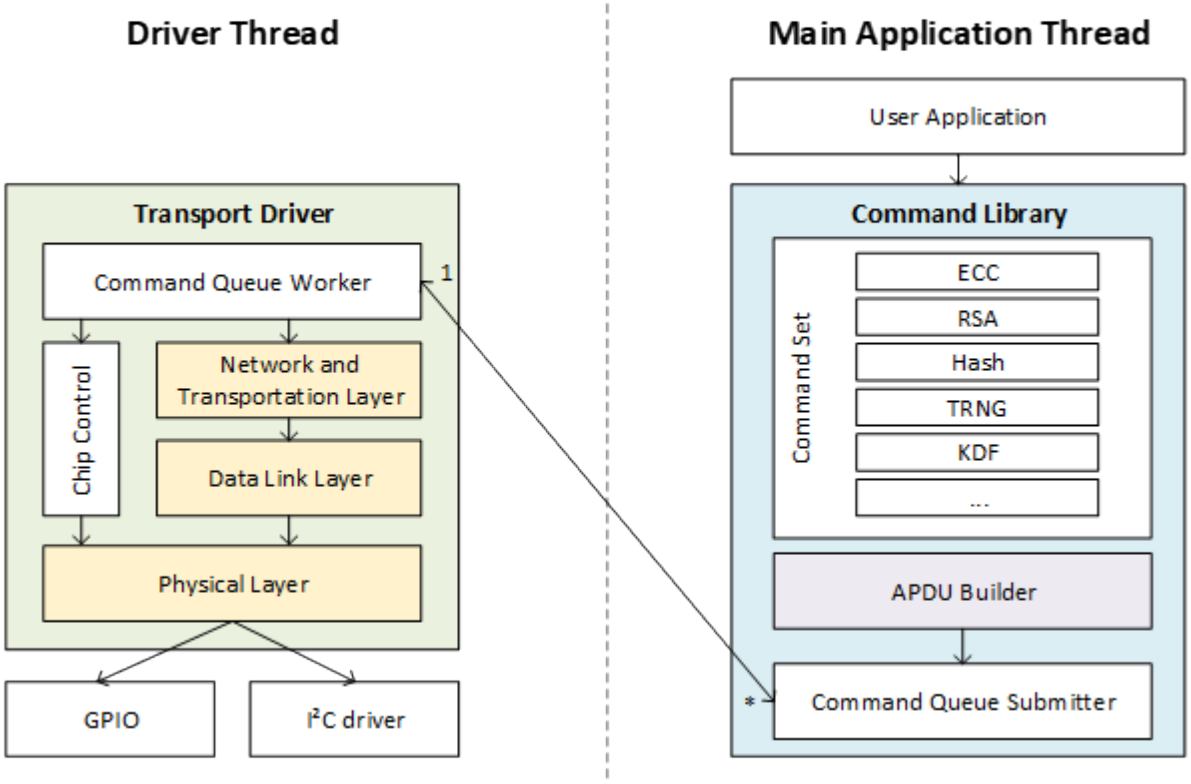


Figure 45: Generic architecture design and implemented SW-API for integrating Infineon HSM modules into a drone

6.6 WP3-13 – Paparazzi UAV

Levels	Platform
Require	C2LINK or Flight Plan definition
Provide	Modular and flexible UAV autopilot system for testing and operation
Input	Depending on the flight mode: • flight plan definition • mission sequence from datalink • position/speed from datalink • low-level remote control
Output	Stabilized aircraft (fixedwing, multicopter, hybrid) on a given trajectory or attitude
C4D building block	Flight control, Actuation, + parts of Perception and Communication
TRL	4 to 6

6.6.1 Current status

Paparazzi is a complete system of open source hardware and software for Unmanned Aircraft Systems (UAS), including both the airborne autopilot as well as complete ground station mission planning and monitoring software utilizing a bi-directional datalink for telemetry and control.

Paparazzi has been created at ENAC in 2003, and is now supported by other institutes such as MAVLAB of the TU-Delft, individual developers, and some private UAV companies from several countries.

The Paparazzi system was initially designed for robust small fixed-wing aircrafts in 2003, but it now supports several other configurations and concepts such as high-aspect ratio gliders, multi-rotors, transitioning vehicles, and rovers.

The communication between the software blocks running on the ground and the airborne autopilot is based on the PPRZLINK library, which provides API in C/C++, Python and OCaml.

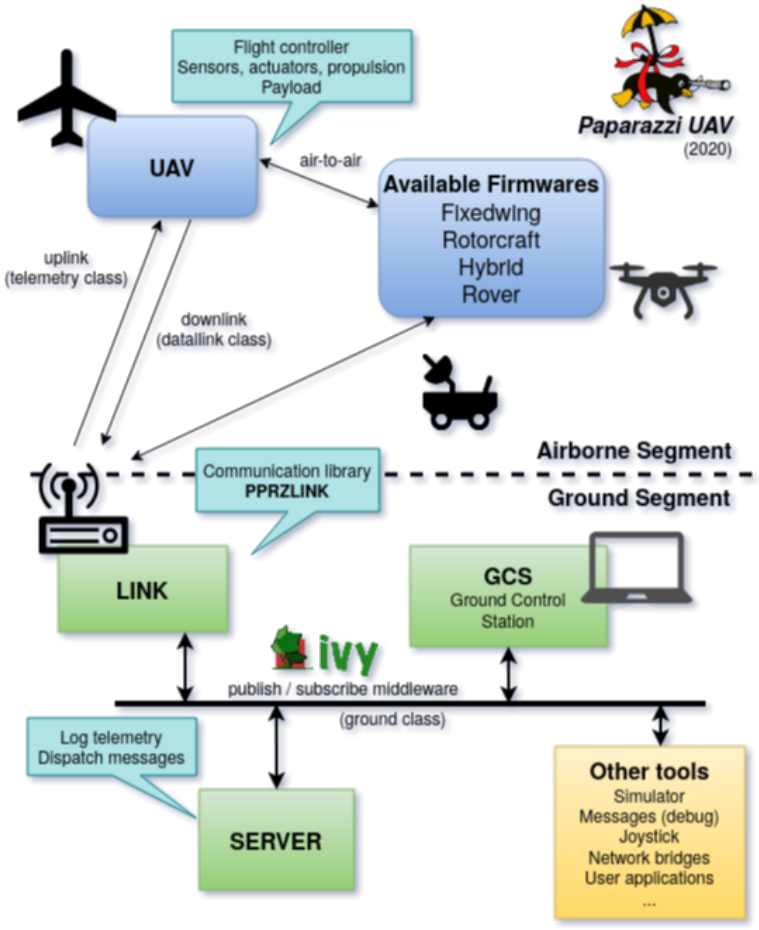


Figure 46: Paparazzi communication architecture

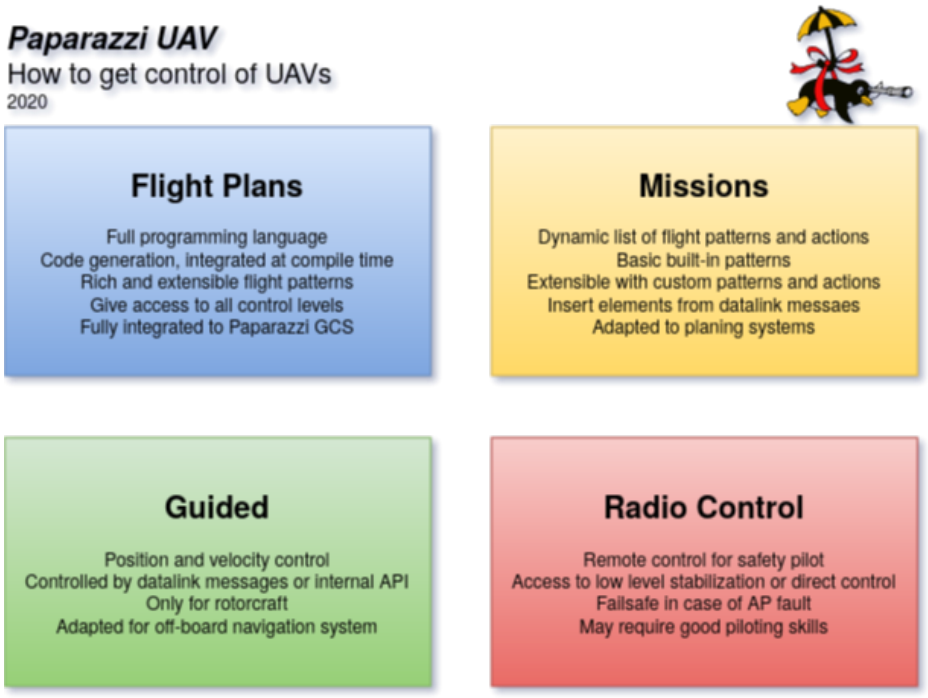


Figure 47: Control options for Paparazzi UAVs

6.6.2 Contribution and Improvements

Several improvements are intended in the scope of C4D projects.

- Improvement of the internal airborne code organization: a detailed analysis of the timing between the elements of the embedded software have led to a new definition of the internal tasks' groups. A particular attention has been given to the timing between the tasks and functions to provide a stable and reliable end-to-end execution time.
- Explicit definition of the dependencies between modules, both at functional and logic level: it allows to simplify the configuration of an aircraft (only relevant high-level modules needs to be specified), the dependency solver will include all the required modules. The overall system is also more robust as it can detect circular dependencies, conflicts and missing functionalities.
- Improvements of the static scheduling for the telemetry messages to spread the link loading over time and avoid buffer saturation and delays. The same approach can be applied to the scheduling of some of the functions calls inside the different tasks of the autopilot, linked to the new code organization mentioned in point 1.

Performance analysis have been carried out. Some results are presented in the following figures. The first two are the different tasks and timing for the legacy architecture for fixedwing and rotorcraft firmwares. It can be seen that the group of tasks are not harmonized and don't reflect the actual functional blocks of the system. If most of the timing are respected, the 'event' polling function that is expected to run at 10kHz is a bit slower than expected.

```
~ [~/dev/paparazzi/sw/tools]↕ perf_profil± ./perf_profil/perf_profil.py ~/work/logs/perf_profil/fixedwing_test3_no_flightrec.LOG
name            (samples)   | period (us)    [std     ] | freq (Hz)   | duty (us)      [std     ] [min     ] [max     ] [nb over ] [%       ]
ap_static       (237)       | 250000.99      [0.994   ] | 4.000       | 43.525         [1.820   ] [40.204  ] [46.690  ] [0        ] [0.0174  ]
sensors         (5956)      | 10001.00       [0.900   ] | 99.990      | 1.481          [0.071   ] [1.282   ] [3.417   ] [0        ] [0.0148  ]
attitude        (3566)      | 16700.94       [458.301 ] | 59.877      | 10.125         [0.510   ] [9.537   ] [14.204  ] [0        ] [0.0606  ]
modules         (5956)      | 10001.00       [12.553  ] | 99.990      | 8.620          [4.819   ] [7.495   ] [59.301  ] [0        ] [0.0862  ]
monitor         (58)        | 1000001.04     [4.042   ] | 1.000       | 2.482          [0.249   ] [2.449   ] [4.366   ] [0        ] [0.0002  ]
telemetry       (5955)      | 10001.00       [14.358  ] | 99.990      | 15.351         [21.589  ] [3.319   ] [141.833 ] [0        ] [0.1535  ]
event           (589*1000)  | 101.08         [0.115   ] | 9892.986    | N/A            [N/A     ] [1.704   ] [110.838 ] [582      ] [N/A     ]
```

Figure 48: Performance analysis of fixed-wing legacy firmware at 100Hz

```
~ [~/dev/paparazzi/sw/tools]1 perf_profil*± ./perf_profil/perf_profil.py ~/work/logs/perf_profil/rotorcraft_test1.LOG
name                (samples)    | period (us)    [std      ] | freq (Hz)   | duty (us)      [std      ] [min      ] [max      ] [nb over ] [%         ]
main                (61661)      | 1001.00        [2.579    ] | 999.006     | 7.238          [0.395    ] [6.949    ] [11.245   ] [0        ] [0.7231    ]
modules_sync        (61660)      | 1001.00        [2.642    ] | 999.006     | 5.551          [0.992    ] [4.833    ] [11.579   ] [0        ] [0.5545    ]
radio               (3692)       | 16700.89       [458.302  ] | 59.877      | 1.476          [0.078    ] [1.213    ] [3.403    ] [0        ] [0.0088    ]
failsafe            (1232)       | 50001.00       [2.108    ] | 20.000      | 1.747          [0.154    ] [1.657    ] [3.653    ] [0        ] [0.0035    ]
electrical          (616)        | 100001.00      [2.474    ] | 10.000      | 1.484          [0.010    ] [1.444    ] [1.486    ] [0        ] [0.0015    ]
telemetry           (61660)      | 1001.00        [3.055    ] | 999.006     | 4.056          [8.676    ] [2.949    ] [154.773  ] [0        ] [0.4052    ]
event               (555*1000)   | 110.89         [0.290    ] | 9018.073    | N/A            [N/A      ] [1.954    ] [129.847  ] [1        ] [N/A       ]
```

Figure 49: Performance analysis of rotorcraft legacy firmware at 1000Hz

The same work with the new architecture shows that the tasks are (almost) harmonized and reflect the reference architecture presented in the next section. The timings are all aligned with the base frequency and the 'event' polling function is called at the expected rate. What is not visible but is a result of this work is that the time between the sensor task (start reading and getting data from digital sensors) and the rest of the guidance and control loop is fixed and provides a correct sequencing of the data flow regardless of the user configuration.

```
[~/dev/paparazzi/sw/tools/perf_profil] perf_profil_grouped(21)+*± ./perf_profil.py ~/work/logs/perf_profil/FW_final_100Hz_test1.LOG
name                (samples)    | period (us)    [std      ] | freq (Hz)   | duty (us)      [std      ] [min      ] [max      ] [nb over ] [%         ]
sensors             (6765)       | 10001.00       [6.198    ] | 99.990      | 11.725         [0.663    ] [10.954   ] [17.671   ] [0        ] [0.1172    ]
estimation          (6765)       | 10001.00       [2.322    ] | 99.990      | 1.005          [0.000    ] [1.005    ] [1.005    ] [0        ] [0.0100    ]
control             (6765)       | 10001.00       [2.322    ] | 99.990      | 10.373         [0.675    ] [5.157    ] [16.634   ] [0        ] [0.1037    ]
default             (6765)       | 10001.00       [2.486    ] | 99.990      | 1.271          [0.074    ] [1.259    ] [3.606    ] [0        ] [0.0127    ]
core                (6765)       | 10001.00       [6.307    ] | 99.990      | 3.019          [15.511   ] [1.319    ] [164.949  ] [0        ] [0.0302    ]
telemetry           (6764)       | 10001.00       [22.753   ] | 99.990      | 25.503         [39.428   ] [3.389    ] [246.671  ] [0        ] [0.2550    ]
event               (676*1000)   | 100.10         [0.128    ] | 9990.384    | 1.376          [N/A      ] [1.426    ] [78.324   ] [0        ] [N/A       ]
```

Figure 50: Performance analysis of new fixed-wing legacy firmware at 100Hz

```
[~/dev/paparazzi/sw/tools/perf_profil] perf_profil_grouped(21)+* 1± ./perf_profil.py ~/work/logs/perf_profil/RC_final_1000Hz_test1.LOG
name                (samples)    | period (us)    [std      ] | freq (Hz)   | duty (us)      [std      ] [min      ] [max      ] [nb over ] [%         ]
sensors             (37125)      | 1001.05        [23.506   ] | 998.952     | 7.256          [1.379    ] [6.356    ] [16.190   ] [0        ] [0.7249    ]
estimation          (37124)      | 1001.05        [19.081   ] | 998.952     | 1.014          [0.026    ] [1.014    ] [3.236    ] [0        ] [0.1013    ]
control             (37124)      | 1001.05        [19.081   ] | 998.952     | 1.459          [0.306    ] [1.412    ] [6.236    ] [0        ] [0.1458    ]
default             (37124)      | 1001.05        [19.083   ] | 998.952     | 8.936          [0.386    ] [8.778    ] [15.208   ] [0        ] [0.8927    ]
core                (37124)      | 1001.05        [23.702   ] | 998.952     | 1.665          [4.753    ] [1.458    ] [163.426  ] [0        ] [0.1663    ]
radio               (2249)       | 16500.99       [15.135   ] | 60.602      | 1.390          [0.109    ] [1.310    ] [3.704    ] [0        ] [0.0084    ]
failsafe            (741)        | 50001.00       [16.552   ] | 20.000      | 1.360          [0.091    ] [1.352    ] [3.829    ] [0        ] [0.0027    ]
telemetry           (37124)      | 1001.05        [24.641   ] | 998.952     | 4.915          [14.848   ] [3.051    ] [221.352  ] [0        ] [0.4910    ]
event               (370*1000)   | 100.25         [0.225    ] | 9975.061    | 4.469          [N/A      ] [1.634    ] [79.921   ] [0        ] [N/A       ]
```

Figure 51: Performance analysis of new rotorcraft legacy firmware at 1000Hz

6.6.3 Design and Implementation

The new architecture corresponds to the architecture shown below. The grey boxes are tasks grouping related functionalities: sensors, estimation, control, actuators, payload, communication, etc. The components of the systems, known as **modules**, are described and configured thanks to an XML file, providing the relevant information:

- module name, task group
- documentation
- dependencies (required modules and functionalities, provided functionalities, conflicts)
- initialization, periodic and event based functions to be called
- source files and compilation flags
- testing flags for unit tests

The module generator is using a topological tree search algorithm to solve the module dependencies, and then is generating C code for calling the functions, including the static schedulers for the periodic functions according to the system base frequency.

The final sequence is controlled by the static dispatcher to guarantee that each task is called within a predefined order.

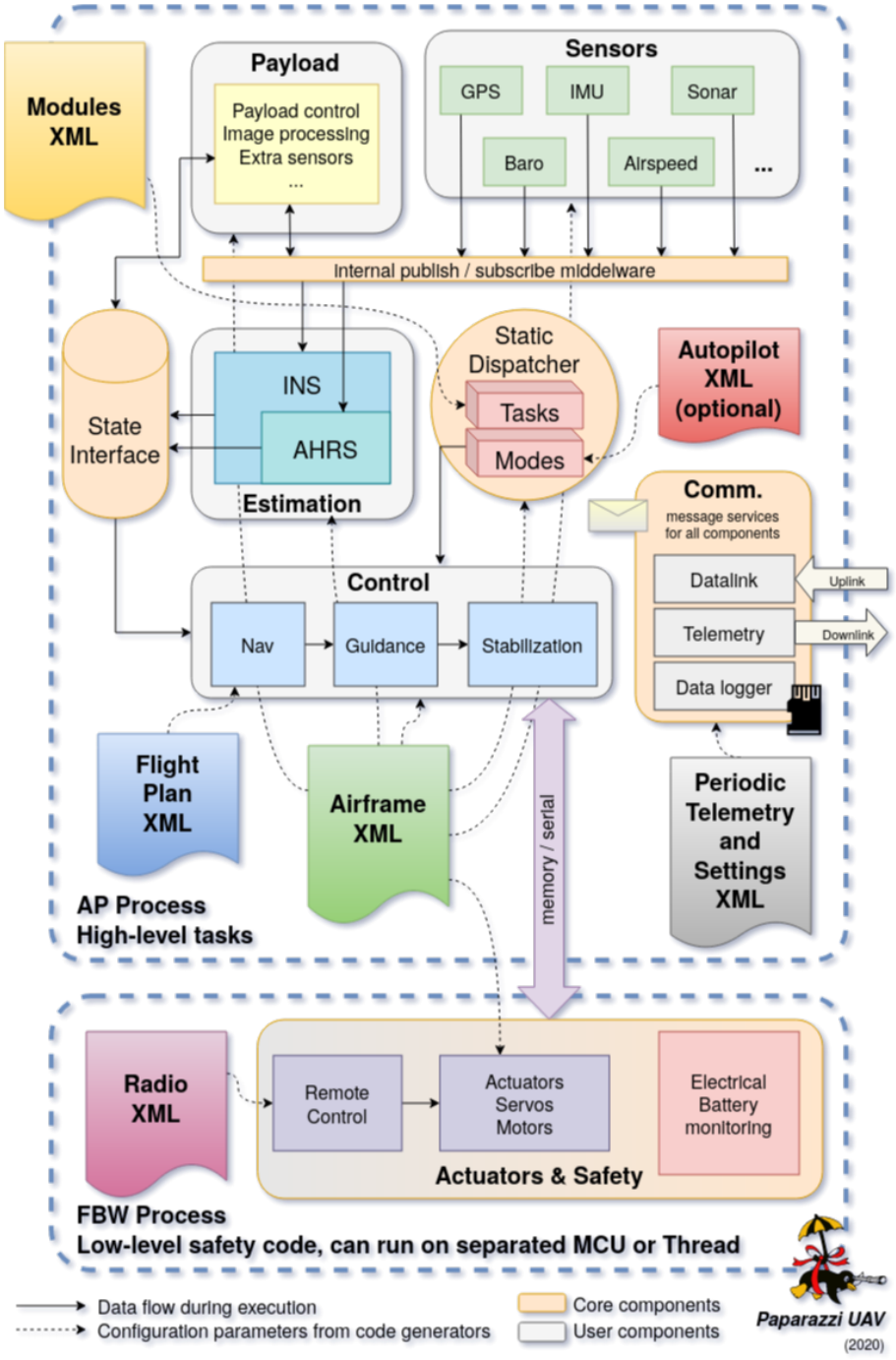


Figure 52: Paparazzi modules

6.7 WP3-14_1 – Control component for implementation of potential barriers

Levels	Function
Require	Proximity sensor
Provide:	Geofencing, obstacle detection and avoidance, geofence preservation, safe trajectory
Input	X, Y, Z coordinates of the drone Roll, pitch and yaw angles of the drone Distance to the obstacle from sensor proximity detection Geofence virtual boundary information
Output	Required X, Y, Z torques and thrust or rotors' speed
C4D building block	Geo-fencing and geo-awareness
TRL	3

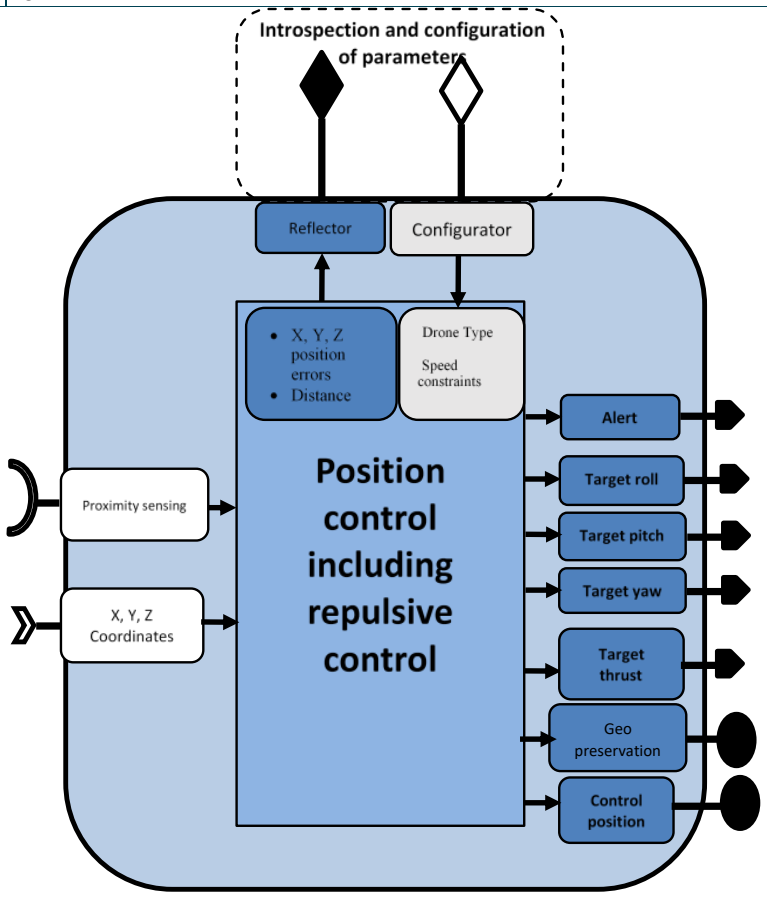


Figure 53: Building Block diagram for WP3-14_1

6.7.1 Detailed Description

The growing popularity of small civilian drones has generated a wide array of complex and unprecedented challenges related to the risks posed to security, safety, potential to disrupt people's privacy and interfere with activities on the ground and with the manned aircrafts. This necessitates the development of technologies that allow UAVs to safely navigate according to the regulations.

Geofencing technique can be used to specify no-fly zones for drones. It is a technique that defines virtual boundaries in a specific geographical area. Once these virtual boundaries are defined, the drone should never be able to penetrate through these boundaries. In other words there should be a potential

barrier that prevents drones from crossing the geofence boundary and redirect their trajectory to avoid any conflict.

The drones must not only respect the geofence but should also be capable of detecting and avoiding any obstacle in their path. The obstacles could be stationary (e.g., buildings) or moving objects (e.g. birds, other drones or manned aircrafts).

A drone could either be autonomous or remotely controlled by an operator. In both cases, the potential barriers must repel the drone automatically and generate a warning of alert whenever it is near the restricted area or any obstacle in path that could cause a collision.

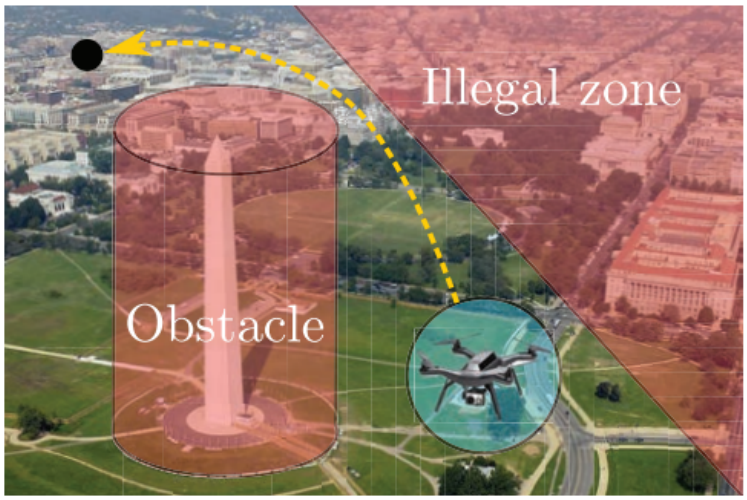


Figure 54: Geofencing example

6.7.2 Specifications and contribution

The purpose of this component is to provide a mechanism which ensures that the collision between the drones and obstacles never happens and geofence is always preserved.

From a technical viewpoint, preventing drones from violating the boundaries defined by the geofence system can be considered as a constrained control problem. Constrained control addresses the problem of enforcing constraints satisfaction at all times while ensuring that control objectives are achieved. It is required that the geofence information is provided to the drone and that information could be updated online while the drone is in operation.

In order to avoid collision with an obstacle, it is assumed that drone is equipped with proximity sensors and can detect any relative position of both non-cooperative and cooperative entities within a sensing range. A cooperative entity is another drone which has the same capability. In contrary, a non-cooperative entity is an entity without collision avoidance system.

Sensing capability is required to sense the presence of any other entities in its close vicinity which may lead to a collision. These sensors only give the relative position of any entity in its range in the local frame and do not provide position information in the global frame. It is to note that this assumption is used for the purpose of collision avoidance only.

6.7.3 Design and Implementation

A potential barrier for geofencing and collision avoidance can be achieved using Artificial Potential Function (APF). In APF, a drone is considered as a point in a potential field. This drone experiences a repulsion force from the obstacles or geofence and therefore, instead of colliding with them, it steers away from them.

Typically, potential functions are based on the relative distance between drone and the obstacle or the geofence and do not require any global information. Based on the practical aspects, an ideal potential function must have the following properties:

- The range of the potential field must be bounded. Usually, it depends on the range of obstacle sensors mounted on the agent.
- The value of the potential field and the corresponding repulsion must be infinity at the boundary of the obstacle/geofence and must decrease with the increase in the distance
- First and second derivatives of the potential function must exist in order to have a smooth repulsion force

APF based repulsion mechanism is combined with the control algorithm, for instance it could be combined with the position control of the drone. The repulsive force remains zero when the distance is greater than some predefined value and position controller works normally. However, when the distance becomes less than the threshold, the repulsive force comes into play and lesser the distance more will be the repulsive force.

Figure 55 shows a graph of distance based potential function.

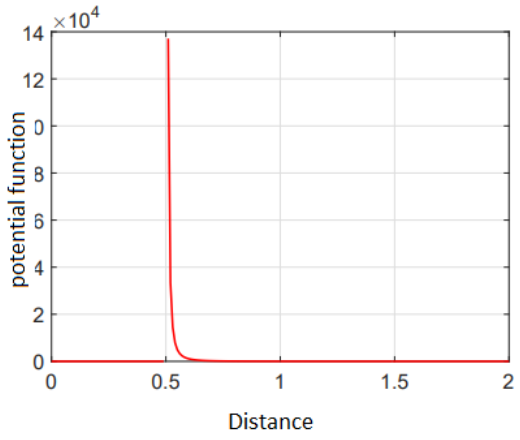


Figure 55: Potential function for collision avoidance and geo fencing

The detection range is considered as 2 meters and the protection radius around the drone is 0.5 meters. It can be observed that the value of the potential field increases as the distance between the fence/obstacle and the drone. This will act as the repulsive force and push the drone away from the fence/obstacle.

6.8 WP3-14_2 – Multi-agent swarm control

Levels	Functional
Require	Communication service to obtain data from the neighbours
Provide	Control algorithm for swarm formation control and cooperation, Inter-agent collision avoidance, Obstacle detection and avoidance, as well as mathematical stability proof of formation tracking
Input	X, Y, Z coordinates of the drone Roll, pitch and yaw angles of the drone X, Y, Z coordinates of neighbours Formation vector Possibly proximity sensors
Output	Required roll and pitch angles and thrust if used standalone, or rotors' speed if used jointly with the attitude controller
C4D building block	Flight guidance if standalone, both Flight guidance and Flight control if used jointly with the attitude controller
TRL	4

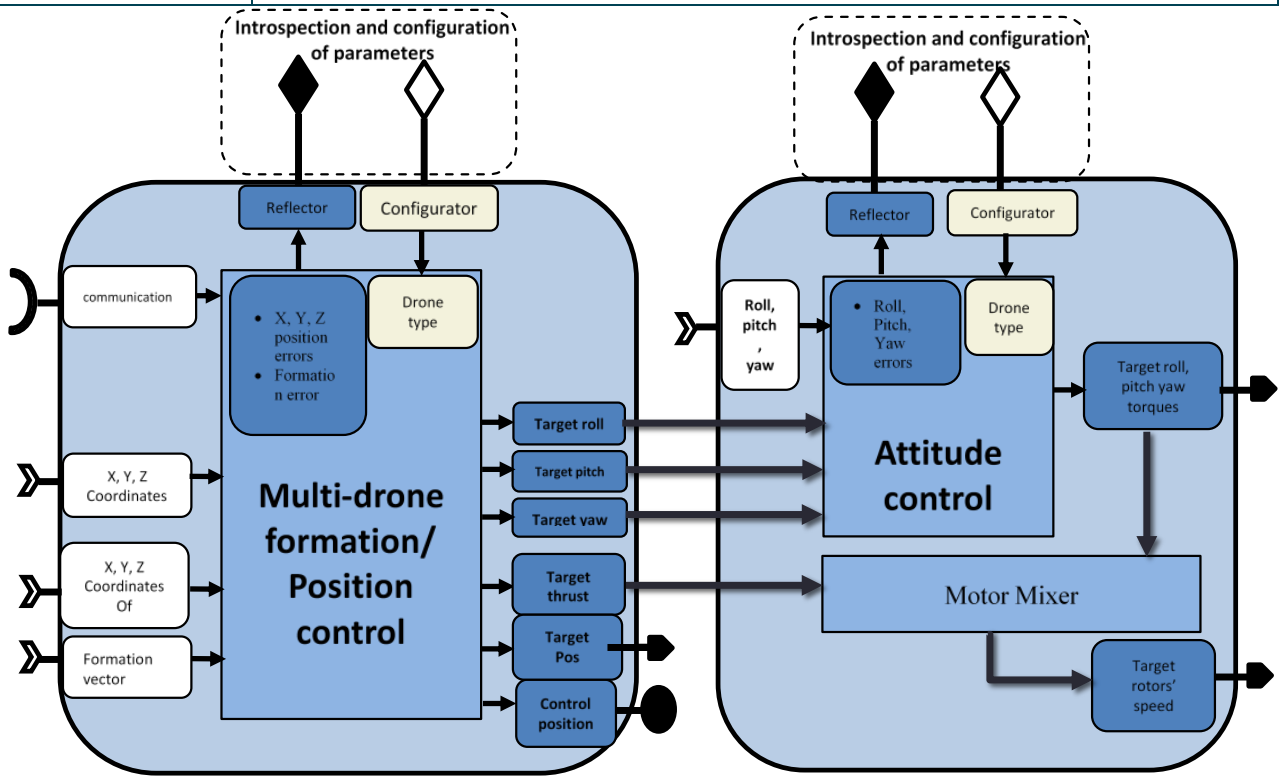


Figure 56: Building Block diagram for WP3-14_2

6.8.1 Detailed Description

The effectiveness of the drones can be further improved by utilizing them in a cooperative manner to accomplish much more complex tasks which cannot be realized by solo operation. Control of such cooperative teamwork of Multi-Agent Systems (MAS) heavily depends on communication and information exchange among the agents. This gives rise to two natural choices of communication network which are centralised and distributed control network. In centralized communication, as apparent from the name, there is a central unit to which all the agents are connected to and send their information. The central station handles all the information exchange and sends control commands to the agents. This kind of scheme has some major drawbacks. For example, if the central unit fails, the

whole network will collapse. Moreover, a centralized control scheme cannot handle large number of agents due to network saturation and/or limited computing and processing capability etc. On the other hand, a distributed control scheme does not require a central control unit. Agents communicate directly with their neighbouring agents and exchange information. The information received from neighbours is also called local information. Each agent in the distributed network uses this local information to compute its own control input. Distributed control schemes offer many advantages over the centralized control scheme in terms of efficiency, adaptability, robustness, and scalability.

In many practical scenarios, it is required that the agents of MAS create and maintain a desired geometric shape. The required shape could either be fixed or time varying. In some cases, it is further required that the agents follow a trajectory while maintaining the shape. The reference trajectory is produced by a virtual or real leader. This is known as formation tracking.

6.8.2 Contribution and Improvements

The main goal of this enabling technology is to offer a distributed formation tracking controller for a swarm of drones. From the state of art, it is clear that the available cooperative control schemes do not take various practical limitations in-to account. Motivated by this, we focused on the design and implementation of distributed cooperative control laws for a swarm of drones with communication and sensor constraints described below in details.

- Generally, formation controller requires both position and velocity of the neighbours for its computation. However, it is not always possible to measure all the states in real applications. Moreover, sending partial information consumes less bandwidth and thus is very cost effective. Therefore, it is considered that only position information is shared between the agents. We consider that velocity and control input of neighbours are not available.
- The data is transmitted between the agents at irregular and asynchronous time instants. Here, asynchronous means that the time of the transmission of the position data by one agent does not depend on any other agent in the network. In other words, an agent can receive information from its neighbours at different time instants. In fact, having irregular and asynchronous time periods in any practical case is totally inevitable.
- Another constraint that is taken into account is that the communication between the agents could either be directed or undirected. Directed communication means that if agent *i* can receive information from agent *j* then it does not imply that agent *j* can receive information from agent *i*.
- Only a few agents in the network have the access to the reference trajectory produced by the virtual or real leader

Since inter-agent collision is another important issue in formation tracking control, a potential function based collision avoidance algorithm is incorporated with the proposed formation tracking controller. The collision avoidance algorithm ensures that the drones converge to produce the desired geometric shape without colliding with each other. The desired geometric shape can be defined by a formation vector.

6.8.3 Design and Implementation

Since the drone is an under-actuated system (e.g. quadrotor has 6 states and 4 controllable rotor speeds), the design of controller is not straightforward. To achieve the desired formation of a swarm of drones, the overall dynamics of the drone is divided into two blocks namely position dynamics and attitude dynamics with a strong nonlinear coupling among them. The idea is to introduce an auxiliary position controller $\mu = [\mu_x; \mu_y; \mu_z]^T$ and design it while keeping all the above mentioned constraints in mind. The required thrust and attitude then can be computed through this auxiliary position controller. An attitude controller then can be designed to achieve the required attitude by the drone and which makes the drone to reach to the required position to achieve the desired formation shape. Figure 57 shows the block diagram of the overall control architecture of one drone to achieve formation tracking of the swarm of drones. The position controller computes auxiliary position controller μ which is then

used to compute desired roll and pitch through nonlinear decoupling equations. These desired roll and pitch angles act as a reference input to the attitude controller which computes the required torques to track these desired angles.

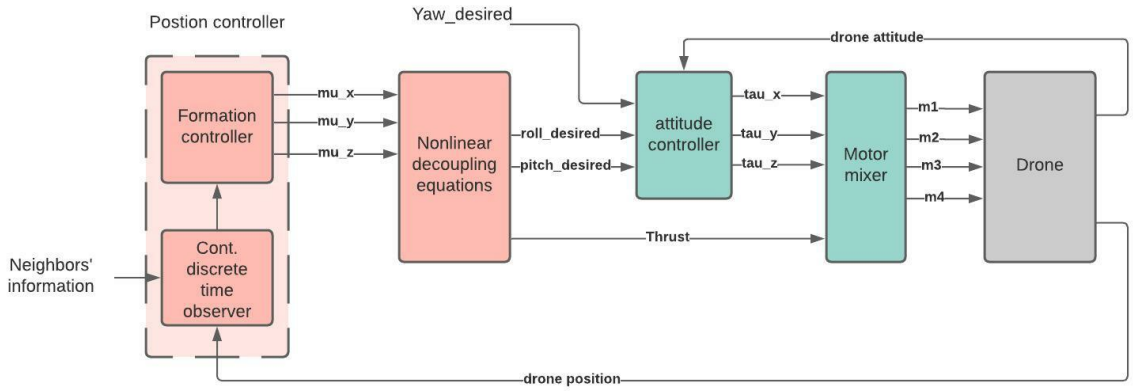


Figure 57: Overall control scheme for formation tracking

Formation controller: The basic idea is to use a classical continuous linear consensus controller for MAS of second order dynamics. By introducing the auxiliary controller, the position dynamics can be represented as double integrator. However, trivial formation controller requires both position and velocity in continuous time. A continuous-discrete-time high-gain observer is used to compute both position and velocity from available discrete position data. Each follower has a local observer which not only estimates the agent's own states but also the states of its neighbours. The structure of the proposed observer-based formation tracking algorithm also has some advantages when it comes to communication delays and data packet dropouts. As each controller is using the estimated states provided by the local observer, by time stamping the measured position data, the estimation can be provided as soon as the data is received, even with a delay, by compensating it through increasing the computation speed of the observer. Moreover, if the information packet is lost during communication, the observer could still provide an estimation.

An Artificial Potential Function (APF) based repulsive controller is also incorporated with the formation controller to avoid any kind of collision among the agents and with obstacles. APF provides a repulsive force whenever the distance between two bodies becomes less than some threshold. It is considered that each drone is equipped with some kind of proximity sensor to detect anything in its path.

Attitude controller: For the Matlab simulations, the desired attitude is computed through the auxiliary controller and then a finite-time attitude controller is used to compute the required torques to achieve this desired attitude.

Simulations: The designed control scheme is simulated for a group of one leader and three follower quadrotors where followers are required to make a triangle around the leader in xy-plane and maintain the same altitude as of the leader. Figure 58 illustrates the formation with a stationary leader while Figure 59 shows formation tracking results with a moving leader. The threshold distance for the repulsive force is considered as 2 meters while the protection radius around each drone is 0.5 meter. Figure 60 shows that the inter-agent distance always remains greater than 1 meter which implies that the agents never collide during the whole process.

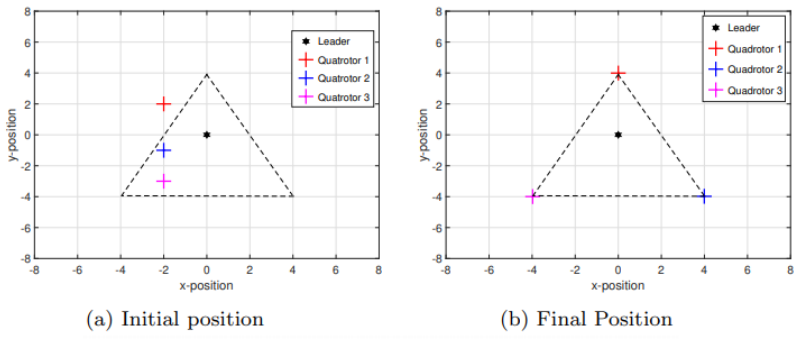


Figure 58: Formation tracking with stationary leader, distances in meters

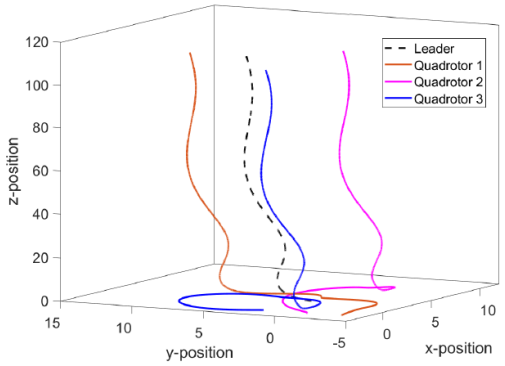


Figure 59: Formation tracking with a moving leader, distances in meters

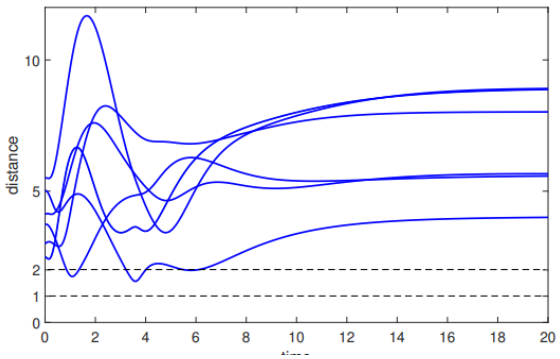


Figure 60: Inter-drone distance, in meters

The simulation results show the stability of the system with the designed algorithm for formation tracking. However, simulation results are not sufficient to prove the stability of the system. Therefore, a rigorous mathematical proof has been also carried out to ensure the formation tracking and stability of the closed loop system with the designed controller.

6.9 WP3-15 UWB-based indoor positioning system

Levels	Platform, Function
Require	Energy, Raw sensed data from UWB transceiver and IMU
Provide	Navigation Sensor
Input	Raw sensed data from UWB transceiver and low-cost INS
Output	Position (main focus), attitude
C4D building block	UWB-based indoor positioning system
TRL	4

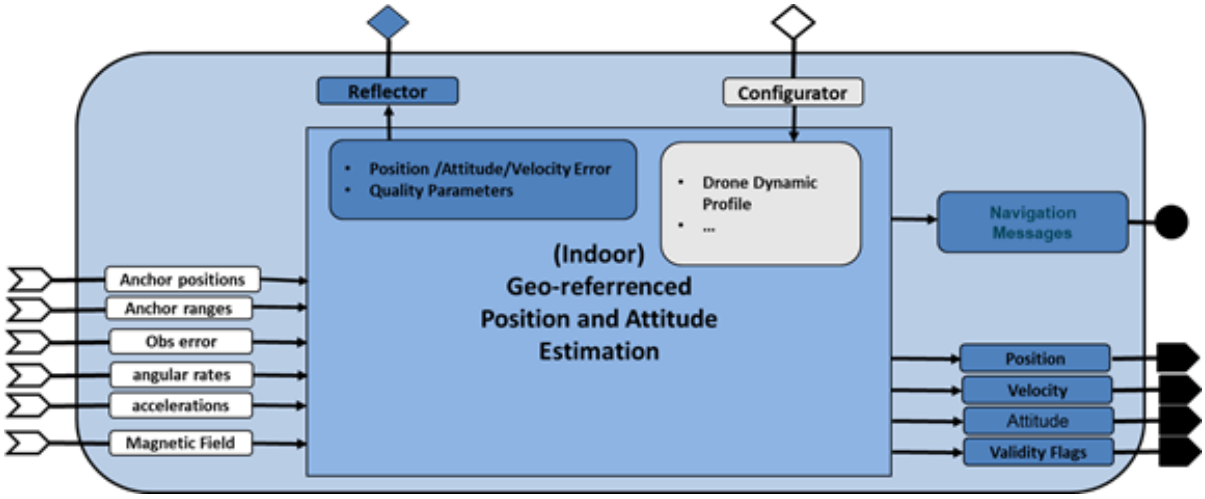


Figure 61: Overall UWB-based indoor positioning system block (as reported in D2.3)

6.9.1 Detailed Description

In COMP4DRONES, ACORDE is developing an Indoor Positioning System (IPS), a solution for the challenges posed by the construction use case, demo tunnel (UC2-demo 2). ACORDE IPS corresponds to the building block in section 6.3.1 of D3.2 [24]. ACORDE IPS is specifically oriented to serve reliable and precise geo-referenced position to a drone flying on a tunnel under construction, in charge of the capture of raw data for tunnel digitization, to be employed for Building Information Modelling (BIM) [25]. The objective of the ACORDE IPS is two-fold, i.e., to ensure a safe navigation along a planned path, and to optimize the accuracy of the navigation data (and of raw UWB ranges) synchronized with the thermography or LIDAR capture so that it can be exploited in the offline digitalization process associated to the BIM.

The specific environment and the usage scenario posed by the construction stakeholder (ACCIONA) and the drone integrator&operator (FADA-CATEC), configures a set of constraints in terms of cost of the solution, power consumption, size, weight, and precision (reported in [26]and [27]) which, added to the specific geometry of the indoor infrastructure, has leaded ACORDE to a novel and improved IPS solution. ACORDE IPS solution was first introduced in [28]. Here it is briefly overviewed and represented in Figure 62 (update of Fig.42 in [28]) for reader convenience. It is based on mounting a “tag node” on the indoor drone, in order to measure distances to a set of strategically deployed “anchor nodes”, called ranges. The tag will rely on an INS sensor for providing a processed tag position. In addition, the tag can also provide range data (if required). A standard Mavlink interface supports the provision of these data to the drone navigation system.

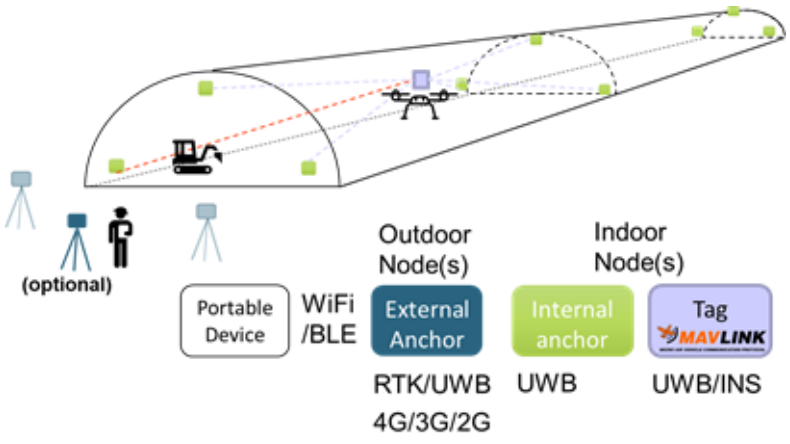


Figure 62: Indoor Positioning System developed by ACORDE

The posed solution contributes improvements and innovations accounting the scenario and the current state of the art (as explained in section 1.2). The solution is also “complete” in the sense that ACORDE covers the design&implementation of the solution (anchor&tags platforms and their firmware). Moreover, ACORDE has also improved its design flow by developing the IPS-MAF tool. All this activity, comprised by technical packages WP3-WP6, can be summarized as follows:

- Design and implementation of anchor and tag platforms (including the required board support package (BSP) (WP3-15_2).
- Design and implementation of anchor and tag firmware, including the configuration, positioning algorithms and interfacing (WP4-17).
- Ensuring a robust and enriched communication among anchors, and among the tag (within the drone) and the anchors, for a more robust and improved positioning (WP5-19-ACO).
- Developing an Indoor Positioning System Modelling and Analysis Framework (IPS-MAF) for indoor structures, specifically tunnels (WP6).
- Providing at the tag a Mavlink [29] interface to support providing both processed positioning information, and “pre-processed” information (ranges) (WP3, WP4, and WP5).

Current Status

Up to the report date, ACORDE has advanced in parallel in all the afore mentioned tasks. The current advance can be summarized as follows:

- For anchor and tag platform design, ACORDE has already decided the ranging technology, i.e., Ultrawideband (UWB), and the module sustaining the implementation (DW1000 from Qorvo [DW1K]). ACORDE has also evaluated the basic performance of this technology, after acquiring an evaluation kit. Moreover, ACORDE has already decided on main components of the anchor and tag architectures (e.g., microcontrolers), and provisioning them (which has turned out to be critical during these COVID’19 times).
- Related to application design, ACORDE checked code available with acquired evaluation kit and its unsuitability for the posed use case. ACORDE has been already able to start to develop and test modified trilateration algorithms for anchors autopositioning and tag positioning on top of IPS-MAF simulations.
- The assessment, implementation and basic test of a novel and improved Medium Access Control (MAC) and ranging protocol for single-tag scenario has been also performed.
- A former version of IPS-MAF supporting static and simulation-based analysis is ready.
- A Mavlink library ready for supporting the tag to provide position is ready. A Mavlink extension is also proposed for enabling the tag providing the ranges.

This is allowing ACORDE tackling the final design and implementation of the anchor&tag platforms, firmware implementation and validation (according to the plan in [D1.2]).

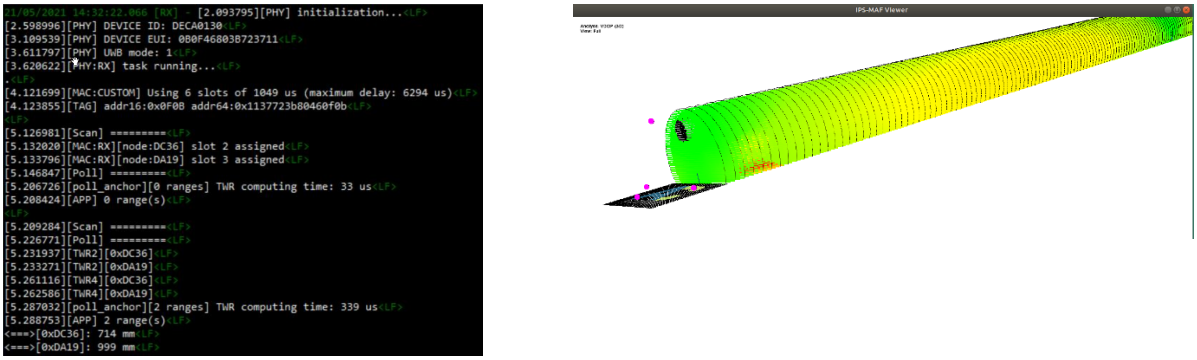
Contribution and Improvements

The stakeholder (ACCIONA) described the digitization scenario at a specific time slot where the construction 7days/24hours activity stopped for 1-2hours. At that time, it is possible to lie some “mobile” anchors, together with the “fixed” anchors, and later let the drone perform a pre-programmed digitization flight, where the tunnel under-construction can present some eventual obstacles (e.g., machines, signals). The posed scenario presents several challenges, many of them specific in relation to other existing real-time location systems:

- The specific long geometry of tunnel (which challenges UWB ranges, cost of the solution and the validity of conventional trilateration algorithms).
- Providing features for a flexible and agile anchor deployment, which does not oblige surveyors to geo-positioning all anchors, especially mobile ones.

- Need to provide real-time, 3D geo-referenced positioning on the tag (while most solutions focus on 2D relative positioning, commonly computed on a ground platform which centralizes data from anchors).
- Cost and energy/power optimization (the latter important for fixed anchors)
- Need for system-level design tooling. The IPS is a complex system with many parameters and aspects (specific deployment, transmission powers, sensitivities, latencies of the ranging phases, algorithmic alternatives, etc) with potential significant impact on the overall performance. Means for facilitating a holistic design, accounting for those many different aspects at early design stage are required even for experienced designers in the field.

The same scenario also presented the opportunity to exploit the fact that during digitization flight only one drone, i.e., one tag needs to be serviced. The question is how to exploit this single-tag assumption for a safer flight and better digitalization.



a) A capture of a test of the ATSA-DRA protocol on the development kit.

b) IPS-MAF showing Vertical DOP at the initial part of a tunnel for a specific anchor deployment and considering obstacles.

Figure 63: Some results at the current status of ACORDE IPS system Development.

The ACORDE solution tackles these challenges and opportunities. So far, a number of contributions in comparison to other existing solutions are highlighted:

- Customized, cost-effective anchor and tag platforms, specifically designed to cope with the computational needs (specially for tag) and energy/power efficiency needs (specially anchors).
- Novel MAC protocol called Asynchronous Tag trigger, Slotted Anchor response with Deterministic and Random Allocation (ATSA-DRA), specifically adapted to the single-tag assumption, that ensures deterministic latency, while optimizing the number of anchors in view. This protocol has been already implemented and formerly tested on the evaluation kit as reported in [30].
- At application level, the solution enables geo-referenced, real-time 3D positioning. Moreover, the application overcomes the challenges of the tunnel geometry. A modified trilateration algorithm has been already developed which enables positioning in regions of limited anchors visibility (coverage) and poor dilution of precision (DOP), where a conventional least-squares based approach is not working. This is specifically required for enabling anchors auto-positioning at the initial phase.
- The development of IPS-MAF, is a qualitative step on ACORDE capabilities for tackling custom design of IPS systems for long indoor infrastructures (tunnels, mines, large pipes, …). It enables a newer system-level design flow (as explained later). It is remarkable that IPS-MAF already served to detect the afore mentioned issues on the usage of conventional least-squares algorithms, and to design and test the modified trilateration algorithm, even before the anchor or tag platforms are ready. IPS-MAF reveals not straightforward outcomes, e.g., notice the “coverage” pattern and the “shadow” provoked by obstacles in Figure 63b.

- The support of a Mavlink interface at the tag side, to provide both completely processed (e.g., position data) and partially processed (e.g., ranges). The latter enable the offering ACORDE IPS as a complementary positioning sub-system, which integrators can fuse with other alternative technologies. Moreover, ranges transfer is a “smooth” (in the form of dialect) Mavlink extension contributed by ACORDE in COMP4DRONES [30].

Design and Implementation

ACORDE IPS development was started in COMP4DRONES, motivated by the needs of the stakeholder (ACCIONA) and drone integrator (FADA-CATEC), by relying on the expertise of ACORDE on outdoor positioning, and on embedded HW and SW development. The statement of the solution requirements and the wish list from the stakeholder and the drone integrator, made early evident that a more holistic, model-based design was required. It motivated the development of the IPS-MAF, which, as shown in Figure 64, can be now used at early design stage to build up a holistic model of the IPS system. IPS-MAF can be used to analyse and decide key aspect at different levels of system (deployment of the anchors, sensitivities and transmission powers, transmission frequencies, etc) while keeping a holistic view of the system.

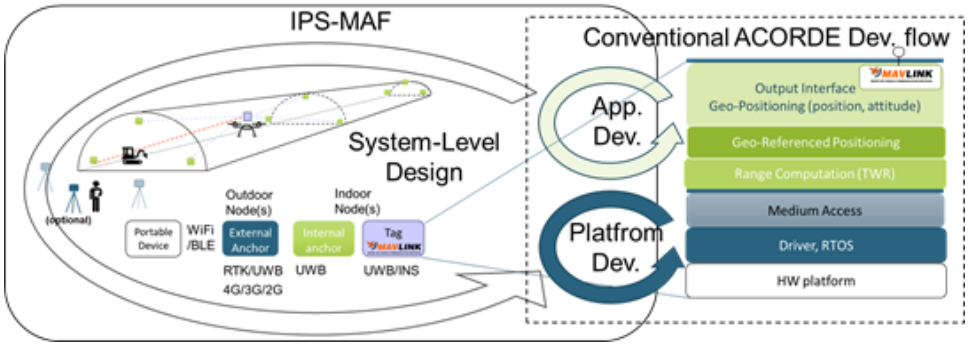


Figure 64: IPS-MAF provides a qualitative step towards a holistic, model-based design of Indoor Positioning Solutions for long infrastructures

That is, IPS-MAF enables a holistic model of the IPS, while integrating some actual pieces of the application, so in that sense it also enables to advance some part of the application development. Therefore IPS-MAF feeds the conventional platform development and embedded application development phases, where ACORDE has already long expertise. At the same time, the measurement and characterizations done for platform and application development serve to feedback and polish the holistic model. Summing up, an extended system-level design flow has been enabled, after coupling IPS-MAF to conventional ACORDE development processes for platform development (which includes PCB design, mechanical design, drivers’ development, embedded application development) and application development (where ACORDE typically develops in C/C++, relying on a GUI based (typically Eclipse), microcontroller specific cross-development environment.

6.10 WP3-15_X Geo-referenced Position and Attitude Estimation

Levels	Platform, Function
Require	Energy, Raw sensed data navigation messages of GNSS receivers, raw data from low-cost IMU and barometer.
Provide	Navigation Sensor
Input	Raw sensed data from UWB transceiver and low-cost INS
Output	Position (main focus), attitude
C4D building block	Geo-referenced Position and Attitude Estimation System
TRL	5

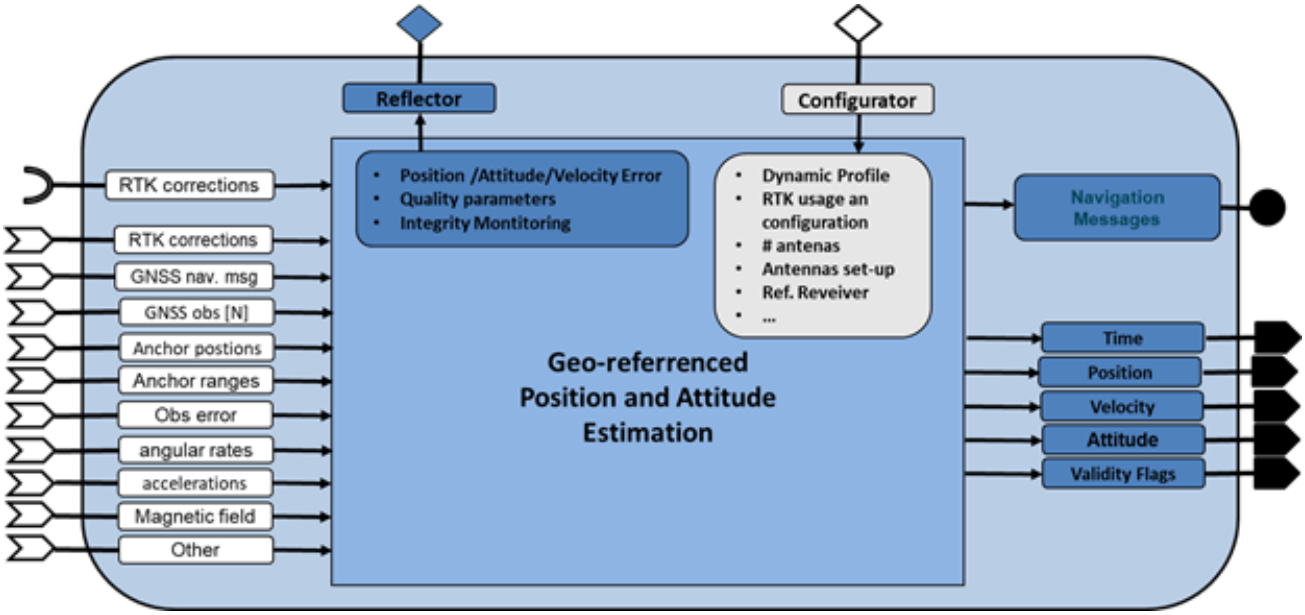


Figure 65: Geo-referenced Position and Attitude Estimation system block (as reported in D2.3)

6.10.1 Detailed Description

In COMP4DRONES, ACORDE is developing GLAD+, and outdoor geo-referencing system which abides to the "Geo-referenced Positioning and Attitude estimation system" block introduced in the COMP4DRONES architecture introduced (section 8.11 of D3.2 [24]), and whose representation is reproduced in Figure 65 for convenience. GLAD+ is an improved version of its predecessor GLAD (GNSS-based Low-Cost position and Attitude Determination system). Section 1.2 sums up the contributions and improvements brought by COMP4DRONES with respect to GLAD.

GLAD+ is specifically oriented to provide drones quality navigation information (position, velocity and attitude) at a reduced cost in challenged outdoor scenarios. Drone scenarios expose position/attitude estimation systems to challenging conditions (e.g., shadows, more challenging dynamics than land-vehicles, calibration constraints), and stringent cost, size and weight requirements, as the ones reported in [26] and [27]. As GLAD, GLAD+ is a "complete" solution from ACORDE, in the sense that ACORDE performs both application and platform design and development (including a COTS based HW design). In addition to a significant upgrade of its positioning products, in COMP4DRONES ACORDE is also aiming the improvement of its productivity on the modelling, design, implementation and validation procedures of these type of systems.

ACORDE is tackling the design and development of GLAD+, and the improvement of the related design flow in different parallel activities:

- The design of and improved HW/SW platform for the navigation solutions (WP3-15_1). It includes a new HW platform with improved capabilities on GNSS receivers, and also the assessment of license-free real-time RTOS on top.
- Enhancement of the navigation software, for adapting it to the new platform and to apply algorithmic improvements, including the assessment of possible AI based improvements (WP4-16).
- Providing support of anti-jamming and anti-spoofing features (WP5-11-ACO).
- Providing an improved/extended interface for a smoother integration on drone systems (activity globally associated to WP3, with implications in WP4 and WP5).

Current status

So far, ACORDE has advanced in parallel in all the afore mentioned tasks. The current advance can be summarized as follows:

- The new HW platform for GLAD+ has been designed and implemented (shown in Figure 66a). The new HW platform includes new low-cost GNSS receivers, with multi-constellation and anti-jamming and anti-spoofing capabilities.
- A custom Linux Real-Time port for the GLAD+ HW platform has been developed. This port relies on an updated *u-boot*, on *buildroot-2020.02-LTS* (generating a filesystem to be loaded as ramdisk), and on *kernel-rt-4.19.59* (includes RT-PREEMT patch).
- In past GLAD development, a platform abstraction layer was developed in order to separate platform dependent code (e.g., threads creation, interruption management, POSIX calls, call to drivers). This layer has been already updated to cover RT-Linux target.
- A preliminary version of the driver for making available jamming and spoofing events has been already developed and tested.
- A logger system has been updated and sent together the GLAD+ HW/SW platform ACORDE to the drone integrator in construction use case (UC2-demo1). ACORDE generated documentation for integration and supported the integrator in what resulted a smooth and quick integration.
- The integrator performed flight tests. ACORDE supported the integrator with additional documentation (i.e., for operation and tests of the integration, and for indicating specific maneuvers to let validation tests cover as many working conditions as possible).
- With that information, ACORDE has done some preliminary evaluation of the current algorithms, and tested the potential of some improvements, i.e.., multi-constellation support. Algorithmic improvement is in progress.

a. New HW/SW platform of GLAD+ b. Integration of GLAD+ on drone platform.

Figure 66: New HW/SW platform for the ACORDE outdoor geo-referencing system and integration on a drone platform int he COMP4DRONES for the outdoor demo of the construction use case

Contribution and Improvements

GLAD+ is expected to bring important improvements or contributions that can be summarized as follows:

- ***More interoperability***, given by the support of a Mavlink interface for providing the position estimates.
- ***Security features***, enabled by the anti-jamming and anti-spoofing capabilities
- ***Lesser cost***, a validated Linux-based port implementation means an important license cost saving with regard GLAD implementation.
- ***Better performance*** (higher resilience, precision, and integrity) of the positioning solution (including attitude).

- A proposal of ***smooth Mavlink extension*** to provide all attitude components, and to report jamming and spoofing events.

Figure 67a) shows the boot of the custom RT Linux distribution developed by ACORDE for GLAD+ platform. As can be seen, the kernel boots in little bit more 3 seconds (time labels are relative to kernel start). Taking into account the bootloader (not shown in the capture) took around4-5s, it leads to ~8s for booting the position&attitude estimation application (actually, the boot process goes on to get Ethernet link ready and launching SSH services, however, they are used for configuration or debugging, and the position&attitude application is launched before triggering). Further tests were conducted. Different resolution timers (from 100 to 1000 Hz) were tested. Moreover, the latencies of the scheduler (context switches) were estimated under heavy workload conditions, relying on the "cyclictest" [31] and "hackbench" [32] standard Linux utilities. An average latency 41 us, with 72 us maximum latency was obtained. This is a key figure for tick timer setting. For instance, 20 context-switches per second would take 1.4ms max. overhead, and less than 1ms on average, which seems affordable for GLAD+ application margins. Further study with final GLAD+ firmware is still required though.

a. Tests of the RT-Linux port on the new GLAD+ platform.

b. Check on static capture on how multi-constellation data enables more resilience and accuracy on attitude estimation

Figure 67: Some results of the activity on design and development of GLAD+ in COMP4DRONES

Figure 67b) shows the results of some preliminary analysis enabled by the data log from the integration tests of Dec. 2020. That test was performed in static conditions in front of the facilities of the integrator (FADA-CATEC). Specifically, Figure 67b) shows three stacked graphics with the three components of a "tail-head" baseline estimation, fundamental pre-processing for attitude estimation (heading and pitch components can be directly derived via a simple formula). The top one is a non-causal estimate, and thus only possible after offline post-processing, which is taken as a ground truth reference. Green colors reflect convergence of the ambiguity resolution algorithms (and thus resembles a precise estimate), while orange reflect no-convergence, and thus no-valid, in general, inaccurate estimate. The bottom graphic shows a "forward" (i.e., that can be computed in real-time) estimate of the baseline using only GPS constellation. As can be seen, most of the capture (97.9%) lacks convergence. The problem is related to the buildings around the drone at the capture time, which shadowed several satellites of the GPS constellation. The graphic on the middle, shows a forward estimate of the baseline using both GPS and Galileo constellations, as enabled by the GLAD+ HW/SW platform. This estimate gets the same 97.7% figure, but of convergence time, indeed very close to the 98.3% time achieved with non-causal processing.

Design and Implementation

As well as the significant improvements regarding GLAD, which are leading to a new improved low-cost geo-referenced position&attitude estimation system, i.e., GLAD+, in COMP4DRONES, ACORDE is making progress towards a qualitative enhancement of its design procedures. This piece of work is in direct relation to WP6 activities.

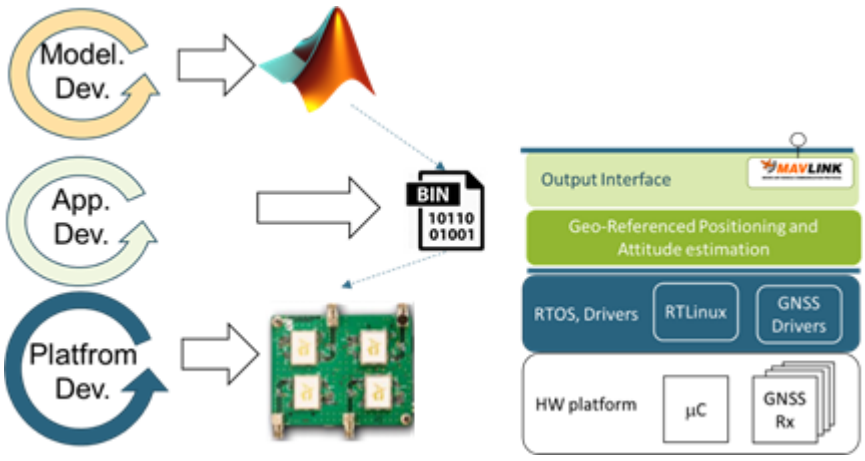


Figure 68: Design flow followed by ACORDE for its outdoor geo-referenced position&attitude estimation systems

Figure 68 shows the development methodology that ACORDE is applying for its georeferenced outdoor positioning systems. The methodology is relatively conventional in its two bottom layers. ACORDE has long expertise in platform development, which includes PCB design, mechanical design, drivers' development. ACORDE also has experience on embedded application development. GLAD, and so GLAD+ were and are developed in C/C++, on a microcontroller targeted cross-development environment, and relying on a GUI based environment (Visual Studio or Eclipse). On top of these two layers, ACORDE develops and tries different versions of the positioning algorithms by using a Matlab model. This model is fed with raw data logged on captures taken on integration tests and real flights. Additional Matlab scripts serve to test and compare output results.

This approach has proven to be useful and enables model building/refinement relying on raw sensor data logged at past captures. However, it has also some important disadvantages. One of them is associated to the conventional, sequential HW/SW development. HW platform design and implementation comes before driver development or RTOS port (if required). In turn, HW/SW platform availability is a pre-condition for application development. Once the application is ready, it is possible to evaluate if the performance (timing, memory, energy and power consumption, etc) and validate it. As well as the long latencies involved, the flow adds the risk of late finding of performance bottlenecks that involve a drastic re-design, of hardware in the worst case. To these drawbacks, we need to add the high cost of translation of Matlab models to C/C++ implementation, and the traceability problems generated at the validation time.

In COMP4DRONES, ACORDE is developing and evaluating a newer approach to overcome those drawbacks. It is sketched in Figure 69. The ESDE framework developed in WP6 [33], is encrusted in at a top level layer, for a system-level approach to embedded software design and development, to encompass the conventional HW/SW platform development capabilities and processes of ACORDE.

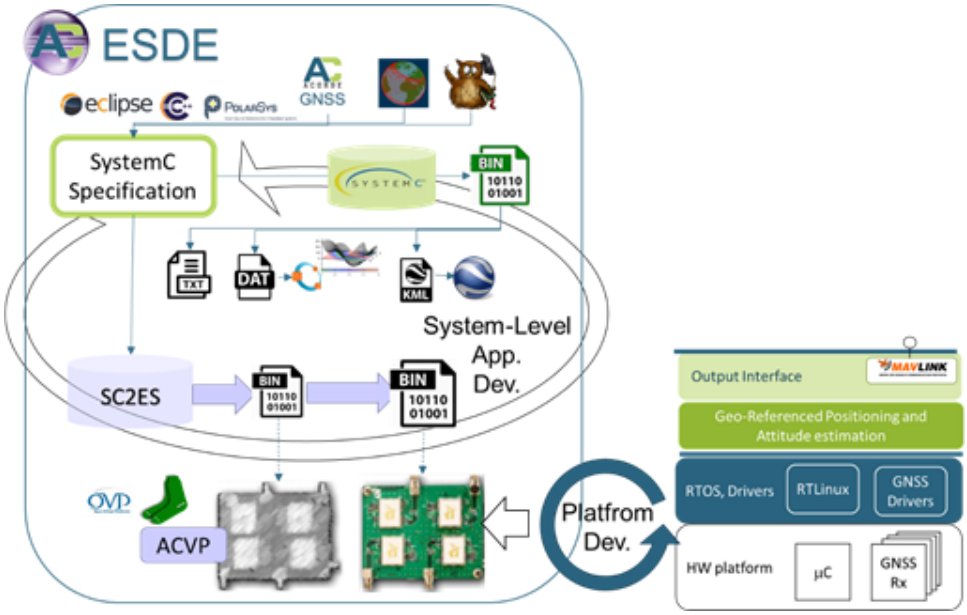


Figure 69: System-level methodology for the design and implementation of outdoor geo-referenced position&attitude estimation systems in COMP4DRONES

Some key aspects of the ESDE flow for productivity improvement are:

- The fast functional models that can be built at the top lever, able to significantly speed-up functional validation vs Matlab model execution.
- The automated embedded software generation mechanisms that avoid a significant translation effort from the system model to the implementation C/C++ code.
- The possibility to parallelize of the development of firmware (binary or object file), or very close version of it with HW development, by relying on a high-fidelity virtual platform.
- The possibility to validate firmware (without availability of the physical platform), eventually using several virtual platforms for test parallelization

More details on ESDE are reported on WP6 reports.

6.11 WP3-16 EZ_Chains Fleet Architecture

Levels	Functional
Require	Interface with autopilot, communication link with other drones in fleet
Provide	Mission planning and monitoring for fleet missions, Flight planning and guidance taking into account the other UAVs
Input	Mission status, periodic updates of other UAVs, UAV position and attitude from autopilot
Output	Mission updates, periodic updates on current UAV, commands to the autopilot
C4D building block	Mission Management, Flight Guidance, Flight Planning
TRL	7

6.11.1 Detailed description

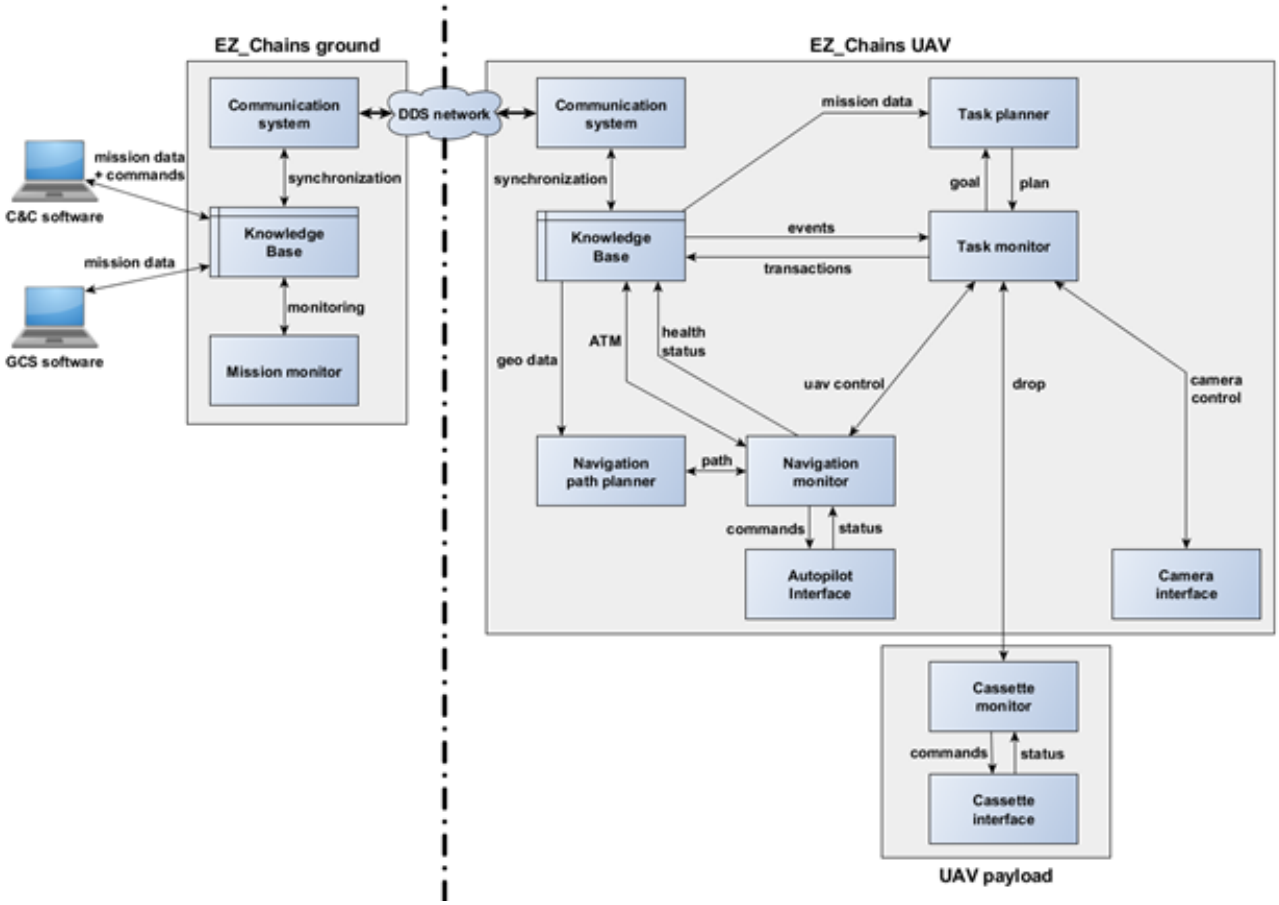


Figure 70: EZ_Chains general architecture

EZ_Chains is a generic architecture that allows agents to perform collaborative mission as a fleet. The architecture is built for distributed decision, and each agent relies on its (synchronised) copy of the knowledge base to choose the next steps. This knowledge base contains the current mission status, updated with transaction sent by the agents of the system (the transaction is validated by the ground segment, centralising the consistency process) and the status of each agent, updated periodically.

EZ_Chains can be seen as three main layers, mission management to handle the highest level of information, flight management to handle the flight while ensuring coordination with the rest of the fleet, and finally the low-level interfaces to the autopilot and payload.

The mission management is composed of two main components, the first is the task planner that computes the next best actions to achieve based on the current mission status and actions planned by the other agents. The second component is the task monitor that ensures the execution of the chosen actions, it triggers the other components. It is also responsible of the sanity of the agent, it uses watchdogs to verify the status of the different components and also detect important changes in the mission parameters that forces to reconsider the current planned actions.

The management of the flight is also composed of two main components, similarly a planner and a monitor. The path planner ensures the computation of trajectories that avoid the geofences but also reduce the intersection with the other agent trajectories. The navigation monitor triggers the path planner when requested by the task monitor (to perform a given action), but also handles the air traffic management internal to the fleet. It books airspace as segments of the trajectory as the agent progress over its trajectory. It allows to prevent collision by stopping one of the agents that would try to use or crossed an airspace already booked. The navigation monitor also verifies the validity of the trajectory

during its execution. Indeed if the mission status changes in the knowledge base, due to the addition of a new geofence for instance, the navigation monitor may request a new trajectory. Finally the navigation monitor talks to the autopilot for the agent to perform the planned trajectory.

6.11.2 Contribution and Improvements

EZ_Chains architecture described above has been designed to allow UAVs to perform collaborative missions. The work carried out during C4D is twofold: (i) generalise the architecture to work with new components and demonstrate its genericity, (ii) extend it to support mort type of agents than just UAVs. The description in the previous section uses the notion of agent (instead of only UAV) to demonstrate that the concepts we use in the architecture actually extend to other types of agents, which demonstrates that the architecture is mostly ready for new types of agents at a formal level.

In order to ease the work aimed at in C4D, we used the first and second year to integrate new missions and tasks for UAVs. Making the system capable of using a fleet of heterogeneous UAV. It allowed us to focus on realising the demonstration for the UC3 D1 (Logistics, dropping sensors). We aim at using the third year to work on integrating new types of agents and components.

Integrating new types of missions, already required to define new types of agents in the knowledge base. We now have a hierarchy of agents, for instance we have UAV>Dropper, UAV>Surveillance and GCS. To support different agents, at different level of hierarchy, required to improve the internal structure, the relation between the mission status and the periodic updates sent by the agents.

Once the knowledge base can support different types of agents, the task monitor must take it into account when following the mission progress before triggering the task planner. For instance, in the UC3 D1 demonstration, due to regulatory limitations, we had to ask the fleet to trigger a landing when the surveillance UAV declared its trajectory back to base due to low battery. New behaviour must be created when new hierarchies of agents are added. In the future we would like to add, weather stations as part of the system, they would also require the triggering of conservative actions when the weather becomes too bad.

The task planner is improved to allow new mission decomposition, indeed each new agent has its own goal and types of actions it can achieve. For instance, again with the surveillance UAV, it required the definition of new goals called Point of Interest (POI), where the UAV had to go survey the area border and cycle though all those POI. The definition of those POI, a goal only corresponding to the surveillance UAV, had to be incorporated in the knowledge base. Additionally the task planner, for the surveillance UAV, could not reason with the strategy used by the droppers, it required to redefine how such a mission is completed.

6.12 WP3-19_1 – Hyperspectral Payload

Levels	Functional
Require	Power, GPS coordinates, data & control channel to ground controller
Provide	Data analytics & spectral data
Input	Start/end capture signal, exposure time, GPS coordinates, system time
Output	Current captured hyperspectral frame, classified images, data analytics, spectral hyperspectral data
C4D building block	Hyperspectral payload
TRL	6

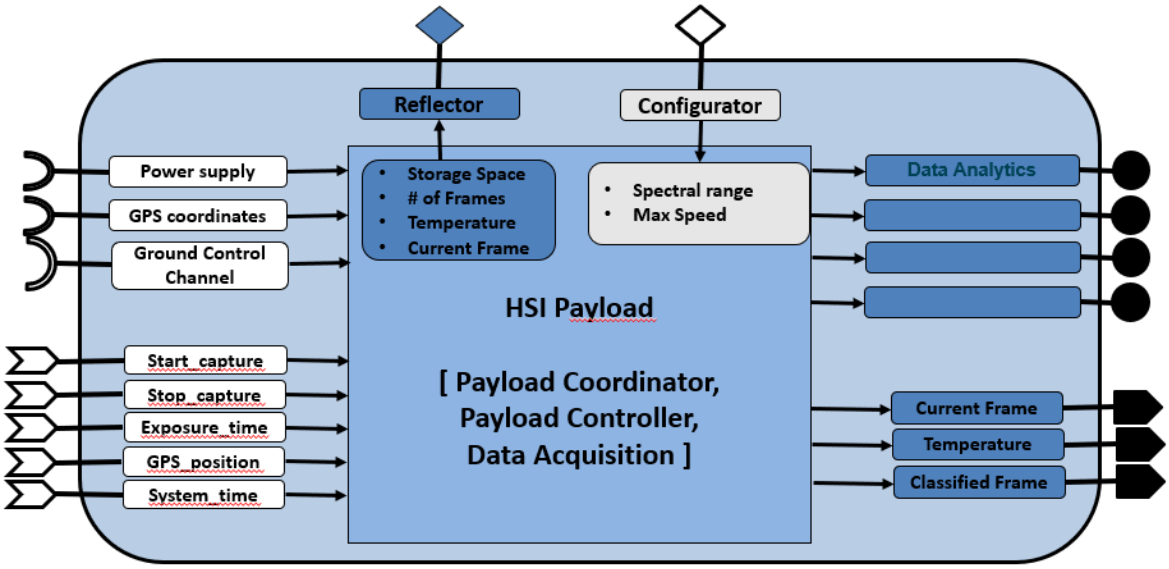


Figure 71: Building Block diagram

6.12.1 Detailed Description

Essentially, the Hyperspectral payload captures hyperspectral images, tags the images together with GPS coordinates, stores the images locally, processes the data to provide certain analytics, and sends the raw and classified/processed images to ground controller.

The hyperspectral payload can be coupled together with a certain drone system. An example of the integration is shown in the figures below. The integration is on three aspects: physical connection, control interface and the data connection between drone and payload. The physical connection is usually done via a gimbal. The control connection is done via a serial port over which commands can be sent to the payload to set the relevant parameters for the sensors in the payload. The data connection is done via Ethernet/HDMI connection, where the drone controller can request specific frames/info from the sensors that can eventually be sent to a ground controller. This information can also be used to synchronize between payload data and other sensors (e.g., GPS) or other payloads connected to the drone. The images from this sensor are unique in that they can provide hyper/multi-spectral images of up to 40 spectral bands in the range 450-900 nm.

6.12.2 Contribution and Improvements

Currently, there are no real lightweight hyperspectral UAV cameras which have more than 4/5 bands. Such a camera would be a real breakthrough in the domain of UAV precision agriculture. Parrot's sequoia multispectral camera with about 4-5 spectral bands is the leading state of the art in this domain. However, with 4-5 spectral bands only simple agriculture indices like NDVI can be extracted. Tetra cam's 3-filter camera or multi-camera systems supporting up to 12 bands are other alternatives. However, multi camera systems lead to much more bulkier systems with additional complexity of software to register images from different cameras to obtain the same spatial field of view, which could potentially lead to loss in image quality. For our target applications more spectral information would be required (>10 bands in VISNIR) to provide accurate diagnostic and actionable information. Our proposed camera can enable such applications. Headwall's micro-hyperspec is another camera intended for UAV platforms, which uses conventional grating-- based solutions for the spectral unit. This leads to a bulkier camera than our proposed solution, making this unsuitable for lightweight drones. Micro-hyperspecs cameras can weigh up to 1kg or more, making this perhaps more suitable for larger drones/UAVs.

Compared to other multispectral payload systems, this system has a significant improvement in the number of spectral bands, typically from 5-10spectral bands to up to 40 spectral bands. This enables

higher precision and accuracies in current inspections and also enables new inspection methods (e.g., inspection of soil quality, which conventionally would have required sending the samples to a lab).

6.12.3 Design and Implementation

The core design of the payload and its integration with the drone system is shown in Figure 72.

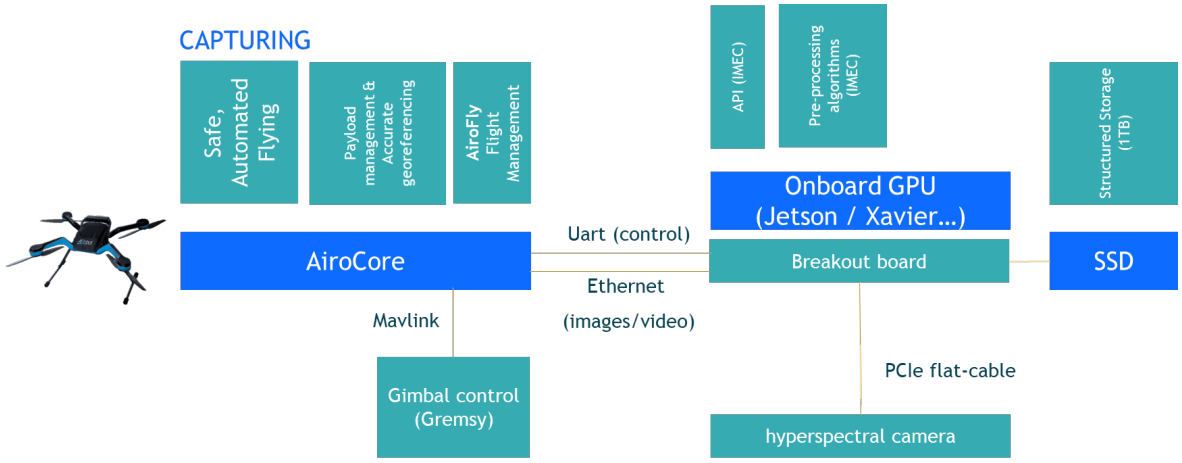


Figure 72: System architecture of UAV payload with compute enabled system

A first prototype payload has been built by Airobot to be able successfully perform first data collection flights. The pictures below show the integration on the Airobot Mapper drone and of the first test flight.

Figure 73: Prototype payload on Airobot Mapper

As a second prototype, IMEC-BG has implemented a second iteration of the payload. The payload consists of two multispectral sensors in the spectral range 470-900nm, a jetson Tx2 to enable onboard computation and about 1TB of storage to collect spectral data during flight. In addition to the payload integration with a drone as described by Airobot, IMEC has done initial integration (both hardware and software) of this payload with a DJI M600 drone. This integration was done to show the modular and flexible aspect of our payload and data acquisition software blocks. Current software development for this version of the prototype has two parts (1) firmware/acquisition software running on the payload/jetson system and (2) ground controller software running on a tablet which is connected to a drone controller. The key functionality of firmware block is to capture the data from the two multispectral sensors and perform initial pre-processing steps and store them on onboard disk. The key functionality of the ground controller software is to enable user to control camera parameters and to provide a live preview of the images from spectral sensors.

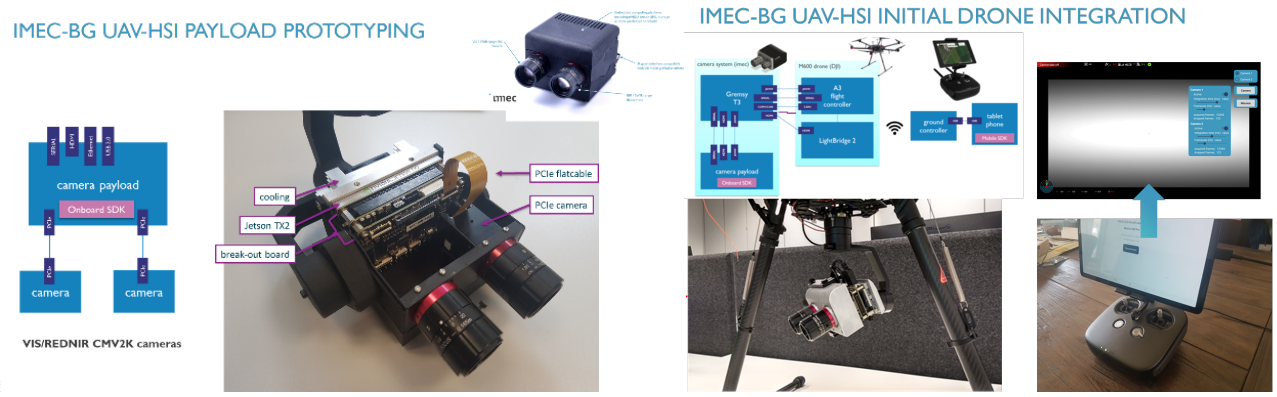


Figure 74: Prototype IMEC payload

Finally, Airobot has been working together with IMEC on working out the detailed design to integrate their payload. The mechanical, electrical and software interface has been defined. To speed up the development a setup has been created so that the software development can be done without needing physical access to the drone.

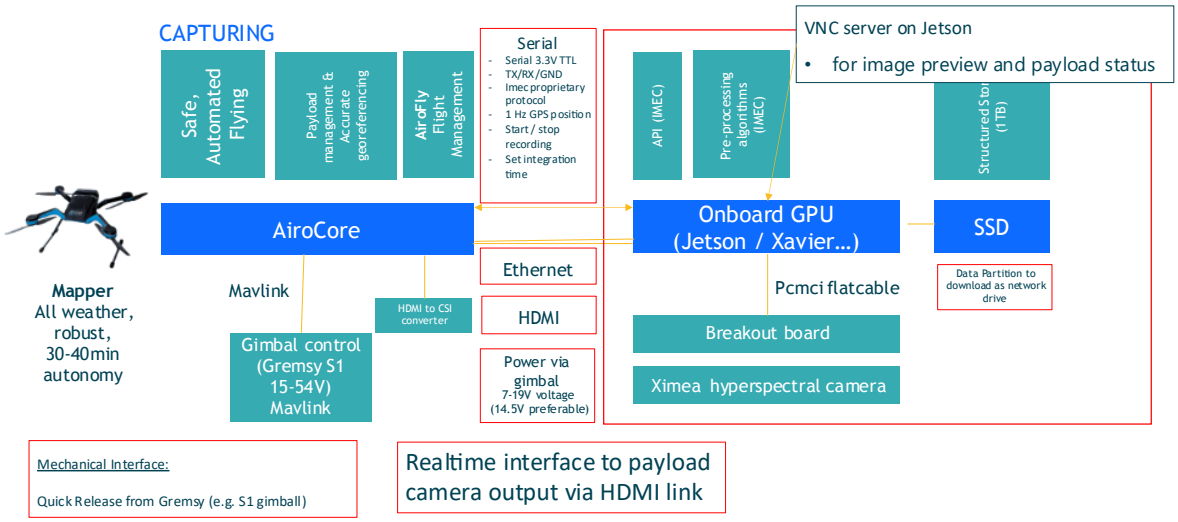


Figure 75: Architecture of interface with IMEC payload

6.13 WP3-19_2 – Hyperspectral Image Processing

6.13.1 Detailed Description

The hyperspectral imaging (HSI) pipeline is a system that processes and analyses the hyperspectral data originating from the hyperspectral payload developed by IMEC. It supports a drone system in many aspects. First, it provides an API to easily access radiometric- and reflectance-corrected hyperspectral images. Second, it implements tools to compress the data in a lossless manner to limit storage needs. Third, it provides a fully automated way of stitching the images together into a georeferenced orthomosaic. Finally, the HSI pipeline also accommodates an API to easily interpret the data by semantically labelling the data (i.e., assigning a class label to each pixel of the image) also known as classification. The latter is done in a semi-supervised way and supports many different use cases, including applications on surveillance and inspection.

In the context of the reference architecture, the hyperspectral imaging (HSI) pipeline supports payload data analytics block.

The second innovation involves data-analytics on images to detect imperfections, like corrosion, deterioration of paint, real-time or offline. The hyperspectral imaging (HIS) processing pipeline that we are developing is depicted below. It is made up of three main modules: pre-processing, on-board analysis, and post-processing.

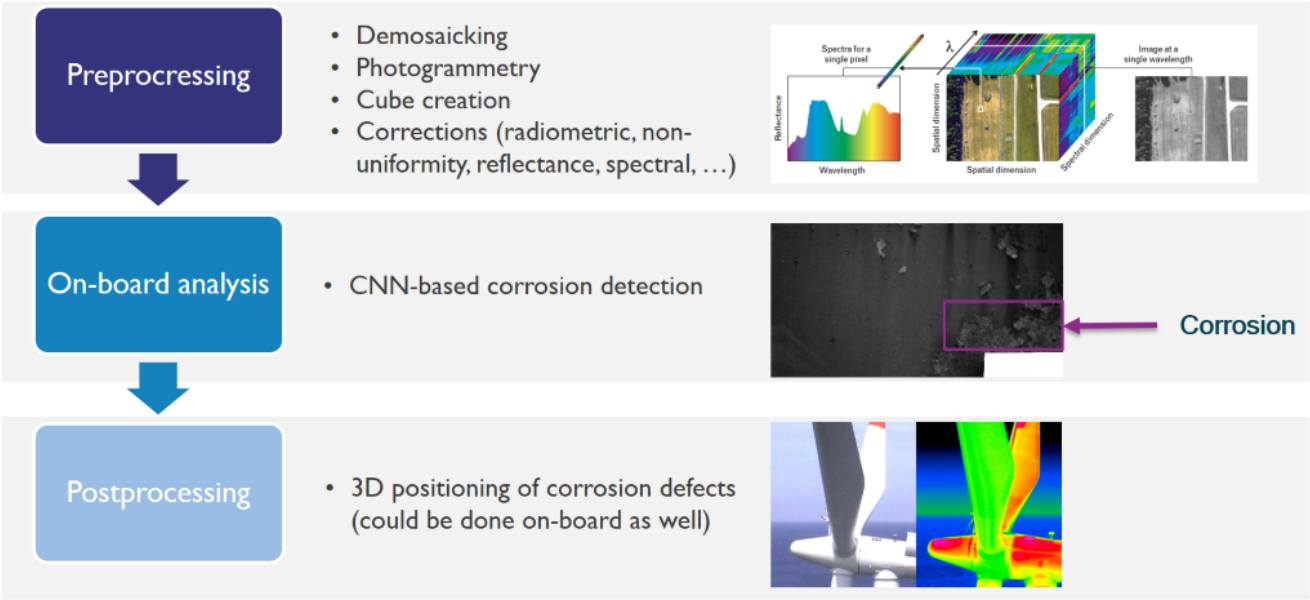


Figure 76: The hyperspectral imaging processing pipeline

The hyperspectral images are finally stitched together, and a 3D model is constructed. To that end, our photogrammetry (or structure from motion) software was further enhanced in a way that it is better suited to cope with repetitive patterns in the environment. Photogrammetry in such challenging conditions generally fails using existing, commercial, software packages. Experiments with our improvements show that our pipeline is a lot better suited to cope with repetitive patterns in the scene.

The on-board analysis is dealing with the detection of degradations, i.e., corrosion, of the infrastructure. To this end we are developing AI-algorithms (CNN-based) that exploit both spectral and spatial futures to identify the regions where corrosion appears.

6.13.2 Contribution and Improvements

Currently, there exists many hyperspectral image processing algorithms (e.g., de-mosaicking for mosaicked sensor layouts or deep learning based detection, segmentation or classification). However, they are developed and designed for (off-board) PC platforms and are totally not optimized for the IMEC's hyperspectral dual camera payload, integrated nor run on embedded hardware platforms such as the Jetson TX2 board.

Classic deep learning frameworks rely on massive amount of annotated data, over which we will not dispose (and are not able to collect ourselves). Therefore, we rely on recently developed few-shot learning techniques, which are trained with only a limited number of annotated samples. However, the robustness under various noise conditions and few-shot learning performance needs further research. In the case of hyperspectral imaging, this will also impact the acquisition: e.g., the varying incident sun light will create different appearances of the same physical material. Proper normalization procedures are needed to be developed.

The entire pipeline is made up of three main modules: pre-processing, on-board analysis, and post-processing, as can be seen in the figure above.

The on-board analysis is dealing with the detection of degradations, i.e., corrosion, of the infrastructure. To this end, AI-algorithms (CNN-based) are developed. A result of the current corrosion detection

algorithm is depicted in Figure 40. The purpose is to implement these algorithms on the Jetson TX2 board for them to be executed in real-time. This way the corrosion parts can be identified and located online, and the drone can be instructed to fly towards the most degraded areas in order to limit fly-time and avoid those relevant areas remain uncaptured. The final goal is to achieve automatic HS-image-based detection and quantification of corrosion using AI technology with an accuracy of 80% compared to human inspections.

6.13.3 Design and Implementation

A schematic overview of the dataflow for the HSI processing pipeline is depicted on the figure below. A raw HS image acquired by the payload is sent to the preprocessing module where several corrections are carried out, de-mosaicking is taking place and a HS cube is created. The corrected HS-cubes are then fed to the offline analysis module, where traditional classification algorithms can generate a labelled HS image. Alternatively, the labelled HS image can be fed to the dataset generation and model definition submodule in order to set up a CNN training environment. After the training, the CNN model is used in the online analysis module where inference can be applied on another HS cube. In parallel, position and orientation information (provided by the GNSS and IMU) together with the corrected HS cube is fed to the photogrammetry module to generate both, the HS point cloud and the camera poses. Eventually, both are fed to the inference submodule to create labelled HS images and a fully georeferenced, labelled HS point cloud.

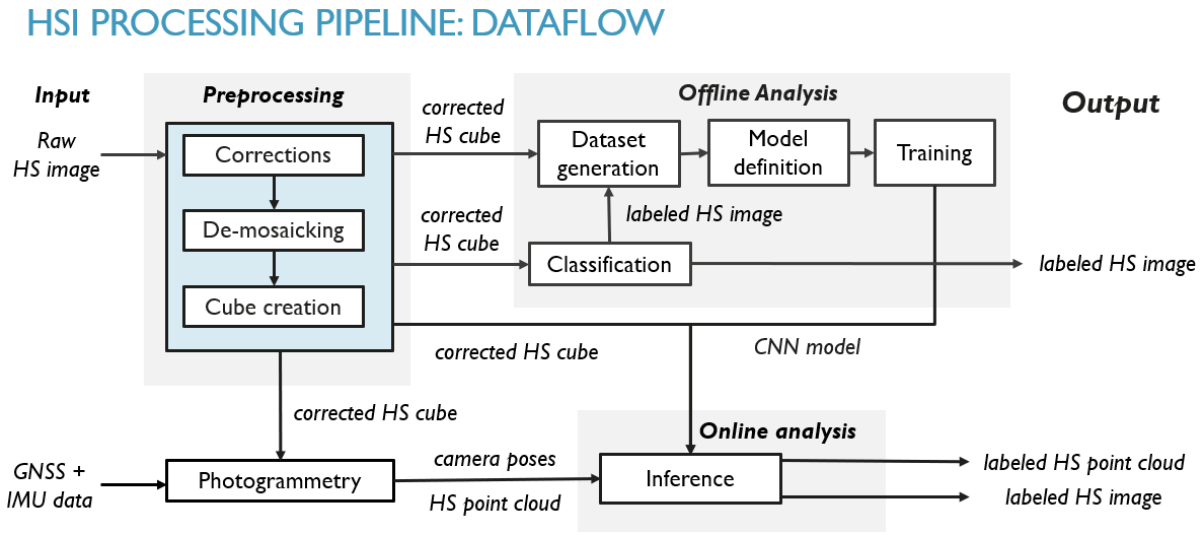


Figure 77: The hyperspectral imaging processing pipeline

The HSI processing pipeline can be used to develop a state-of-the-art detector for corrosion on infrastructure or pollution in soil. A few collected HS cubes can be classified using traditional techniques to serve as a training dataset (after minor manual verification). A CNN model can then be trained and used to classify additional images using the inference module. Together with the output from the photogrammetry a georeferenced and labelled HS point cloud can be obtained.

The deep learning framework was evaluated using data captured near a gate that was subject to corrosion. Experiments demonstrated that we achieved an accuracy of 95% to 98% (depending on the CNN model) for classifying corroded areas (i.e., pixels) within the HS images. Moreover, we successfully tested the deployment of the network on a Nvidia Jetson Xavier NX. The inference to generate a labelled HS Image of 254x510 pixels can be conducted in a handful seconds on the Xavier NX. This allows for online processing and hence real-time applications, e.g., to instruct the drone to fly towards the areas of interest (indicated by the labelled image).

Using some of the prototypes from IMEC & AIROBOT, initial work on corrosion detection shows promising results as indicated below. The focus for the next period will be to reorient the instantiation of these algorithms towards soil quality instead of corrosion.

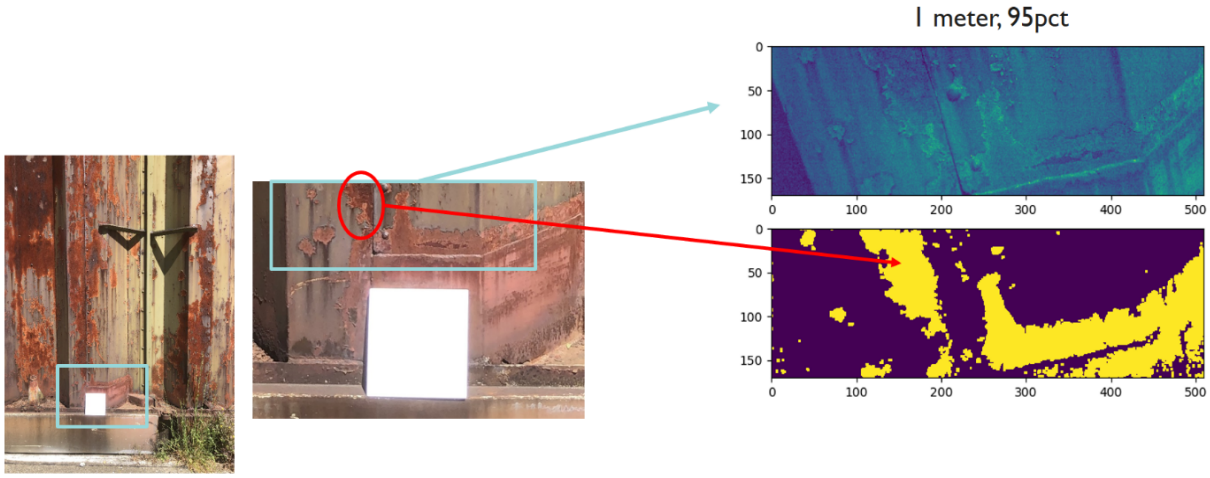


Figure 78: Result of the corrosion detection algorithm at 1 meter distance: purple denotes back ground, yellow denotes corrosion

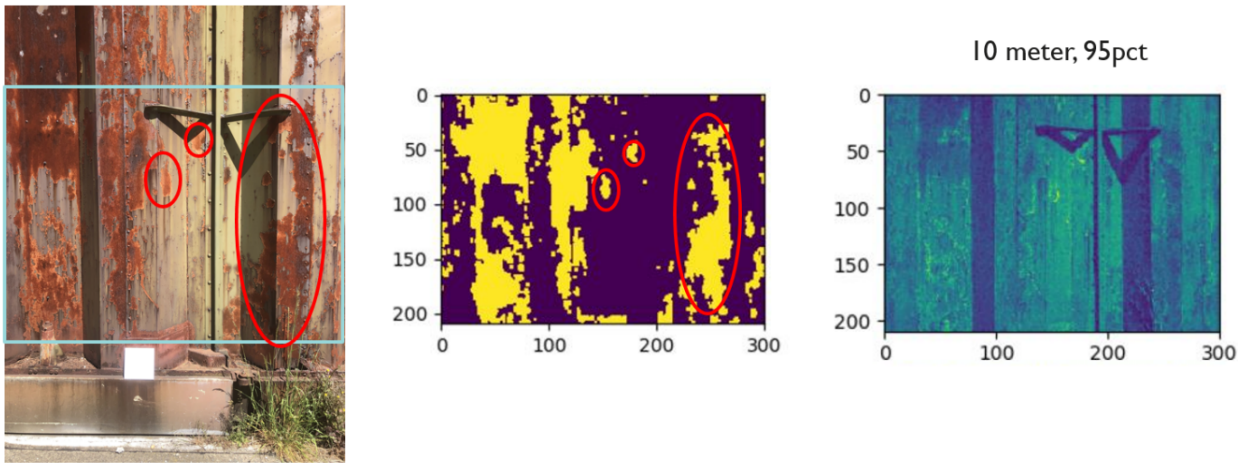


Figure 79: Result of the corrosion detection algorithm at 10 meter distance: purple denotes back ground, yellow denotes corrosion

6.14 WP3-20 Multi-sensor positioning

Levels	Functional
Require	
Provide	GPS spoofing message detection using ML technique , position computation in absence of GPS based on sensors data
Input	Sensors data (gyro, accelerometer, magnetometer, GPS)
Output	Binary data to indicate absence or not of spoofing; X,Y,Z coordinates estimation if spoofing is detected
C4D building block	Continuous flight in without GPS information, spoofing detection
TRL	4

6.14.1 Detailed Description

Px4 autopilot is a system involving open source hardware and software freely available to everyone under BSD license. A Pixhawk board is developed according to the Pixhawk standard and conforms to the Pixhawk standard requirements. The Holybro board based on CPU STM32F765 is one of these supported HW and has chosen as the main board for drone controller.

Holybro board is therefore responsible for piloting the drone, being equipped with an IMU sensor to control its angle, acceleration and orientation. The path that the drone must follow is established in advance and is followed by the drone thanks to the GPS receiver.

Detection of GPS spoofed messages is done with ML techniques, so a dedicated board is used to perform in parallel this task. There is also the need to collect debugging data and monitor the state of the execution so a Raspberripy 4B board is used to accomplish this.

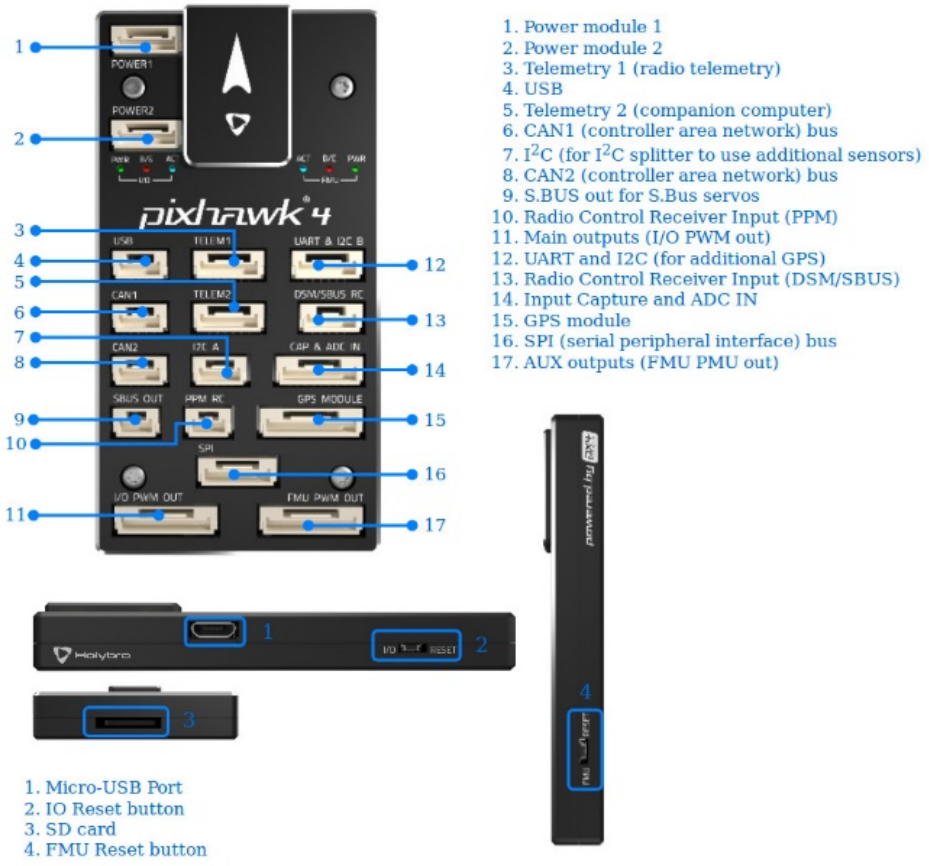


Figure 80: A Pixhawk board

Figure 81: A Raspberry Pi board

Raspberry Pi has the ability to interact with the outside world and has been used in a wide array of digital maker projects, from music machines and parent detectors to weather stations and tweeting birdhouses with infra-red cameras.

The computational power is given by a quad core CPU (ARM) and its main memory (8 GB), which allows to manage the amount of necessary data to execute ML algorithms giving an answer in an acceptable time.

Raspberry Pi OS (Raspbian) is the recommended operating system for normal use on Raspberry Pi, it is Linux based and permits all useful canonical operation such as remote connection (ssh) and log management.

6.14.2 Contribution and Improvements

Currently commercial drones do not have the ability to avoid certain attacks on the GPS for several reasons: there are different strategies to avoid a spoofing attack, many of these include expensive hardware devices both from the point of view of energy and space resources therefore not suitable for mounting on a UAV.

The proposed solution combines the already strongly supported family of opensource drones with the potential of mini computers such as the raspberry Pi, and an attack detention strategy based on a supervised machine learning technique.

This has several advantages:

- decoupling between flight management and GPS flow analysis,
- different power lines for the boards,
- continuous improvement of the classifier performance without affecting anything else of the system…

6.14.3 Design and Implementation

In order to cooperate between the two devices, Holybro and Raspberry are connected via a UART serial interface.

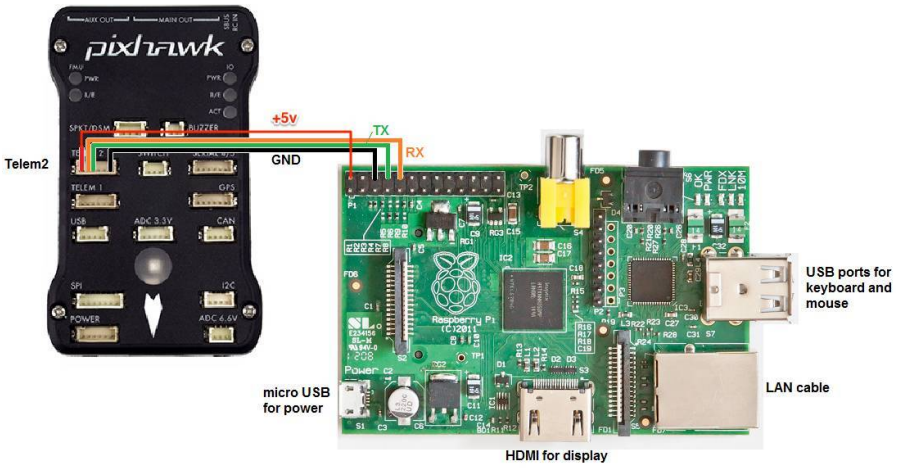


Figure 82: Holybro and Raspberry communication

Both devices are tuned to the same transmission frequency and a data exchange algorithm is established to synchronize the devices on read/write.

The Px4 basically has the task of recovering data, while the Raspberry is equipped with:

- a SVM (Support Vector machine) classifier previously trained on a private dataset of about 6000 data;
- a custom implementation of the extended Kalman filter for estimating the next geographic point without using GPS coordinates.

Basically, the Px4's task is to collect a quantity of data coming from the sensors over a period of time and sending them to the Raspberry Pi. On the other hand, the Raspberry reads the received data, classifies it using an SVM classifier, in case of spoofing it calculates the correct coordinates using a Kalman filter. Finally, it sends back the answer to Px4 which decides whenever to overwrite or not its GPS position based on the response.

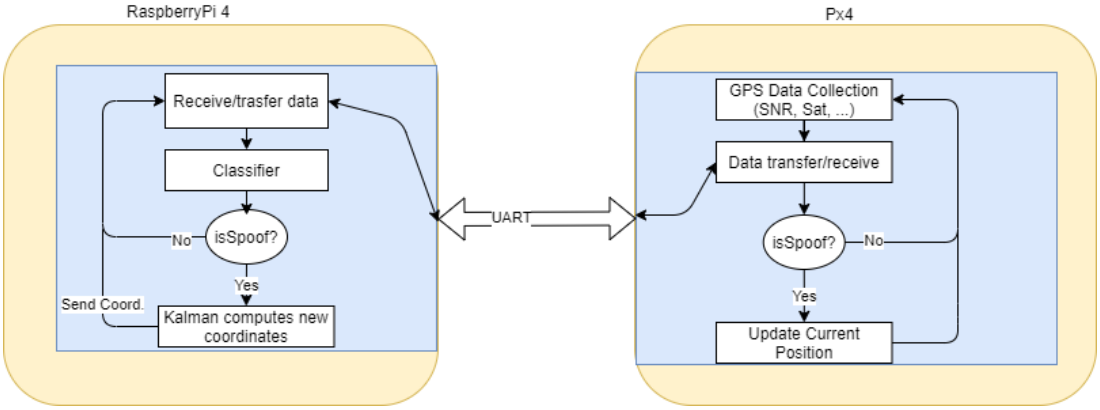


Figure 83: Global behaviour of the anti-spoofing function

6.15 WP3-22 – Onboard Overlay Compute Platform (OOCP)

Levels	System
Require	Application definition and FPGA-based System-on-Chip
Provide:	Accelerator-rich Overlay for FPGA-based System-on-Chip
Input	Application Specific HW Accelerators description in HDL, or HLS, or using WP3-28 Methodology
Output	Synthesizable Overlay for FPGA with integrated Application Specific HW Accelerators
C4D building block	The methodology is generic and applicable to host HW accelerators for different tasks and scenarios. With respect to C4D, it could be used to implement HW accelerators related to perception, actuation, flight-control, payload management or data management.
TRL	4

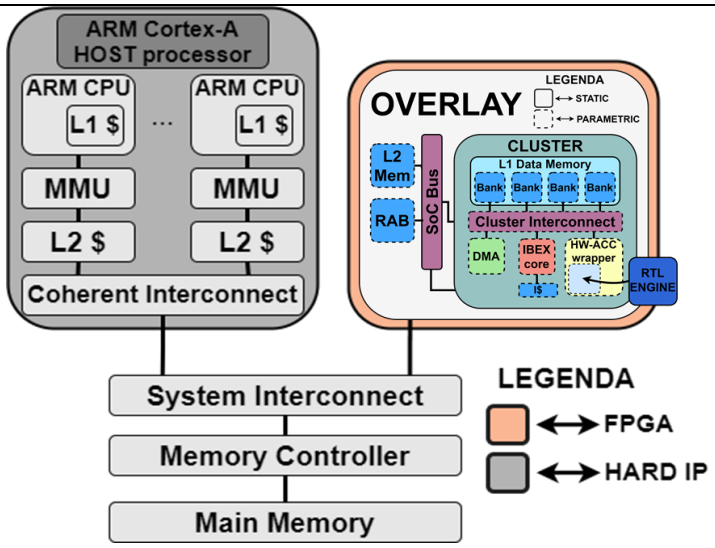


Figure 84: Building Block diagram for WP3-22

6.15.1 Detailed Description

The **Onboard Overlay Compute Platform Design Methodology** is composed of two main contributions:

- **Onboard Overlay Compute Platform (OOCP).** The OOCP is an evolution of the HERO architecture targeting specifically FPGA acceleration on FPGA-based heterogeneous systems-on-chip (HeSoCs).
- **Onboard Overlay Development Kit (OODK).** The OODK contains tools and library for the automatic integration, programming, and offloading to Application Specific Hardware Accelerators.

The **Onboard Overlay Compute Platform Design Methodology** is generic and applicable to host accelerators for different tasks and scenarios. With respect to C4D Drone Reference Architecture, the OOCP could potentially be used to host and integrate on a FPGA-based SoC different HW accelerators (e.g., perception, actuation, flight-control, payload management or data management).

6.15.2 Specifications and contribution

Increased demand of autonomy on UAV requires adequate on-board smart sensing and computing capability to support safe decision making, based on large amounts of data that is sensed, analysed and understood in real-time. The capability of flexibly defining parallel, non-Von-Neumann processing

logic and custom memory hierarchies, all within contained power envelopes, makes the FPGA-based heterogeneous systems-on-chip (HeSoCs) an ideal candidate for implementing onboard compute platforms for UAV.

Withing the C4D project, we are developing an **Onboard Overlay Compute Platform Design Methodology** for FPGA-based HeSoCs and leverages *soft-cores* for flexible control of user-defined, *application-specific accelerators*. Different accelerators can flexibly operate and re-configure their operation without the costly need for host intervention, thus avoiding significant performance degradation. Normal accelerator operation and accelerator reconfiguration can both be achieved via standard computation offloading from the host CPU to the soft-cores (e.g., OpenMP v4.x+). The user can rely on any methodology of his/her/their choice to design the accelerators (e.g., by WP3-28 C4D components or Vivado HLS). Moreover, the **Onboard Overlay Compute Platform** includes dedicated logic (the wrapper) to provide *plug-and-play* HW/SW integration of such accelerators developed within C4D WP6 activities.

6.15.3 Design and Implementation

Figure 84 shows an overview of the proposed **Onboard Overlay Compute Platform (OOCP)**. The Onboard Overlay is based on The Parallel Ultra Low Power Platform (PULP)[11], and particularly on HERO[12] is an open-source research platform based on FPGA emulation of PULP-based heterogeneous many-core systems. HERO can be instantiated on FPGA SoCs like the Xilinx Zynq family.

HERO constitutes a convenient starting point to implement the Onboard Overlay Compute Platform: being conceived as a many-core architecture, HERO naturally complies with some of the basic requirements to build an accelerator-rich design, most notably the cluster-based design and the multi-bank shared memory design. However, HERO clusters are designed for general-purpose (or, at best, signal-processing oriented) parallel execution and thus have substantial limitations in the context of FPGA hardware acceleration that we target. HERO uses the FPGA merely as a medium for emulation of projects meant for IC realization.

The proposed Onboard Overlay Compute Platform uses the FPGA as a target for acceleration. For an overlay to be an efficient and convenient solution, it should offer: (i) System-level design capabilities; (ii) transparent accelerator integration flow; (iii) streamlined resource usage.

The Onboard Overlay Compute Platform is designed to be light (in terms of resources utilization) and configurable. The OOCP features a customized number of clusters, and each cluster is composed of one (or more) RISC-V IBEX [13]core (RV32IMC), an instruction cache, a DMA, and a multi-ported multi-banked L1 Data Memory (scratchpad). The cluster can host one (or more) Application Specific Accelerators that can interfaced to the shared L1 Data Memory thought a *wrapper*.

6.16 WP3-26 Droneport: an autonomous drone battery management system

Levels	Functional
Require	Communication service to obtain data from the assigned drones and master controller
Provide	Autonomous battery management and robotic battery exchange
Input	UAV mission, UAV battery status
Output	Available battery resources
C4D building block	Health management, Mission management
TRL	4

[11] https://pulp-platform.org/
[12] https://pulp-platform.org/hero.html
[13] https://github.com/lowRISC/ibex

6.16.1 Detailed Description

Droneport (DP) is a system for autonomous drone battery management. Droneport complies with a reference architecture, dealing with drone power management, more precisely battery management. Regarding the reference architecture, the DP consists of a battery management unit for charging and storage, a data link to the drone, and telemetry data.

Power and storage management building blocks are implemented in Droneport, with a datalink connection to the outside world. Users can use telemetry data via MAVLink protocol to monitor the state of the DP. The DP acts autonomously, it monitors the state of the batteries, controls the charging and battery exchange process provides necessary navigation information for drone landing and broadcasts the status of the remaining batteries.

6.16.2 Contribution and Improvement

The Droneport aims to extend the UAV mission by autonomously changing the battery of the drone. It behaves as a standalone autonomous unit that is wirelessly interconnected with one or many UAVs. It is compatible with other C4D components. it uses an open MAVLink protocol with defined messages for communication. At the beginning of the project, there was only a notion of how to extend the UAVs mission. We are targeting TRL 4 (the prototype) at the end. The device would autonomously manage the batteries and cooperates with selected drones.

6.16.3 Design and Implementation

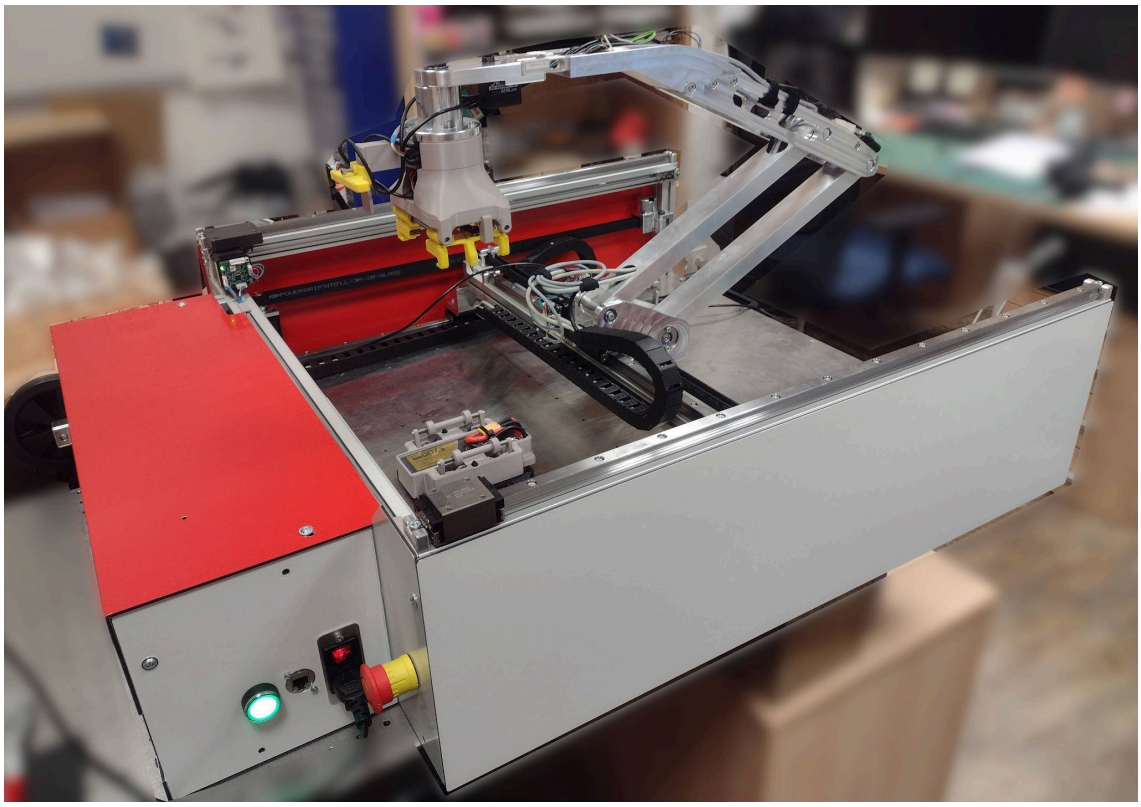


Figure 85: Droneport manipulator arm with a custom gripper

The Droneport development is done together with SmartMotion company. Our team collaborates together on hardware and software design. The DP will be a unique device that will be able to communicate with various UAVs. For better support of battery exchange, the DP contains a newly designed gripper and a special battery holder.

Droneport hardware consists of

- Drone landing place with ArUco based landing markers
- Uniquely designed robotic manipulator
- Uniquely designed gripper
- Battery management unit for charging and storage
- Wireless communication to the drone/swarms

The whole device will communicate wirelessly over MAVLink protocol [29] with other devices in the network. The device is controlled using Linux based PC with REXYGEN [34] as real-time control software.

Droneport SW architecture consists of

- Drone to DP communication protocol (defined MAVLink messages)
- Drone landing assistant - vision navigation to ArUco markers
- Battery exchange system control - manipulator controller
- Charge control - communication interface between charges and battery slots
- Battery management software
- DP software provides an open API for interoperability with various drones flight controllers.

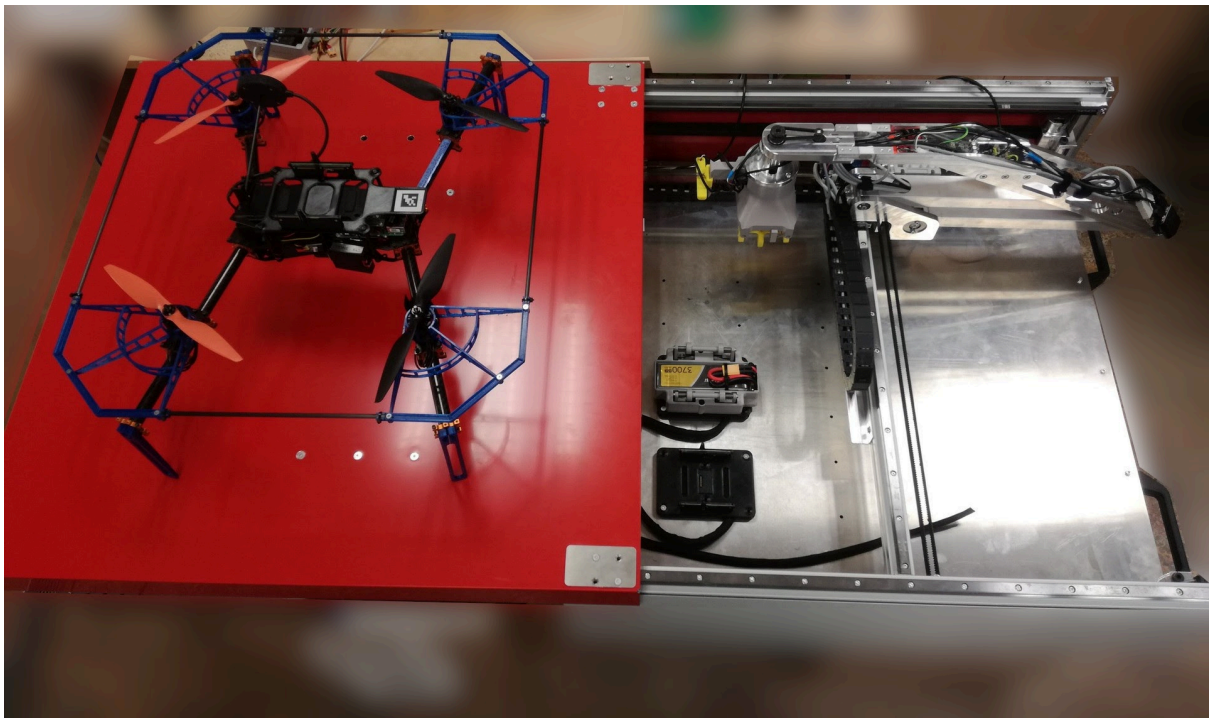


Figure 86: Top view of the Droneport with landed drone

6.17 WP3-28 – Accelerator Design Methodology for OOCP

Levels	System
Require	Application definition and FPGA-based System-on-Chip
Provide:	Ready-to-use reconfigurable HW accelerator
Input	Dataflow application specification(s) HDL actor definition(s) Communication protocol Target architecture
Output	Multi-dataflow network Coarse-grained reconfigurable accelerator RTL

	Co-processor RTL Programming tables
C4D building block	The methodology is generic and applicable to generate accelerators for different tasks and scenarios. With respect to C4D, it could POTENTIALLY be used to implement HW accelerators related to perception, actuation, flight-control, payload management or data management.
TRL	From 3 to 4

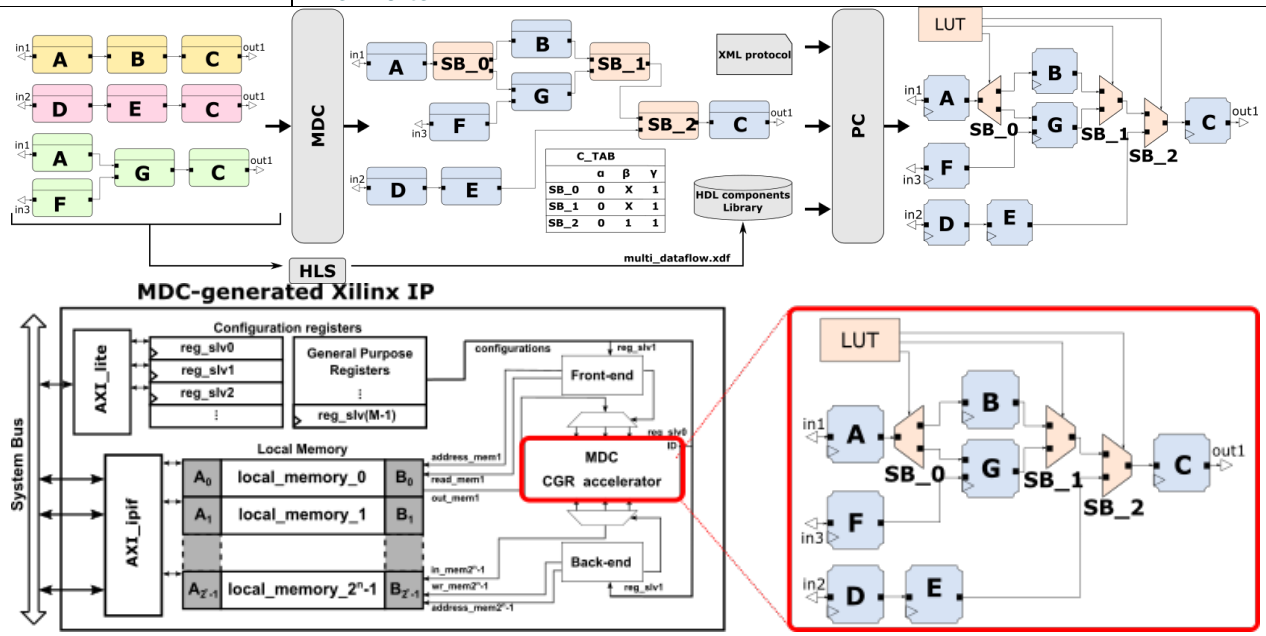


Figure 87: Building Block diagram for WP3-28

6.17.1 Detailed Description

Although FPGA technology has the potential to satisfy the many performances, energy and predictability requirements of drone systems and applications, FPGA development is notoriously a complex task.

To deal with this problematic, the baseline feature of this component revolves around the composition of coarse-grained reconfigurable HW accelerators (CGRA) starting from a set of dataflow applications. The baseline feature involves two main components:

- **Multi-Dataflow Generator (MDG)**: it merges together different dataflows into one unique reconfigurable multi-dataflow by the insertion of switching modules. Currently, two merging algorithms are supported: empiric and Moreano. The former is more suitable for non-recursive dataflows but less optimized than the latter.
- **Platform composer (PC)**: it derives the RTL description of the CGRA from the multi-dataflow. It requires the user to define the communication protocol between actors in hardware (XML) and the RTL description of the actors involved in the dataflows (HDL Components Library, HCL).

This component also provides an automatic coprocessor generation, which automatically embeds the generated CGRA into a ready-to-use Xilinx IP. The user can choose among different options:

- **Processor**: soft-core (Microblaze) or hardcore (ARM)
- **Processor-Coprocessor coupling**: Memory-mapped or FIFO-based
- **Direct Access Memory Module**: enable or not the usage of DMA

6.17.2 Specifications and contribution

The purpose of this component is to provide a methodology to generate application-specific HW accelerators that can be directly plugged in the final system. Additionally, this component automatically

enables coarse-grained reconfiguration capabilities, which enables the capability of having a HW accelerator that can work at different working points (i.e. trade-offs among Quality of Service and Energy consumption) or functionalities (i.e. different implementations related to the same C4D building block).

Regarding the contribution associated to C4D, this component will be extended to be able to automatically generate plug-and-play coarse-grained reconfigurable HW accelerators that can be used by WP3-22 component. To do so, a unified methodology will be provided so as to combine the FPGA overlay provided in WP3-22 with the CGRA generation supported by this component.

6.17.3 Design and Implementation

Considering that the HW accelerators that are generated using this component are application specific, the required steps to generate the multi-dataflow networks and its associated CGRA are the following:

- Define the three inputs that are required:
  - Implement the task(s) to be accelerated using a dataflow approach.
  - Define the HDL version of the actors composing the tasks (manually or with HLS tools).
  - Define the communication protocol to be used inside and outside the accelerator.
- Using the MDG functionality, if more than one task has been specified, merge the tasks to be accelerated into a reconfigurable multi-dataflow.
- Depending on the target architecture, the user must select the files to be generated, that could include the accelerator RTL description and a wrapping logic surrounding the accelerator itself that could allow the user to 1) use the coarse-grained reconfigurable accelerator as a co-processor or 2) to plug the accelerator with an FPGA overlay, as in the case of this project.
- Run the automatically generated scripts to port the code to Vivado.
- Synthesize and implement it on the target FPGA device.

6.18 WP3-36_1 - Smart and Predictive Energy Management System

Levels	Functional
Require	Sensors that provide measurements and mission details
Provide	Energy trajectories to perform path tracking missions
Input	X, Y, Z coordinates of the drone and its velocities Roll, pitch, and yaw angles of the drone Initial and final position of the mission
Output	Rotor's speed (or forces)
C4D building block	Flight planning
TRL	3-4

6.18.1 Detailed Description

An energy management system is vital to optimize the energy life and the purpose of the system. It will continuously monitor important system parameters, while dealing with the varying power demands of the many aspects, the objectives of the mission and optimizing the usage of the energy. Given the initial and final positions, the objective of the component is to compute the control inputs that rule the motion and vehicle trajectory to optimize energy consumption and the computational burden of the algorithm.

The aim of this activity is to use a model-based approach to control UAV flight to minimize energy consumption. It is therefore necessary to consider the flight dynamics, battery, and energy flow and actuator models. The strategy that led to good results in some research is the optimal control and therefore this type is considered in the activity. A first objective is to improve control by making it robust with respect to model approximation errors or disturbance (for example under wind conditions).

6.18.2 Contribution and Improvements

The main contribution of the component is to provide excellent trajectories from an energy point of view for the position, speeds that are the result of an elaboration of the associated optimal control problem and analysis of the excellent results. The improvement is due to the fact that a real-time resolution of the control is not required, but rule-based strategies are required which therefore have a low computational cost. Furthermore, in collaboration with other partners, they want to experiment in non-ideal conditions (presence of wind)

6.18.3 Design and Implementation

The designed energy management system will be verified and tested via Software in The Loop using software Matlab-Simulink. The reference generator will be implemented to execute a mission and the associated controller and will be tested both with Matlab-Simulink and with complex simulators in less than ideal conditions.

6.19 WP3-36_2 - AI drone system modules

Levels	Functional
Require	Dataset
Provide	Designed and implemented algorithm
Input	Plant images
Output	Disease classification
C4D building block	Data analytics
TRL	4-5

6.19.1 Detailed Description

An artificial intelligent method to classify leaf diseases will be designed and implemented. The methods inputs are images from cameras, and the output is the plant health status classification. Artificial intelligence algorithms with different characteristics will be designed and implemented with the aim of taking into account the computational cost of each, as well as the over-fitting problem and the dataset that is not always adequate, and the diagnostic performance is drastically decreased when used on test datasets from new environments (i.e. generalization problem).

A large amount of data increases the performance of machine learning algorithms and avoids over-fitting problems. Creating a large amount of data in the agricultural sector for the design of models for the diagnosis and detection of plant diseases is an open and challenging task that takes a lot of time and resources. One way to increase the dataset will be to use data augmentation techniques. It increases the diversity of training data for machine learning algorithms without collecting new data.

6.19.2 Contribution and Improvements

The improving objective of this component is to develop and test the algorithms, increase the reference dataset using basic image manipulation and deep learning based image augmentation techniques such as image flipping, cropping, rotation, colour transformation, PCA colour augmentation, noise injection, Generative Adversarial Networks (GANs) and Neural Style Transfer (NST) techniques. Performance of the data augmentation techniques was studied using state of the art transfer learning techniques both in terms of accuracy and computational cost.

6.19.3 Design and Implementation

It is developed through the Tensorflow and Python framework.

6.20 WP3-37 Video and data analytics

Levels	System
Require	On board camera processing units with sufficient resources
Provide	Object detection
Input	Video flows
Output	Images and video flows (both with bounding box to highlight target detected), metadata (es. JSON files which describe the detected information), alarms, other information
C4D building block	Perception
TRL	5

6.20.1 Detailed Description

Video analytics component (WP3-37) is a software module that implements video analysis algorithms based on Deep Learning approaches. It will be used to process RGB (mainly) and infrared (eventually) images.

This block is deputy to analyse image collected by onboard camera in order to extract relevant information, description/understanding of what has happened or what is happening. One of the most common tasks is **object detection**, that consists in recognizing the presence of a person or a specific object in an image, also detecting its position. Video analytics for object detection consists of the following four steps, as reported in Figure 88. While training phase is done offline, obviously not on the drone, the final steps, the real detection, can be done in real-time or in post processing and can be done on board or on the ground segment.

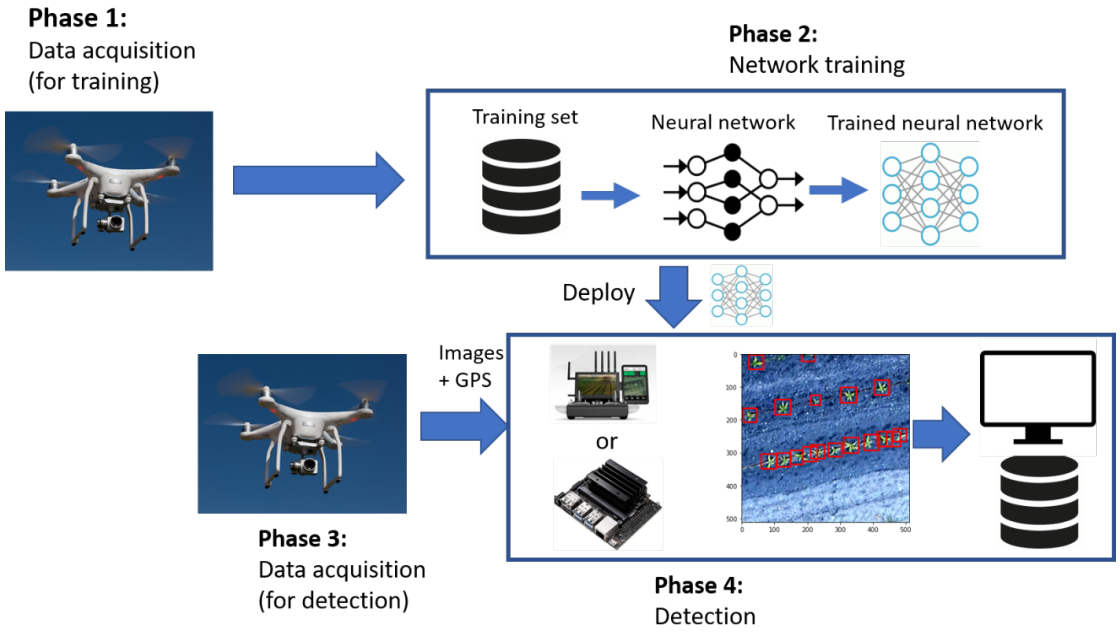


Figure 88: Steps in detecting objects from video

6.20.2 Contribution and Improvements

Neural network optimization needed to simplify its complexity. In this way computational requirements can be relaxed and therefore such networks can be deployed also on low-power edge devices. A proper training phase is needed according to the application requirements. In particular, this component will be

part of Use Case 5, and therefore will be adapted to be used in the context of the smart agriculture for plants detection.

6.20.3 Design and Implementation

Further details about this component are reported in this section. Targets are detected analysing each video frame as an individual image. The adopted technique is known as **SSD** (**Single Shot Detector**) in which the detection is done in a single stage (a single shot). Such approach is faster and simpler compared with other ones like RPNs (Regional Proposal Networks) that require at least two shots to perform the same task. In Figure 89 is reported the structure of an SSD.

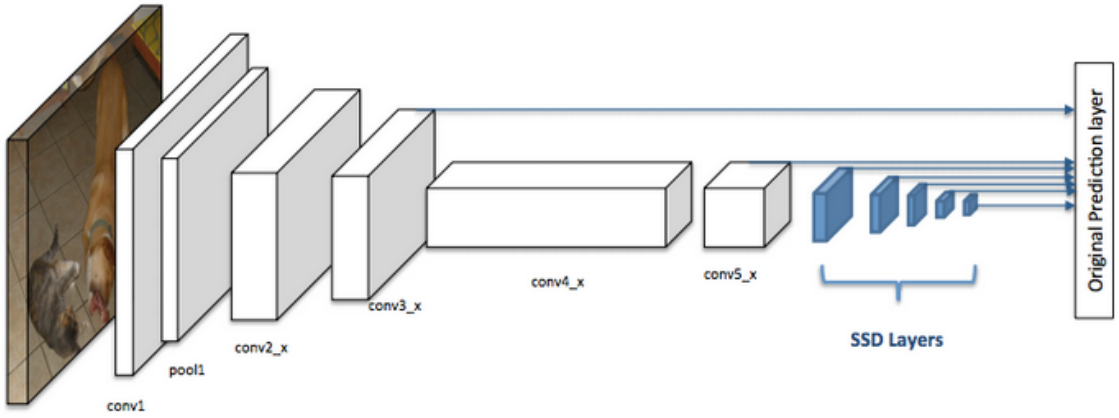


Figure 89: Structure of a Single Shot Detector

The main characteristics of an SSD are reported here below:

- **Single Shot:** this means that the tasks of object localization and classification are done in a *single forward pass* of the network
- **Detector:** The network is an object detector that also classifies those detected objects
- **Grid:** an SSD divides the image using a grid and have each grid cell be responsible for detecting objects in that region of the image.
- **Priors:** each grid cell in SSD can be assigned with multiple anchor/prior boxes. These anchor boxes are pre-defined and each one is responsible for a size and shape within a grid cell

7 Conclusion

In this deliverable, a first implementation of a drone reference architecture features is provided. Rather than providing yet another autopilot or yet another meta-model, the philosophy of this document is to allow existing frameworks to include specificities regarding autopilot implementations. Rather than closing on one solution and technology, this document opens to different framework semantics augmentation, targeting a larger adoption. One of the main focuses of the chosen specific concepts is to allow a fine grain representation of existing autopilots, and especially off-the-shelf autopilots.

This will allow current and future enabling technologies to identify exactly where and how their component can interact with the autopilot, minimizing effort and risks of interfering uncontrollably with it. We believe that doing so will allow an easy integration and customization of existing autopilots. The second main role of this effort is to leverage actual analysis and design helper frameworks in order for them to be used to toughen existing and customized autopilots. By toughening we mean be able to check some functional and non-functional properties, to raise confidence in the software, and allow to address some of the objectives that are required to qualify a system regarding standards such as the DO-178C.

This implementation is therefore an “abstract implementation”, and it was chosen instead of a single concrete one (a defined modelling language, for example) in order to leave the decision of an implementation open for developers, as long as the implementation conforms to the characteristics presented in the text.

Based on preceding deliverables presenting the building blocks of a drone system, we explore current quality standards, norms and methodologies that were developed to ensure the safety and correctness of critical systems. These concepts are fundamental to defining a suitable architecture from which critical aspects can be analysed and validated.

Furthermore, current technologies widely used in critical system development are discussed, and whose concepts will be integrated in the specification. They will allow the representation of different parts of the software as components interconnected to their surroundings without losing important information.

Finally, the components of the work package have had their description updated to follow up with current advances.